Developing Applications With

Objective Caml

和訳スナップショット: October 3, 2004

Emmanuel Chailloux Pascal Manoury Bruno Pagano

Developing Applications With
Objective Caml

Translated by

Francisco Albacete • Mark Andrew • Martin Anlauf •
Christopher Browne • David Casperson • Gang Chen •
Harry Chomsky • Ruchira Datta • Seth Delackner •
Patrick Doane • Andreas Eder • Manuel Fahndrich •
Joshua Guttman • Theo Honohan • Xavier Leroy •
Markus Mottl • Alan Schmitt • Paul Steckler •
Perdita Stevens • François Thomasset

和訳（五十音順）

五十嵐 淳・Jacques Garrigue・住井 英二郎・関口 龍郎・
玉田 嘉紀・富沢 伸行・橋本 政朋・原 耕司・
坂内 英夫・古瀬 淳・細谷 晴夫・脇田 建

Éditions O'REILLY
18 rue Séguier
75006 Paris
FRANCE
france@oreilly.com
<url:http://www.editions-oreilly.fr/>

O'REILLY®

Cambridge • Cologne • Farnham • Paris • Pékin • Sebastopol • Taipei • Tokyo

The original edition of this book (ISBN 2-84177-121-0) was published in France by O'REILLY & Associates under the title *Dveloppement d'applications avec Objective Caml.*

Historique :

- Version 19990324???????????

© O'REILLY & Associates, 2000

Cover concept by Ellie Volckhausen.

Édition : Xavier Cazin.

Les programmes figurant dans ce livre ont pour but d'illustrer les sujets traités. Il n'est donné aucune garantie quant à leur fonctionnement une fois compilés, assemblés ou interprétés dans le cadre d'une utilisation professionnelle ou commerciale.

© Éditions O'Reilly, Paris, 2000
ISBN

Toute représentation ou reproduction, intégrale ou partielle, faite sans le consentement de l'auteur, de ses ayants droit, ou ayants cause, est illicite (loi du 11 mars 1957, alinéa 1er de l'article 40). Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait une contrefaçon sanctionnée par les articles 425 et suivants du Code pénal. La loi du 11 mars 1957 autorise uniquement, aux termes des alinéas 2 et 3 de l'article 41, les copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective d'une part et, d'autre part, les analyses et les courtes citations dans un but d'exemple et d'illustration.

前書き

Objective Caml の教科書を書きたいという欲求は、著者らが Objective Caml 言語を通してプログラミングとは何かを教えていた間に得た教育体験から沸いてきたものです。Pierre et Marie Curie 大学には、様々な分野を専門にする学生や、社会人教育を受けている技術者たちがいましたが、彼らの積極性と批判力のおかげで、この本での言語の説明の仕方は大幅な改良をすることができました。特に、この本で出てくる例のいくつかは、彼らのプロジェクトから直接思いついたものです。

Caml 言語の開発は、15 年もの間ずっと続けられてきました。この開発は、初めは Formel プロジェクトとして、続いて Cristal プロジェクトとして、INRIA にて、Denis Diderot 大学と École Normale Supéieure の共同で行われてきました。これらのチームにいた研究者たちのたゆまぬ努力（開発もさながら、理論的な下支えの構築も）は、長年にわたって最高品質の言語を産み出してきました。彼らは、プログラミング言語の分野の止まらざる進歩についていくと同時に、新しいプログラミングパラダイムを理論的枠組みに取り入れていくことのできる人たちだったのです。彼らの言語は、広く普及するに値するでしょう。私たちは、この著作を通じて、それに貢献できればと思っています。

数々の同士たちの協力がなければ、この本は今あるような形にはならなかったでしょう。彼らは、初期の草稿を嫌がらず何度も読んでくれました。執筆の間、常に彼らのコメントのおかげで説明方法を改善することができたのです。私たちが特に感謝を捧げたい方々は、María-Virginia Aponte、Sylvain Baro、Christian Codognet、Hélène Cottier、Guy Cousineau、Pierre Crégut、Titou Durand、Christophe Gonzales、Michelle Morcrette、Christian Queinnec、Attila Raksany、Didier Rémy の各氏です。

この本の HTML 版は、`hevea` と `VideoC` というツールがなければ日の目を見なかったでしょう。それぞれの作者 Luc Maranget 氏と Christian Queinnec 氏は、私たちの質問やツールへの変更希望に対して常に短時間で返事をしてくれました。深く感謝します。

Contents

前書き v

Table of contents *vii*

はじめに *xxi*

1: How to obtain Objective Caml *1*

Description of the CD-ROM 1
Downloading 2
Installation 2
    Installation under Windows 2
    Installation under LINUX 4
    Installation under MacOS 4
    Installation from source under Unix 5
    Installation of the HTML documentation 5
Testing the installation 5

I Language Core **7**

2: 関数型プログラミング *11*

Objective Caml の核となる関数型の部分 12
    プリミティブな値、関数、および型 12
    条件制御構造 18

値の宣言 . . . 19
関数式、関数 . . . 21
多相性と型制約 . . . 28
例 . . . 31
型宣言とパターンマッチング . . . 33
パターンマッチング . . . 33
型宣言 . . . 41
レコード . . . 42
和型 . . . 44
再帰型 . . . 47
パラメータ化された型 . . . 47
宣言のスコープ . . . 48
関数型 . . . 49
例：木の表現 . . . 50
関数以外の再帰的値 . . . 52
型付け、定義域、例外 . . . 53
部分関数と例外 . . . 53
例外の定義 . . . 54
例外の発生 . . . 55
例外処理 . . . 56
多相性と関数の返り値 . . . 57
電卓 . . . 59
練習問題 . . . 62
二つのリストをマージする . . . 62
字句木 . . . 62
Graph traversal . . . 63
Summary . . . 64
To learn more . . . 64

3: 手続き型プログラミング *65*

変更可能データ構造 . . . 66
ベクトル . . . 66
文字列 . . . 70
レコードの変更可能フィールド . . . 71
参照 . . . 72
多相性と書き換え可能な値 . . . 72
入出力 . . . 74
チャネル . . . 75
読み書き . . . 75
例：数当てゲーム . . . 76
制御構造 . . . 77
連接 . . . 77
繰り返し . . . 79
例：スタックの実装 . . . 80
例：行列計算 . . . 82
引数の評価順序 . . . 84

メモリつき電卓 84
練習問題 87
二重リンクリスト 87
連立一次方程式を解く 87
まとめ 88
もっと知りたい方へ 88

4: Functional and Imperative Styles 89

Comparison between Functional and Imperative 90
The Functional Side 91
The Imperative Side 91
Recursive or Iterative 93
Which Style to Choose? 94
Sequence or Composition of Functions 95
Shared or Copy Values 97
How to Choose your Style 99
Mixing Styles 101
Closures and Side Effects 101
Physical Modifications and Exceptions 103
Modifiable Functional Data Structures 103
Lazy Modifiable Data Structures 105
Streams of Data 108
Construction 108
Destruction and Matching of Streams 110
Exercises 114
Binary Trees 114
Spelling Corrector 114
Set of Prime Numbers 115
Summary 115
To Learn More 115

5: The Graphics Interface 117

Using the `Graphics` Module 118
Basic notions 118
Graphical display 119
Reference point and graphical context 119
Colors 120
Drawing and filling 121
Text 123
Bitmaps 125
Example: drawing of boxes with relief patterns 126
Animation 130
Events 132
Types and functions for events 132
Program skeleton 133

Example: telecran . . . 134
A Graphical Calculator . . . 136
Exercises . . . 141
Polar coordinates . . . 141
Bitmap editor . . . 142
Earth worm . . . 143
Summary . . . 144
To learn more . . . 144

6: Applications *147*

Database queries . . . 148
Data format . . . 148
Reading a database from a file . . . 150
General principles for database processing . . . 151
Selection criteria . . . 153
Processing and computation . . . 156
An example . . . 157
Further work . . . 159
BASIC interpreter . . . 159
Abstract syntax . . . 160
Program pretty printing . . . 162
Lexing . . . 163
Parsing . . . 165
Evaluation . . . 170
Finishing touches . . . 174
Further work . . . 176
Minesweeper . . . 176
The abstract mine field . . . 177
Displaying the Minesweeper game . . . 182
Interaction with the player . . . 188
Exercises . . . 192

II Development Tools **193**

7: Compilation and Portability *197*

Steps of Compilation . . . 198
The Objective Caml Compilers . . . 198
Description of the Bytecode Compiler . . . 199
Compilation . . . 201
Command Names . . . 201
Compilation Unit . . . 201
Naming Rules for File Extensions . . . 202
The Bytecode Compiler . . . 202
Native Compiler . . . 204

Toplevel Loop . . . 205
Construction of a New Interactive System . . . 206
Standalone Executables . . . 207
Portability and Efficiency . . . 208
Standalone Files and Portability . . . 208
Efficiency of Execution . . . 208
Exercises . . . 209
Creation of a Toplevel and Standalone Executable . . . 209
Comparison of Performance . . . 209
Summary . . . 210
To Learn More . . . 210

8: Libraries *213*

Categorization and Use of the Libraries . . . 214
Preloaded Library . . . 215
Standard Library . . . 215
Utilities . . . 216
Linear Data Structures . . . 217
Input-output . . . 223
Persistence . . . 228
Interface with the System . . . 234
Other Libraries in the Distribution . . . 239
Exact Math . . . 240
Dynamic Loading of Code . . . 242
Exercises . . . 245
Resolution of Linear Systems . . . 245
Search for Prime Numbers . . . 245
Displaying *Bitmaps* . . . 246
Summary . . . 246
To Learn More . . . 246

9: ガーベージコレクション *249*

プログラムの使用するメモリ . . . 250
メモリの割り当てと解放 . . . 251
明示的な割り当て . . . 251
明示的な解放 . . . 252
暗黙の解放 . . . 253
ガーベージコレクション . . . 253
参照カウント . . . 254
スイープアルゴリズム . . . 255
マーク・アンド・スイープ . . . 256
ストップ・アンド・コピー . . . 258
他のガーベージコレクションアルゴリズム . . . 261
Objective Caml でのメモリ管理 . . . 263
`Gc` モジュール . . . 265

Weak モジュール 267
練習問題 270
ヒープの変化の追跡 270
メモリ使用量とプログラミングスタイル 271
まとめ 271
もっと知りたい人へ 272

10: プログラム解析ツール 273

依存性解析 274
デバッグツール 275
Trace（トレース） 275
デバッグ 280
実行の制御 281
プロファイリング 282
コンパイル方法 283
プログラムの実行 283
結果の出力 285
練習問題 287
関数適用のトレース 287
性能解析 287
まとめ 287
もっと知りたい人へ 288

11: 字句解析と構文解析のためのツール 289

語彙 290
Genlex モジュール 290
文字の流れの制御 291
正規表現 292
Str ライブラリ 294
ocamllex ツール 295
構文 297
文法 297
生成と認識 298
トップダウン解析 299
ボトムアップ解析 302
ocamlyacc ツール 305
文脈依存文法 308
Basic 再考 309
basic_parser.mly ファイル 310
basic_lexer.mll ファイル 312
コンパイルとリンク 313
練習問題 314
コメントの除去 314
評価器 314
まとめ 315

もっと知りたい人へ . . . 315

12: Interoperability with C 317

Communication between C and Objective Caml . . . 319
External declarations . . . 320
Declaration of the C functions . . . 320
Linking with C . . . 322
Mixing input-output in C and in Objective Caml . . . 325
Exploring Objective Caml values from C . . . 325
Classification of Objective Caml representations . . . 326
Accessing immediate values . . . 327
Representation of structured values . . . 328
Creating and modifying Objective Caml values from C . . . 336
Modifying Objective Caml values . . . 337
Allocating new blocks . . . 337
Storing C data in the Objective Caml heap . . . 338
Garbage collection and C parameters and local variables . . . 342
Calling an Objective Caml closure from C . . . 343
Exception handling in C and in Objective Caml . . . 345
Raising a predefined exception . . . 345
Raising a user-defined exception . . . 345
Catching an exception . . . 346
Main program in C . . . 348
Linking Objective Caml code with C . . . 348
Exercises . . . 348
Polymorphic Printing Function . . . 348
Matrix Product . . . 349
Counting Words: Main Program in C . . . 349
Summary . . . 350
To Learn More . . . 350

13: Applications 353

グラフィカルインターフェースの構築 . . . 353
グラフィックスコンテキスト、イベント、オプション . . . 354
コンポーネントとコンテナ . . . 358
イベントハンドリング . . . 362
コンポーネントの定義 . . . 367
拡張コンポーネント . . . 378
Awi ライブラリのセットアップ . . . 380
Example: A フラン-ユーロ変換器 . . . 381
もっと知りたい方は . . . 384
最小コスト経路を見つける . . . 384
グラフの表現 . . . 385
ダイクストラ法 . . . 389
キャッシュの紹介 . . . 393

グラフィカルインターフェース 395
スタンドアロンアプリケーションの作成 401
最後に 404

III Application Structure 405

14: モジュールを使ったプログラミング 409

コンパイル単位としてのモジュール 410
インターフェースと実装 410
インターフェースと実装を関連づける 412
分割コンパイル 413
モジュール言語 414
ふたつのスタックモジュール 415
モジュールと情報隠蔽 418
モジュール間の型共有 420
単純なモジュールを拡張する 422
パラメータつきモジュール 423
ファンクタとコード再利用 424
モジュールの局所定義 427
より大きな例: 銀行口座の管理 428
プログラムの構成 428
モジュールパラメータのシグネチャ 428
口座管理用パラメータつきモジュール 430
パラメータの実装 432
練習問題 436
連想リスト 436
パラメータつきベクトル 436
字句解析木 437
まとめ 437
もっと学びたい人のために 437

15: オブジェクト指向プログラミング 439

クラス、オブジェクト、メソッド 440
オブジェクト指向についての用語 440
クラス宣言 441
インスタンス生成 444
メッセージ送信 444
クラスの間の関係 445
集約 445
継承関係 447
その他のオブジェクト指向の機能 449
特別な参照 `self` と `super` 449
遅延束縛 450
オブジェクトの内部表現とメッセージの発送 451

初期化 . . . 452
プライベートメソッド . . . 453
型と総称性 . . . 454
抽象クラスと抽象メソッド . . . 455
クラス、型、オブジェクト . . . 457
多重継承 . . . 461
型パラメータを持つクラス . . . 464
部分型と包含的多相性 . . . 470
Example . . . 470
部分型関係は継承ではない . . . 471
包含的多相性 . . . 473
オブジェクトの等価性 . . . 474
関数型スタイル . . . 475
オブジェクト拡張機能の他の部分について . . . 478
インターフェース . . . 479
クラス内での局所宣言 . . . 480
練習問題 . . . 482
オブジェクトによるスタック . . . 482
遅延束縛 . . . 483
抽象クラスと式の評価器 . . . 484
ライフゲームとオブジェクト . . . 485
まとめ . . . 485
もっと知りたい人へ . . . 485

16: アプリケーションの構成モデルの比較 *487*

モジュールとオブジェクトの比較 . . . 488
モジュールをクラスに変換する . . . 491
モジュールにおける継承の実現 . . . 493
それぞれのモデルの限界 . . . 494
コンポーネントの拡張 . . . 496
関数型モデルの場合 . . . 497
オブジェクトモデルの場合 . . . 497
データとメソッドの拡張 . . . 499
混在した構成 . . . 501
練習問題 . . . 502
データ構造へのクラスとモジュール . . . 502
抽象型 . . . 502
まとめ . . . 503
さらに学びたい人のために . . . 503

17: アプリケーション *505*

二人ゲーム . . . 505
二人プレーヤー用ゲームの問題 . . . 506
ミニマックス $\alpha\beta$ . . . 506
ゲームプログラムの構造 . . . 514

Connect Four . . . 519
Stonehenge . . . 531
ファンシー・ロボット . . . 554
「抽象」ロボット . . . 555
純粋仮想世界 . . . 557
文字ロボット . . . 558
テキストの仮想世界 . . . 560
グラフィカルなロボット . . . 562
グラフィカルな世界 . . . 565
さらに学びたい人のために . . . 567

IV 並行分散プログラミング 569

18: 通信とプロセス *575*

`Unix` モジュール . . . 576
エラー処理 . . . 577
システムコールの互換性 . . . 577
ファイル記述子 . . . 577
ファイル操作 . . . 579
ファイル入出力 . . . 580
プロセス . . . 582
プログラムの実行 . . . 583
プロセス生成 . . . 584
複製によるプロセス生成 . . . 586
実行の順序と継続時間 . . . 588
プロセスの死、プロセスの葬儀 . . . 589
プロセス間通信 . . . 591
通信パイプ . . . 591
通信チャネル . . . 592
Unix のシグナル . . . 594
練習問題 . . . 598
語数計算: `wc` . . . 599
パイプを使ったスペルチェック . . . 599
計算状態の対話的な取得 . . . 599
まとめ . . . 600
もっと知りたい人へ . . . 600

19: 並行プログラミング *601*

並行プロセス . . . 602
スレッドを使ったプログラムのコンパイル . . . 603
`Thread` モジュール . . . 604
プロセスの同期 . . . 606
クリティカルセクションと相互排除 . . . 606
`Mutex` モジュール . . . 606

待機と同期 . . . 611
Condition モジュール . . . 611
同期通信 . . . 615
通信イベントを使った同期 . . . 615
転送される値 . . . 615
Event モジュール . . . 616
例題: 郵便局 . . . 618
資源と主体の実装 . . . 618
客と職員 . . . 620
システム全体 . . . 622
練習問題 . . . 622
食事にありつく哲学者 . . . 622
郵便局の拡張 . . . 622
生産者消費者のオブジェクトバージョン . . . 623
まとめ . . . 624
もっと知りたい人へ . . . 624

20: 分散プログラミング *625*

インターネット（The Internet） . . . 626
Unix モジュールと IP アドレッシング . . . 628
ソケット . . . 629
記述と生成 . . . 630
アドレスと接続 . . . 631
クライアント - サーバ . . . 632
クライアント - サーバの動作モデル . . . 633
クライアント - サーバプログラミング . . . 633
サーバのコード . . . 634
telnet を利用したテスト . . . 637
クライアントコード . . . 637
ライト級プロセスを用いたクライアント - サーバプログラミング . . . 641
多段クライアント - サーバプログラミング . . . 644
クライアント - サーバプログラムに関するコメント . . . 644
通信プロトコル . . . 645
テクストプロトコル . . . 646
確認応答と時間制限のあるプロトコル . . . 648
内部表現のままの値の転送 . . . 649
異言語間相互運用 . . . 649
演習問題 . . . 650
サービス： 時計 . . . 650
ネットワーク コーヒー自動販売機 . . . 650
まとめ . . . 651
さらに進んだ話題 . . . 651

21: アプリケーション *653*

Client-server Toolbox . . . 653

Protocols . . . 654
Communication . . . 654
Server . . . 655
Client . . . 657
To Learn More . . . 658
The Robots of Dawn . . . 658
World-Server . . . 659
Observer-client . . . 663
Robot-Client . . . 664
To Learn More . . . 666
HTTP Servlets . . . 666
HTTP and CGI Formats . . . 667
HTML Servlet Interface . . . 672
Dynamic Pages for Managing the Association Database . . . 675
Analysis of Requests and Response . . . 677
Main Entry Point and Application . . . 677

*22: Objective Caml*でのアプリケーション開発 *681*

評価項目 . . . 682
言語 . . . 682
ライブラリとツール . . . 683
文書 . . . 684
他の開発ツール . . . 684
編集ツール . . . 684
文法拡張 . . . 685
他言語インターフェース . . . 685
グラフィックインターフェース . . . 685
並列分散プログラミング . . . 686
Objective Caml で開発されたアプリケーション . . . 687
類似の関数型言語 . . . 688
ML 系列 . . . 688
Scheme . . . 689
遅延評価の機能を持つ言語 . . . 689
通信記述言語 . . . 692
オブジェクト指向言語 — Java との比較 . . . 692
主な特徴 . . . 693
Objective Caml との違い . . . 693
Objective Caml 開発の将来 . . . 694

おわりに *697*

V Appendices 699

A: Cyclic Types 701

Cyclic types 701
Option `-rectypes` 703

B: Objective Caml 3.04 *707*

Language Extensions 707
Labels 708
Optional arguments 710
Labels and objects 712
Polymorphic variants 713
`LablTk` Library 716
OCamlBrowser 716

参考文献 *719*

概念索引 *723*

言語要素索引 *729*

和訳担当　前書き・はじめに: 細谷, 1 章: (未定), 2 章: 住井, 3 章: 細谷, 4 章: 脇田, 5 章: 古瀬, 6 章: (未定), 7 章: 玉田, 8 章: 坂内, 9–11 章: 関口, 12 章: Garrigue, 13 章: 原, 14 章: 五十嵐, 15 章: 関口, 16–17 章: 富沢, 18–19 章: 関口, 20 章: 橋本, 21–22 章: 関口, 参考文献: 富沢

はじめに

Objective Caml はプログラミング言語の一つです。どうしてまた新しい言語が要るのか？と聞く人もいるかも知れません。実際、今すでに数多くの言語が存在していますし、新しいものもどんどん現れています。まずこれらの言語の違いはさておき、それぞれの理念と起源は、共通した動機からきています。それは「抽象化したい」ということです。

計算機の抽象化 まずプログラミング言語では、計算機の「機械」的な部分を無視できるようになります。マイクロプロセッサやオペレーティングシステムさえも、プログラムがその上を走っているのにもかかわらず、忘れ去ることができます。

操作的モデルの抽象化 大概の言語は、関数という概念を何らかの形で持っていますが、それは数学から借りてきた概念であって、電子工学からではありません。一般に言語は、純粋に計算機的な視点ではなく、数学的モデルで考えさせてくれます。こうして言語は**表現力**を得るのです。

エラーの抽象化 これは、実行の安全性を保証しようという話です。プログラムは、エラーが発生したときに突如停止したり不整合を起こしたりするべきではありません。安全性の保証を得るひとつの方法は、プログラムに強い静的型付けを施し、適切な場所で例外機構を使用するということです。

部品の抽象化（I） プログラミング言語では、一つのアプリケーションを様々なおおよそ独立自立したソフトウェア部品に分解できるようになります。モジュールを使うと、複雑な一つのアプリケーション全体を高度な方法で構造化することができます。

部品の抽象化（II） プログラムの単位というものがあるおかげで、今度はそれを、開発時に想定されていた場所以外で再利用できる可能性が出てきました。**オブジェクト指向**言語は、速攻プロトタイピングを可能にする再利用性への新しいアプローチです。

Objective Caml は、最近現れた言語であり、プログラミング言語の歴史の中では Lisp の遠い子孫と位置づけられます。また、従兄にあたる言語からの教訓を生かしたり、その

ほかの言語からも主要概念を取り込んだりしてきました。この言語は、INRIA[1]で開発され、長期に渡る ML 言語族の概念構築の経験の上に成り立っています。Objective Caml は、記号および数値アルゴリズムを記述するための汎用言語です。オブジェクト指向でもあり、またパラメータつきモジュールシステムも備えています。並列・分散アプリケーションの開発のための機能もサポートしています。静的型チェック、例外機構、ガベージコレクションのおかげで優れた実行時安全性を提供しています。高速で、しかもポータブルです。開発環境も充実しています。

しかしながら、Objective Caml は「一般人」向けに宣伝される機会がありませんでした。これは、この本の著者たちに課せられた任務です。この著作には 3 つの目的があります。

1. Objective Caml 言語、およびそのライブラリ、開発環境を詳しく解説すること。
2. Objective Caml で使えるプログラミングスタイルの裏には、どういう概念が潜んでいるのかを明らかにすること。
3. 多数の例を通して、Objective Caml がいかに多種多様なアプリケーションの開発言語として役に立つかということを例示すること。

著者の最終目標は、どのようにプログラミングスタイルを選び、どのようにプログラムを構築するかということへの知見を提供することです。プログラムを組むときには、与えられた問題と整合性を持つように組むべきで、そうすることによってそのプログラムが維持可能になり、その部品が再利用可能になるのです。

言語の特徴

Objective Caml は関数型言語です。　この言語では、関数を値として扱うことができます。これらはさらに他の関数への引数として、もしくは関数の結果として返すことができます

Objective Caml は静的型付きです。　実引数と仮引数の型の整合性は、プログラムのコンパイル時に検査されます。その後、プログラムの実行時にはそういう検査は不要になるので、効率が向上します。さらに、静的型判定によって、打ち間違いや思慮不足から生じるエラーのほとんどを除去でき、実行時の安全性に貢献します。

Objective Caml はパラメータ的多相性を備えています。　もしある関数が、引数のデータ構造全体をなめることがない場合、その引数の型は完全に決定されていなくてもよくなります。このとき、この引数は多相的であるといいます。この機能によって、いろいろなデータ構造のために利用できる汎用的なコードを開発することができるようになります。このコードは、受け取るデータ構造の表現を完全に知っている必要がありません。型付けアルゴリズムが、表現を識別する任務を担います。

1. Institut National de Recherche en Informatique et Automatique (National Institute for Research in Automation and Information Technology).

Objective Caml は型推論を備えています。 プログラマは、プログラムに型情報を全く与えなくて構いません。この言語は、現れる式や宣言の型として最も一般的なものを、コードから自動的に導き出してくれます。この**推論**は、検査と同時に、プログラムコンパイル時に行われます。

Objective Caml は例外機構を備えています。 この機能のおかげで、プログラムは通常実行をある場所で中断して、他の場所で再開するということが可能になります。この機構は、例外的な状況の制御に使えることはもちろん、一つのプログラミングスタイルとして利用することもできます。

Objective Caml は手続き型の機能も備えています。 入出力、値の物理的更新、繰り返し制御構造が、関数型プログラミング機能の助けを借りずに使えます。これら２つのスタイルを混ぜてもよく、それによって非常に柔軟な開発ができ、また新しい種類のデータ構造を定義することもできます。

Objective Caml はスレッドを走らせることができます。 スレッドの生成、同期、共有メモリの管理、スレッド間通信が組み込まれています。

Objective Caml は Internet で通信できます。 異なるマシン間の通信チャネルを開くために必要な関数が組み込まれており、クライアント＝サーバアプリケーションの開発が可能です。

Objective Caml にはライブラリがたくさん揃っています。 古典的なデータ構造、入出力、システム資源とのインタフェース、字句・構文解析、大きな数値を使った計算、永続的な値など。

Objective Caml は開発環境を提供します。 応答型トップレベル、実行トレース、依存関係の計算、プロファイルなど。

Objective Caml は C 言語とインタフェースをとることができます。 これは、C の関数を Objective Caml プログラムから呼んだり、またその逆によって行います。これによって数多くの C のライブラリへアクセスすることが可能になります。

Objective Caml には３つの実行モードがあります。 トップレベルによるインタラクティブモード、仮想機械によって解釈実行されるバイトコードへのコンパイル、ネイティブ機械コードへのコンパイルです。これによってプログラマは、開発の柔軟性、異なるアーキテクチャ間でのオブジェクトコードの可搬性、特定のアーキテクチャでの実行効率、という３つの選択肢を得ることになります。

プログラムの構成

重要なアプリケーションの開発では、プログラマもしくは開発チームは、プログラムをどう整理するか、どう構成するかという問題を考えることが要求されます。Objective Caml では、利点と機能の違う 2 つのモデルが用意されております。

パラメータつきモジュールモデル　データと手続きは、ひとつの入れ物の中にまとめることができます。その入れ物は、コードそのものと、インタフェースという 2 つの側面を持っています。モジュール間の通信はインタフェースを通して行われます。型の定義は、隠れてもよい、つまりモジュールインタフェースに現れなくても構いません。このような抽象的なデータ型を使うと、モジュールの内部実装を変更しながら、使用側のモジュールには影響を与えないということが簡単にできます。さらに、モジュールは別のモジュールをパラメータとして受け取るようにできるので、再利用性が向上できます。

オブジェクトモデル　手続きとデータは、クラスという箱の中にまとめることができます。オブジェクトとは、クラスの一つのインスタンスです。オブジェクト間の通信は、「メッセージ渡し」を通して実現されます。受信オブジェクトは、どの手続きがメッセージに対応するかということを実行時に決定します（遅延束縛）。このように、オブジェクト指向はデータ駆動的です。プログラムの構成はクラス間の関係から成ります。とくに、継承を使うと、あるクラスを定義するのに、他のクラスを拡張することによって行うことができます。このオブジェクトモデルでは、具体クラスも、抽象クラスも、パラメータつきクラスも使えます。さらに、クラス間のサブタイプ関係を定義することによって、包含的多相性をも導入することができます。

モデルとしてこのように選択肢が 2 つあるおかげで、アプリケーションの論理的整理を非常に柔軟に行うことができ、またその維持と進化も容易にします。これら 2 つのモデルの間には双対性があります。モジュールの型にはデータフィールドを追加することはできません（データの拡張不能性）が、データに作用する手続きを新しく追加することはできます（手続きの拡張可能性）。オブジェクトモデルの場合は、クラスのサブクラスを追加することはできます（データの拡張可能性）が、新しい手続きを先祖のクラスに見えるように追加することはできません（手続きの拡張不能性）。それでも、これら 2 つを組合わせることによって、データと手続きの拡張性への新たな可能性が生まれます。

実行時安全性と効率

Objective Caml は、優れた安全性を、効率の犠牲なしに提供してくれます。理論的な言葉では、静的型付けとは実行時の型エラーが起き得ないことの保証です。コンパイラにとっては、静的情報は動的な型チェックで効率が損なわれないようにするために有用です。この利点はオブジェクト指向の機能についても言えることです。さらに、組み込みのガベージコレクションのおかげで処理系の安全性がさらに向上します。Objective Caml は抜群に高速です。例外機構は、プログラムがゼロによる除算や配列の範囲外アクセスをしたときに、不整合な状態に陥らないようにするためのものです。

この本の構成

本著書は、４つのメインの部から成り、その前後には章が２つ、それから付録が２つ、参考文献、言語構成要素とプログラミング概念の索引がついています。

１章： この章では、Objective Caml 言語バージョン 2.04 を現行の主なシステム（Windows、Unix、MacOS）の上にインストールする方法を説明します。

第１部：言語コア 第１部では、Objective Caml 言語の基本要素をすべて網羅して解説します。まず 2 章で、本言語の関数型のコアからスタートします。3 章は、前章の続きとして手続き型の機能を説明します。4 章では、純粋な関数型と手続き型を比較し、両者を使ったスタイルについてもお話しします。5 章では、グラフィックスライブラリを紹介します。6 章では、３つのアプリケーションを書いてみることにします。簡単なデータベース管理、ミニ Basic インタープリタ、そして有名なマインスイーパという一人用ゲームです。

第２部：開発ツール 第２部では、アプリケーション開発のためのいろいろなツールについて解説します。7 章では異なるコンパイルモードを比較します。それは、インタラクティブトップレベル、コマンドラインからのバイトコードコンパイル、ネイティブコードへのコンパイルです。8 章では、言語のディストリビューションに提供されている標準ライブラリを紹介します。 9 章は、ガベージコレクションとは何か、特に Objective Caml で使われているものについて詳しく述べます。10 章では、デバッグやプロファイルを行うためのツールを紹介します。11 章は、字句・構文解析ツールについてやります。12 章では、Objective Caml プログラムの中で C 言語とのインタフェースを取る方法を説明します。13 章では、あるライブラリとアプリケーションを構築してみます。そのライブラリとは、GUI 構築をするためのツールで、アプリケーションの方は、グラフの最小費用経路の探索をするもので、前述のライブラリを使った GUI を持っています。

第３部：アプリケーションの整理 第３部では、２通りあるプログラム整理方法を解説します。つまりモジュールによるものと、オブジェクトによるものです。14 章では、単純なモジュール、およびパラメータつきのモジュールを説明します。15 章では、Objective Caml のオブジェクト指向機能についてやります。16 章では、これら２つの整理方を比較し、２つを混在させることがプログラムの拡張性を向上させるのに役に立つことを示します。17 章では、歯応えのあるアプリケーションを２つ書いてみることにします。１つめは二人用ゲームで、異なる２つのゲームのためにパラメータつきモジュールをいくつも駆使します。もう１つは、ロボット世界のシミュレーションで、オブジェクト間通信を実演します。

第４部：並行・分散プログラミング 第４部では、プロセス（軽量プロセス、および通常のプロセス）の間の通信と、Internet 上の通信について詳しく説明しながら、並行・分散プログラミングというものを紹介します。18 章では、言語とシステムライブラリがどう直結するのか、特にプロセスと通信の概念を説明します。19 章では、Objective Caml のスレッドを紹介しながら、並行プログラミングにおいて決定性がなくなることについて触れます。20 章では、分散メモリモデルにおけるソケットを通じたプロセス間通信について議論します。21 章では、まずクライアント＝サーバアプリケーションを書くためのツールボックスを紹介します。そしてこ

れを使って、前部のロボットの例をクライアント＝サーバモデルに拡張します。最後に、すでに出てきたプログラムを HTTP サーバの形に変身させます。

22 章： この最後の章では、Objective Caml によるアプリケーション開発のよさを値踏みし、ML 言語族で書かれたアプリケーションの中で最も有名なものを紹介します。

付録 １つ目の付録では、オブジェクトの型付けで使われている循環型について説明します。２つ目の付録では、新しいバージョン 3.00 で加えられた言語の変更について述べます。これらの変更は、Objective Caml のこの後のバージョン（3.xx）でも採り入れられています。

各章は、導入、章の構成、いくつかの節からなる本文、練習問題、まとめ、に続いて「さらに知りたい方へ」と題した節が最後に来ます。その最後の節では、その章のテーマに関連した参考文献を挙げます。

1

How to obtain Objective Caml

The various programs used in this work are "free" software [1]. They can be found either on the CD-ROM accompanying this work, or by downloading them from the Internet. This is the case for Objective Caml, developed at INRIA.

Description of the CD-ROM

The CD-ROM is provided as a hierarchy of files. At the root can be found the file `index.html` which presents the CD-ROM, as well as the five subdirectories below:

- book: root of the HTML version of the book along with the solutions to the exercises;
- apps: applications described in the book;
- exercises: independent solutions to the proposed exercises;
- distrib: set of distributions provided by INRIA, as described in the next section;
- tools: set of tools for development in Objective Caml;
- docs: online documentation of the distribution and the tools.

To read the CD-ROM, start by opening the file `index.html` in the root using your browser of choice. To access directly the hypertext version of the book, open the file `book/index.html`. This file hierarchy, updated in accordance with readers' remarks, can be found posted on the editor's site:

リンク: http://www.oreilly.fr

1. "Free software" is not to be confused with "freeware". "Freeware" is software which costs nothing, whereas "free software" is software whose source is also freely available. In the present case, all the programs used cost nothing and their source is available.

Downloading

Objective Caml can be downloaded via web browser at the following address:

リンク: http://caml.inria.fr/ocaml/distrib.html

There one can find binary distributions for Linux (Intel and PPC), for Windows (NT, 95, 98) and for MacOS (7, 8), as well as documentation, in English, in different formats (PDF, PostScript and HTML). The source code for the three systems is available for download as well. Once the desired distribution is copied to one's machine, it's time to install it. This procedure varies according to the operating system used.

Installation

Installing Objective Caml requires about 10MB of free space on one's hard disk drive. The software can easily be uninstalled without corrupting the system.

Installation under Windows

The file containing the binary distribution is called: `ocaml-2.04-win.zip`, indicating the version number (here `2.04`) and the operating system.

警告 Objective Caml only works under recent versions of Windows : Windows 95, 98 and NT. Don't try to install it under Windows 3.x or OS2/Warp.

1. The file is in compressed (`.zip`) format; the first thing to do is decompress it. Use your favorite decompression software for this. You obtain in this way a file hierarchy whose root is named `ocaml`. You can place this directory at any location on your hard disk. It is denoted by `<caml-dir>` in what follows.
2. This directory includes:
   - two subdirectories: `bin` for binaries and `lib` for libraries;
   - two "text" files: `License.txt` and `Changes.txt` containing the license to use the software and the changes relative to previous versions;
   - an application: `OCamlWin` corresponding to the main application;
   - a configuration file: `Ocamlwin.ini` which will need to be modified (see the following point);
   - two files of version notes: the first, `Readme.gen`, for this version and the second, `Readme.win`, for the version under Windows.
3. If you have chosen a directory other than `c:\ocaml` as the root of your file hierarchy, then it is necessary to indicate this in the configuration file. Edit it with `Wordpad` and change the line defining `CmdLine` which is of the form:
   `CmdLine=ocamlrun c:\ocaml\bin\ocaml.exe -I c:\ocaml\lib`
   to

CmdLine=ocamlrun <caml-dir>\bin\ocaml.exe -I <caml-dir>\lib
You have to replace the names of the search paths for binaries and libraries with the name of the Objective Caml root directory. If we have chosen `C:\Lang\ocaml` as the root directory (`<caml-dir>`), the modification becomes:
`CmdLine=ocamlrun C:\Lang\ocaml\bin\ocaml.exe -I C:\Lang\ocaml\lib`

4. Copy the file OCamlWin.ini to the main system directory, that is, `C:\windows` or `C:\win95` or `C:\winnt` according to the installation of your system.

Now it's time to test the `OCamlWin` application by double-clicking on it. You'll get the window in figure 1.1.

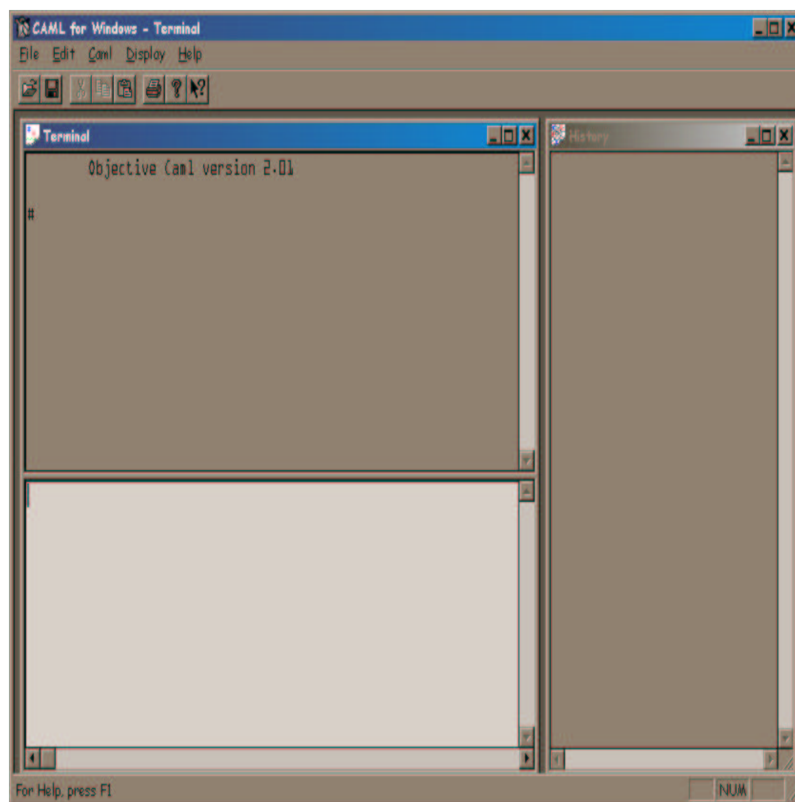


図 1.1: Objective Caml window under Windows.

The configuration of command-line executables, launched from a DOS window, is done by modifying the `PATH variable and the Objective Caml` library search path variable (`CAMLLIB`), as follows:

```
PATH=%PATH%;<caml-dir>\bin
set CAMLLIB=<caml-dir>\lib
```

where `<caml-dir>` is replaced by the path where Objective Caml is installed.

These two commands can be included in the `autoexec.bat` file which every good DOS has. To test the command-line executables, type the command `ocaml` in a DOS window. This executes the file:

```
<caml-dir>/bin/ocaml.exe
```

corresponding to the Objective Caml. text mode toplevel. To exit from this command, type `#quit;;`.

To install Objective Caml from source under Windows is not so easy, because it requires the use of commercial software, in particular the Microsoft C compiler. Refer to the file `Readme.win` of the binary distribution to get the details.

Installation under LINUX

The LINUX installation also has an easy-to-install binary distribution in the form of an `rpm`. package. Installation from source is described in section 1. The file to download is: `ocaml-2.04-2.i386.rpm` which will be used as follows with root privileges:

```
rpm -i ocaml-2.04-2.i386.rpm
```

which installs the executables in the `/usr/bin` directory and the libraries in the `/usr/lib/ocaml` directory.

To test the installation, type: `ocamlc -v` which prints the version of Objective Caml installed on the machine.

```
ocamlc -v
The Objective Caml compiler, version 2.04
Standard library directory: /usr/lib/ocaml
```

You can also execute the command `ocaml` which prints the header of the interactive toplevel.

```
        Objective Caml version 2.04

#
```

The `#` character is the prompt in the interactive toplevel. This interactive toplevel can be exited by the `#quit;;` directive, or by typing `CTRL-D`. The two semi-colons indicate the end of an Objective Caml phrase.

Installation under MacOS

The MacOS distribution is also in the form of a self-extracting binary. The file to download is: `ocaml-2.04-mac.sea.bin` which is compressed. Use your favorite software

to decompress it. Then all you have to do to install it is launch the self-extracting archive and follow the instructions printed in the dialog box to choose the location of the distribution. For the MacOS X server distribution, follow the installation from source under Unix.

Installation from source under Unix

Objective Caml can be installed on systems in the Unix family from the source distribution. Indeed it will be necessary to compile the Objective Caml system. To do this, one must either have a C compiler on one's Unix, machine, which is generally the case, or download one such as `gcc` which works on most Unix. systems. The Objective Caml distribution file containing the source is: `ocaml-2.04.tar.gz`. The file `INSTALL` describes, in a very clear way, the various stages of configuring, making, and then installing the binaries.

Installation of the HTML documentation

Objective Caml's English documentation is present also in the form of a hierarchy of HTML files which can be found in the `docs` directory of the CD-ROM.

This documentation is a reference manual. It is not easy reading for the beginner. Nevertheless it is quite useful as a description of the language, its tools, and its libraries. It will soon become indispensable for anyone who hopes to write a program of more than ten lines.

Testing the installation

Once installation of the Objective Caml development environment is done, it is necessary to test it, mainly to verify the search paths for executables and libraries. The simplest way is to launch the interactive toplevel of the system and write the first little program that follows:

```
String.concat "/" ["a"; "path"; "here"] ;;
```

This expression concatenates several character strings, inserting the "/" character between each word. The notation `String.concat` indicates use of the function `concat` from the `String`. If the library search path is not correct, the system will print an error. It will be noted that the system indicates that the computation returns a character string and prints the result.

The documentation of this function `String.concat` can be found in the online reference manual by following the links "The standard library" then "Module String: string operations".

To exit the interactive toplevel, the user must type the directive "`#quit ;;`".

Part I

Language Core

The first part of this book is a complete introduction to the core of the Objective Caml language, in particular the expression evaluation mechanism, static typing and the data memory model.

An expression is the description of a computation. Evaluation of an expression returns a value at the end of the computation. The execution of an Objective Caml program corresponds to the computation of an expression. Functions, program execution control structures, even conditions or loops, are themselves also expressions.

Static typing guarantees that the computation of an expression cannot cause a run-time type error. In fact, application of a function to some arguments (or actual parameters) isn't accepted unless they all have types compatible with the formal parameters indicated in the definition of the function. Furthermore, the Objective Caml language has type infererence: the compiler automatically determines the most general type of an expression.

Finally a minimal knowledge of the representation of data is indispensable to the programmer in order to master the effects of physical modifications to the data.

Outline

Chapter 2 contains a complete presentation of the purely functional part of the language and the constraints due to static typing. The notion of expression evaluation is illustrated there at length. The following control structures are detailed: conditional, function application and pattern matching. The differences between the type and the domain of a function are discussed in order to introduce the exception mechanism. This feature of the language goes beyond the functional context and allows management of computational breakdowns.

Chapter 3 exhibits the imperative style. The constructions there are closer to classic languages. Associative control structures such as sequence and iteration are presented there, as well as mutable data structures. The interaction between physical modifications and sharing of data is then detailed. Type inference is described there in the context of these new constructions.

Chapter 4 compares the two preceding styles and especially presents different mixed styles. This mixture supports in particular the construction of lazy data structures, including mutable ones.

Chapter 5 demonstrates the use of the `Graphics` library included in the language distribution. The basic notions of graphics programming are exhibited there and immediately put into practice. There's even something about GUI construction thanks to the minimal event control provided by this library.

These first four chapters are illustrated by a complete example, the implementation of a calculator, which evolves from chapter to chapter.

Chapter 6 presents three complete applications: a little database, a mini-BASIC interpreter and the game Minesweeper. The first two examples are constructed mainly in a functional style, while the third is done in an imperative style.

The rudiments of syntax

Before beginning we indicate the first elements of the syntax of the language. A program is a sequence of phrases in the language. A phrase is a complete, directly executable syntactic element (an expression, a declaration). A phrase is terminated with a double semi-colon (;;). There are three different types of declarations which are each marked with a different keyword:

value declaration	:	**let**
exception declaration	:	**exception**
type declaration	:	**type**

All the examples given in this part are to be input into the interactive toplevel of the language.

Here's a first (little) Objective Caml program, to be entered into the toplevel, whose prompt is the pound character (#), in which a function *fact* computing the factorial of a natural number, and its application to a natural number 8, are defined.

```
# let rec fact n = if n < 2 then 1 else n * fact(n-1) ;;
val fact : int -> int = <fun>
# fact 8 ;;
- : int = 40320
```

This program consists of two *phrases*. The first is the declaration of a function value and the second is an expression. One sees that the toplevel prints out three pieces of information which are: the name being declared, or a dash (-) in the case of an expression; the inferred type; and the return value. In the case of a function value, the system prints <fun>.

The following example demonstrates the manipulation of functions as values in the language. There we first of all define the function *succ* which calculates the successor of an integer, then the function *compose* which composes two functions. The latter will be applied to *fact* and *succ*.

```
# let succ x = x+1 ;;
val succ : int -> int = <fun>
# let compose f g x = f(g x)  ;;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# compose fact succ 8 ;;
- : int = 362880
```

This last call carries out the computation `fact(succ 8)` and returns the expected result. Let us note that the functions *fact* and *succ* are passed as parameters to compose in the same way as the natural number 8.

2

関数型プログラミング

最初の関数型言語 Lisp は 1950 年代末に登場しました。これは、最初の代表的な手続き型言語である Fortran と同時期です。この二つの言語は今でも存在しますが、どちらも大幅に進化しており、Fortran の場合は数値計算に、Lisp の場合は記号処理に広く使われています。関数型プログラミングが興味を惹くのは、プログラムを書いたり、プログラムが操作する値を指定するのが非常に簡単であるという点です。プログラムは関数であり、引数に適用され、計算した値を（その計算が停止すれば）プログラムの出力として返します。これにより、プログラムを組み合わせることが簡単になります。関数の合成により、あるプログラムの出力が別のプログラムの入力になるからです。

関数型プログラミングは、三つの構成要素を持つ単純な計算モデルに基づいています。その構成要素とは、変数、関数定義、そして関数を引数に適用することです。このモデルは λ 計算と呼ばれており、アロンゾ・チャーチによって 1932 年、すなわち最初のコンピュータ以前に提案されました。λ 計算は**計算可能性**の概念に一般的・理論的なモデルを与えるために考えられました。λ 計算では、すべての関数を値として扱うことができ、他の関数の引数として使ったり、他の関数を呼び出した結果として返すことができます。λ 計算の理論では、およそ計算可能（すなわちプログラム可能）なものは、この形式論の中で表すことができるとされています。ただし、λ 計算の構文は実用的なプログラミング言語として用いるには小さすぎるので、より使いやすくするために、プリミティブな値（整数や文字列など）、それらに対する演算、制御構造、関数や値に名前をつけることを可能にする宣言、また特に再帰関数等々が追加されてきました。

関数型言語の分類はいくつかありますが、ここでは我々にとってもっとも顕著と思われる二つの特徴に基づいて区別することにします。

- 副作用がない（純粋）か、副作用がある（純粋でない）か。純粋な関数型言語とは、状態の変化がない言語のことです。そのような言語では、すべてが単なる**計算**であり、それが実行される方法は重要ではありません。Lisp や ML のような純粋でない関数型言語は、状態の変化といった手続き型言語の特徴を取り入れ、Fortran のような言語に近いスタイルでアルゴリズムを書くことを許しています。そのような言語では、式を評価する順序が重要です。

- 動的に型付けされるか、静的に型付けされるか。型付けは、関数に渡される引数が実際にその関数の仮引数の型を持っているかどうか、検証することを可能にします。この検証はプログラムの実行中に行うことができます。その場合、この検証を**動的な型付け**と呼びます。もし型エラーが起こったら、プログラムは整合性のとれた状態で停止します。これは Lisp に当てはまります。また、この検証は実行前、つまりコンパイル時に行うこともできます。そのような先に行われる検証のことを**静的な型付け**と呼びます。静的な型付けは一度しか行われないので、プログラムの実行を遅くすることがありません。これは ML や、その方言である Objective Caml に当てはまります。正しく型付けされたプログラム、すなわち型検証器によって受理されたプログラムだけがコンパイルでき、実行されます。

本章の概要

本章では、言語 Objective Caml の関数型部分の基本的要素、すなわち構文的要素、型の文法および例外機構について説明します。その後で、例として初めての完全なプログラムを開発します。

第一節では、プリミティブな値やそれらを操作する関数をはじめ、言語の核となる部分を説明してから、構造を持つ値や関数値に進みます。また、基本的な制御構造や、ローカルおよびグローバルな値宣言を導入します。第二節では、構造を持った値を構成するための型定義と、そのような構造値にアクセスするパターンマッチングについて扱います。第三節では、関数の定義域と、推論される型とを比較することにより、例外機構について説明します。第四節では、単純なアプリケーションである電卓を通じて、これらの概念をまとめて示します。

*Objective Caml*の核となる関数型の部分

あらゆる関数型言語と同様に、Objective Caml は式を中心とする言語であり、主に関数を作ったり適用することによりプログラミングが行われます。一つの式を評価した結果は言語における値であり、プログラムの実行は、そのプログラムを構成するすべての式を評価することにあたります。

プリミティブな値、関数、および型

Objective Caml では、整数、浮動小数、文字、文字列、および論理値があらかじめ定義されています。

数値

Objective Caml には二種類の数があります。型 *int* の整数[1]と型 *float* の浮動小数です。Objective Caml における倍精度浮動小数の表現は、IEEE 754 規格[2]に従っています。整

1. 32 ビットのマシンでは $[-2^{30}, 2^{30}-1]$ の区間、64 ビットのマシンでは $[-2^{62}, 2^{62}-1]$ の区間
2. 浮動小数 $m \times 10^n$ を、53 ビットの仮数部 m と、区間 $[-1022, 1023]$ の指数部 n により表す

数と浮動小数に対する演算を図 2.1 に示します。整数演算の結果が型 *int* の定義されている区間外となるときは、エラーが起こるのではなく、そのシステムにおける整数の区間内の結果となることに注意してください。いいかえれば、すべての整数演算は、その区間の境界を法とする**剰余**の演算となります。

整数	浮動小数
`+` 足し算 `-` 引き算および符号反転 `*` かけ算 `/` 整数の割り算 `mod` 整数の割り算の余り	`+.` 足し算 `-.` 引き算および符号反転 `*.` かけ算 `/.` 割り算 `**` べき乗
# 1 ;; - : int = 1 # 1 + 2 ;; - : int = 3 # 9 / 2 ;; - : int = 4 # 11 **mod** 3 ;; - : int = 2 (* 整数の表現の限界 *) # 2147483650 ;; - : int = 2	# 2.0 ;; - : float = 2 # 1.1 +. 2.2 ;; - : float = 3.3 # 9.1 /. 2.2 ;; - : float = 4.13636363636 # 1. /. 0. ;; - : float = inf (* limits of the representation *) (* of floating-point numbers *) (* 浮動小数の表現の限界 *) # 222222222222.11111 ;; - : float = 222222222222

図 2.1: 数値に対する演算

整数と浮動小数の違い *float* と *int* のように異なる型を持つ値を直接比較することはできません。しかし、一方を他方に変換する関数（float_of_int と int_of_float）があります。

```
# 2 = 2.0 ;;
Characters 5-8:
  2 = 2.0 ;;
      ^^^
This expression has type float but is here used with type int
# 3.0 = float_of_int 3 ;;
- : bool = true
```

同様に、浮動小数に対する演算は整数に対する演算と異なります。

```
# 3 + 2 ;;
- : int = 5
```

```
# 3.0 +. 2.0 ;;
- : float = 5
# 3.0 + 2.0 ;;
Characters 0-3:
  3.0 + 2.0 ;;
  ^^^
This expression has type float but is here used with type int
# sin 3.14159 ;;
- : float = 2.65358979335e-06
```

0 による除算のような、結果が定義されていない計算は例外を発生し（53 ページを参照）、計算が中断します[3]。浮動小数には、無限の値（Inf と表示）や結果が定義されていない計算（NaN[4] と表示）の表現があります。浮動小数に対する主な関数を図 2.2 に示します。

浮動小数関数		三角関数	
ceil	切り上げ	cos	余弦
floor	切り下げ	sin	正弦
sqrt	平方根	tan	正接
exp	指数関数	acos	逆余弦
log	自然対数	asin	逆正弦
log10	底が 10 の log	atan	逆正接

```
# ceil 3.4 ;;
- : float = 4
# floor 3.4 ;;
- : float = 3
# ceil (-.3.4) ;;
- : float = -3
# floor (-.3.4) ;;
- : float = -4
```

```
# sin 1.57078 ;;
- : float = 0.999999999867
# sin (asin 0.707) ;;
- : float = 0.707
# acos 0.0 ;;
- : float = 1.57079632679
# asin 3.14 ;;
- : float = nan
```

図 2.2: 浮動小数関数

文字と文字列

文字は型 *char* を持ち、0 以上 255 以下の整数に対応します。初めの 128 文字は ASCII エンコーディングに従います。関数 char_of_int および int_of_char により、整数と文字の間の変換が可能です。 文字列は型 *string* を持ち、決まった長さ（$2^{24}-6$ 未満）の文字の列です。文字列を連結する演算子は^です。関数 int_of_string、string_of_int、string_of_float および float_of_string は、数値と文字列との間の様々な変換を行います。

3. 訳注：整数の場合
4. Not a Number

```
# 'B' ;;
- : char = 'B'
# int_of_char 'B' ;;
- : int = 66
# "is a string" ;;
- : string = "is a string"
# (string_of_int 1987) ^ " is the year Caml was created" ;;
- : string = "1987 is the year Caml was created"
```

たとえ文字列の中身が数字であっても、明示的な変換を行わない限り、数値に対する演算を使うことはできません。

```
# "1999" + 1 ;;
Characters 1-7:
  "1999" + 1 ;;
  ^^^^^^
This expression has type string but is here used with type int
# (int_of_string "1999") + 1 ;;
- : int = 2000
```

String モジュールには、文字列に対する多数の関数が集められています（217 ページ参照）。

論理値

論理値は型 *bool* を持ち、`true` と `false` という二つの値からなる集合に属します。論理値に対するプリミティブ演算を図 2.3 に示します。歴史的理由により、“and” と “or” の演算子にはそれぞれ二つの形があります。

`not`	否定	`&`	`&&`と同義
`&&`	逐次的 and	`or`	`||`と同義
`||`	逐次的 or		

図 2.3: 論理値に対する演算子

```
# true ;;
- : bool = true
# not true ;;
- : bool = false
# true && false ;;
- : bool = false
```

演算子`&&`および`||`（ないし同義の演算子）は、まず左側の引数を評価し、その結果によっては右側の引数を評価します。これらの演算子は条件文によって書き直すこともできます（18 ページ参照）。

等値演算子と比較演算子を図 2.4 に示します。これらの演算子は多相的です。つまり、整数の比較にも文字列の比較にも使うことができます。唯一の制約として、二つの引数は

=	構造等値	<	より小さい
==	物理等値	>	より大きい
<>	=の否定	<=	以下
!=	==の否定	>=	以上

図 2.4: 等値演算子および比較演算子

同じ型でなければなりません（28 ページ参照）。

```
# 1<=118 && (1=2 || not(1=2)) ;;
- : bool = true
# 1.0 <= 118.0 && (1.0 = 2.0 || not (1.0 = 2.0)) ;;
- : bool = true
# "one" < "two" ;;
- : bool = true
# 0 < '0' ;;
Characters 4-7:
  0 < '0' ;;
      ^^^
This expression has type char but is here used with type int
```

構造等値演算子は二つの値の構造を巡回してそれらの等しさを調べます。これに対して物理等値演算子は、二つの値がメモリの上で同じ場所におかれているかどうかを調べます。どちらの等値演算子も単純な値（論理値、文字、整数、および定数コンストラクタ）については同じ結果を返します（44 ページ参照）.

警告 浮動小数と文字列は構造を持つ値として扱われます。

ユニット

unit 型は、唯一の元 () を持つ集合を表します。

```
# () ;;
- : unit = ()
```

この値は手続き的プログラム（第 3 章 65 ページ参照）において、副作用を持つ関数のためによく使われます。Objective Caml には手続きの概念が存在しませんが、C 言語の void 型と同様に、値 () を結果とする関数により手続きを模倣します。

デカルト積、組

型が異なるかもしれない値を集めて、組にすることができます。組を作る値は、コンマによって区切ります。型コンストラクタ*は組を表します。型 *int * string* は、第 1 要素が整数（型 *int*）で、第 2 要素が文字列（型 *string*）であるような組の型です。

```
# ( 12 , "October" ) ;;
- : int * string = (12, "October")
```

曖昧さがないときは、より単純に

```
# 12 , "October" ;;
- : int * string = (12, "October")
```

と書くこともできます。関数 fst および snd により、二つ組の第 1 要素と第 2 要素にアクセスすることができます。

```
# fst ( 12 , "October" ) ;;
- : int = 12
# snd ( 12 , "October" ) ;;
- : string = "October"
```

この二つの関数はどのような型の値の二つ組でも受け付け、等値演算子と同様に多相的です。

```
# fst;;
- : 'a * 'b -> 'a = <fun>
# fst ( "October", 12 ) ;;
- : string = "October"
```

型 *int * char * string* は第 1 要素が型 *int* を持ち、第 2 要素が型 *char* を持ち、第 3 要素が型 *string* を持つような 3 つ組の型です。この型を持つ値は

```
# ( 65 , 'B' , "ascii" ) ;;
- : int * char * string = (65, 'B', "ascii")
```

のように書きます。

警告 関数 fst および snd を二つ組以外の組に適用すると、型エラーになります。

```
# snd ( 65 , 'B' , "ascii" ) ;;
Characters 7-25:
  snd ( 65 , 'B' , "ascii" ) ;;
        ^^^^^^^^^^^^^^^^^^
This expression has type int * char * string but is here used with type
  'a * 'b
```

実際に、二つ組の型と三つ組の型には違いがあります。型 *int * int * int* は、*(int * int) * int* や *int * (int * int)* のような型とは異なるのです。三つ組やその他の組にアクセスする関数は、言語の核のライブラリでは定義されていません。そのような関数は、必要であれば、パターンマッチングを利用して定義することができます（33 ページ参照）。

リスト

同じ型の値を集めて、リストにすることができます。リストは、空であるか、あるいは同じ型の要素から成ります。

```
# [] ;;
- : 'a list = []
# [ 1 ; 2 ; 3 ] ;;
- : int list = [1; 2; 3]
# [ 1 ; "two" ; 3 ] ;;
Characters 6-11:
  [ 1 ; "two" ; 3 ] ;;
        ^^^^^
```

```
This expression has type string but is here used with type int
```

リストの先頭に要素を追加する関数は、中置演算子 :: です。これは Lisp の *cons* と同様です。

```
# 1 :: 2 :: 3 :: [] ;;
- : int list = [1; 2; 3]
```

リストの連結も中置演算子`@`です。

```
# [ 1 ] @ [ 2 ; 3 ] ;;
- : int list = [1; 2; 3]
# [ 1 ; 2 ] @ [ 3 ] ;;
- : int list = [1; 2; 3]
```

その他のリスト操作関数は、`List` ライブラリに定義されています。このライブラリの関数 `hd` と `tl` は、リストの先頭と末尾を（もしそれらの値があれば）与えます。`List` モジュールに属していることをシステムに対して示すために、これらの関数は `List.hd` および `List.tl` と表されます[5]。

```
# List.hd [ 1 ; 2 ; 3 ] ;;
- : int = 1
# List.hd [] ;;
Exception: Failure "hd".
```

最後の例は、空リストの最初の要素を取得するように要求しており、実際には問題があります。システムが**例外**を発生するのは、まさにこのためです（53 ページ参照）。

条件制御構造

いかなるプログラミング言語でも必要不可欠な制御構造の一つとして、条件の関数として計算を誘導する、**条件文**（あるいは分岐）といわれる構造があります。

構文： **if** $expr_1$ **then** $expr_2$ **else** $expr_3$

式 $expr_1$ は型 *bool* を持ちます。式 $expr_2$ と $expr_3$ の型は何でも構いませんが、同じでなければいけません。

```
# if 3=4 then 0 else 4 ;;
- : int = 4
# if 3=4 then "0" else "4" ;;
- : string = "4"
# if 3=4 then 0 else "4";;
Characters 20-23:
  if 3=4 then 0 else "4";;
                     ^^^
```

5. List モジュールについては 217 ページで説明します。

```
This expression has type string but is here used with type int
```

条件構文はそれ自体も式であり、評価すると値を返します。

```
# (if 3=5 then 8 else 10) + 5 ;;
- : int = 15
```

注意

else 節は省略することができますが、その場合は暗黙に **else** () とおきかえられます。したがって、式 $expr_2$ の型は *unit* でなければいけません（77 ページ参照）。

値の宣言

宣言は、名前を値に束縛します。宣言には、グローバルな宣言とローカルな宣言の二種類があります。前者の場合、宣言された名前は後に続くすべての式で使えます。後者の場合、宣言された名前は一つの式でしか使えません。複数の名前と値の束縛を**同時に**宣言することも同様に可能です。

グローバルな宣言

構文： **let** *name* **=** *expr* **;;**

グローバルな宣言は、名前 *name* と式 *expr* の値との束縛を定義します。その束縛は、後に続くすべての式で用いることができます。

```
# let yr = "1999" ;;
val yr : string = "1999"
# let x = int_of_string(yr) ;;
val x : int = 1999
# x ;;
- : int = 1999
# x + 1 ;;
- : int = 2000
# let new_yr = string_of_int (x + 1) ;;
val new_yr : string = "2000"
```

グローバルな同時宣言

構文：

let $name_1$ **=** $expr_1$
and $name_2$ **=** $expr_2$
⋮
and $name_n$ **=** $expr_n$ **;;**

同時宣言は、異なるシンボルを同じレベルで宣言します。それらのシンボルは、すべての宣言が終わるまで使うことができません。

```
# let x = 1 and y = 2 ;;
val x : int = 1
val y : int = 2
# x + y ;;
- : int = 3
# let z = 3 and t = z + 2 ;;
Characters 18-19:
  let z = 3 and t = z + 2 ;;
                    ^
Unbound value z
```

いくつかのグローバルな宣言を一つの節にまとめることもできます。この場合、それらの宣言の型や値は、二重の "**;;**" で節が終わるまで表示されません。同時宣言と異なり、これらの宣言は逐次に評価されます。

```
# let x = 2
  let y = x + 3  ;;
val x : int = 2
val y : int = 5
```

グローバルな宣言は、同じ名前の新しい宣言により隠すこともできます（26 ページ参照）。

ローカルな宣言

構文 : **let** *name* **=** $expr_1$ **in** $expr_2$**;;**

このローカルな宣言は、名前 *name* を $expr_1$ の値に束縛します。その名前は $expr_2$ を評価する間のみ使えます。

```
# let xl = 3 in xl * xl ;;
- : int = 9
```

この xl を値 3 に束縛するローカルな宣言は、`xl * xl` を評価するときのみ有効です。

```
# xl ;;
Characters 1-3:
  xl ;;
  ^^
Unbound value xl
```

ローカルな宣言は、同じ名前に対する以前の宣言をすべて隠しますが、スコープを抜けると前の値に戻ります。

```
# let x = 2 ;;
val x : int = 2
# let x = 3 in x * x ;;
- : int = 9
# x * x ;;
- : int = 4
```

ローカルな宣言は式であり、他の式を構成するために利用することができます。

```
# (let x = 3 in x * x) + 1 ;;
```

```
- : int = 10
```

ローカルな宣言も同時に行うことができます。

構文：

```
let   name1 = expr1
and   name2 = expr2
⋮
and   namen = exprn
in    expr ;;
```

```
# let a = 3.0 and b = 4.0 in sqrt (a*.a +. b*.b) ;;
- : float = 5
# b ;;
Characters 0-1:
  b ;;
  ^
Unbound value b
```

関数式、関数

関数式は**引数**と**本体**からなります。仮引数は変数名であり、本体は式です。この引数は**抽象的**であると言います。この理由から、関数式は**抽象**とも呼びます。

構文： **function** *p* -> *expr*

したがって、引数を二乗する関数は

```
# function x → x*x ;;
- : int -> int = <fun>
```

と書きます。この関数の型は、Objective Caml のシステムが推論します。**関数型** *int* -> *int* は、型 *int* の引数を期待し、型 *int* の値を返す関数を示します。

関数を引数に適用するには、関数の後に引数を書きます。

```
# (function x → x * x) 5 ;;
- : int = 25
```

適用を評価するということは、仮引数 x を引数の値（すなわち**実引数**）、ここでは 5 で置き換え、関数の本体、ここでは *x * x* を評価することになります。

関数式を構成するとき、*expr* はいかなる式でも構いません。特に、*expr* 自体が関数式でも構いません。

```
# function x → (function y → 3*x + y) ;;
- : int -> int -> int = <fun>
```

括弧はなくても構いません。より単純に、次のように書くこともできます。

```
# function x → function y → 3*x + y ;;
- : int -> int -> int = <fun>
```

この式の型は、二つの整数を期待し、一つの整数を返す関数の型として普通に読むこともできます。しかし Objective Caml のような関数型言語の文脈では、より正確に、一つの整数を期待し、型 *int -> int* の**関数値**を返す関数の型として扱います。

```
# (function x → function y → 3*x + y) 5 ;;
- : int -> int = <fun>
```

もちろん、この関数式を普通に二つの引数に適用することもできます。

```
# (function x → function y → 3*x + y) 4 5 ;;
- : int = 17
```

と書きます。*f a b* と書くときは左側に暗黙の括弧がつくので、この式は (*f a*) *b* と等価です。

(function *x* → **function** *y* → 3**x* + *y*) 4 5

なる適用について詳しく調べましょう。この式の値を計算するには、

(function *x* → **function** *y* → 3**x* + *y*) 4

の値を計算する必要がありますが、これは 3**x* + *y* において *x* を 4 で置き換えて得られる

function *y* → 3*4 + *y*

に等価な**関数式**です。この値（関数）を 5 に適用すると、最終的な値 3*4+5 = 17 が得られます。

```
# (function x → function y → 3*x + y) 4 5 ;;
- : int = 17
```

関数のアリティ

関数の引数の数は、その関数のアリティと呼ばれます。数学に由来する記法では、$f(4,5)$ のように、関数の名前の後で括弧の中に関数の引数を書くことになっています。先に見たように、Objective Caml では $f\ 4\ 5$ のように書くほうが普通です。もちろん、Objective Caml でも $(4,5)$ に適用できるような関数を書くことは可能です。

```
# function (x,y) → 3*x + y ;;
- : int * int -> int = <fun>
```

しかし、型が示すように、この式は二つではなく一つの引数、すなわち整数のペアを期待しています。一つのペアを期待している関数に二つの引数を渡そうとしたり、二つの引数を期待している関数に一つのペアを渡そうとすると、型エラーになります。

```
# (function (x,y) → 3*x + y) 4 5 ;;
```

```
Characters 2-27:
  (function (x,y) -> 3*x + y) 4 5 ;;
   ^^^^^^^^^^^^^^^^^^^^^^^^^
This function is applied to too many arguments
# (function x → function y → 3*x + y) (4, 5) ;;
Characters 39-43:
  (function x -> function y -> 3*x + y) (4, 5) ;;
                                        ^^^^
This expression has type int * int but is here used with type int
```

別の構文

複数の引数を持つ関数式を、もっと簡潔に書く方法もあります。これは Caml 言語の以前のバージョンの遺物で、次のような形をしています。

構文： **fun** $p_1 \ldots p_n$ -> *expr*

これにより、**function** キーワードや矢印の繰り返しを省くことができます。これは次の言い換えと同じです。

function p_1 -> ...-> **function** p_n -> *expr*

```
# fun x y → 3*x + y ;;
- : int -> int -> int = <fun>
# (fun x y → 3*x + y) 4 5 ;;
- : int = 17
```

この形は、特に Objective Caml と一緒に配布されているライブラリの中で、今でもよく出てくることがあります。

クロージャ

Objective Caml は関数式を他の式と同様に扱うことができ、関数式の値を計算することができます。そのような計算によって返される値は関数式であり、**クロージャ**と呼ばれます。Objective Caml のすべての式は、その式より前の宣言に由来する名前と値の束縛からなる**環境**の元で評価されます。クロージャは仮引数の名前、関数の本体、および式の環境の 3 つ組として表すことができます。関数式の本体は、仮引数の他に、先に宣言されたすべての変数を使うことができるので、そのような環境を保存する必要があります。それらの変数は、その関数式において「自由」であるといいます。自由変数の値は、関数式が適用されるときに必要となります。

```
# let m = 3 ;;
val m : int = 3
# function x → x + m ;;
- : int -> int = <fun>
# (function x → x + m) 5 ;;
- : int = 8
```

クロージャを引数に適用して新しいクロージャが返されるとき、新しいクロージャは将来の適用で必要なすべての束縛を環境に保持します。この概念については、変数のスコープに関する節で詳細に説明します（26 ページ参照）。クロージャのメモリ表現については、後で 4 章（101 ページ）および 12 章（334 ページ）において扱います。

今まで扱ってきた関数式は**匿名**でした。関数式に名前をつけられると便利です。

関数値の宣言

関数値は、言語の他の値と同じように、**let** 構文で宣言します。

```
# let succ = function x → x + 1 ;;
val succ : int -> int = <fun>
# succ 420 ;;
- : int = 421
# let g = function x → function y → 2*x + 3*y ;;
val g : int -> int -> int = <fun>
# g 1 2;;
- : int = 8
```

記述を簡単にするために、次のような書き方が許されています。

構文 : **let** *name* p_1 ... p_n **=** *expr*

これは次の形と同じです。

let *name* **= function** p_1 -> ...-> **function** p_n -> *expr*

次の succ と g の宣言は、先の宣言と等価です。

```
# let succ x = x + 1 ;;
val succ : int -> int = <fun>
# let g x y = 2*x + 3*y ;;
val g : int -> int -> int = <fun>
```

次の例では、Objective Caml の完全に関数的な特徴が発揮されています。この例では、g を一つの整数に適用することにより関数 h1 が得られています。このような場合を**部分適用**と言います。

```
# let h1 = g 1 ;;
val h1 : int -> int = <fun>
# h1 2 ;;
- : int = 8
```

g において第 2 引数 *y* の値を固定し、関数 *h2* を定義することもできます。

```
# let h2 = function x → g x 2 ;;
val h2 : int -> int = <fun>
```

```
# h2 1 ;;
- : int = 8
```

インフィックス関数の宣言

ある種の二引数関数は、インフィックス（中置）形式で適用できます。整数の加算がそうです。+を 3 と 5 に適用するには、3 + 5 と書きます。記号+を普通の関数値として使用するには、そのインフィックス記号を括弧でかこむ構文により指示する必要があります。その構文は次の通りです。

構文： (op)

次の例は、(+) を使って関数 succ を定義します。

```
# ( + ) ;;
- : int -> int -> int = <fun>
# let succ = ( + ) 1 ;;
val succ : int -> int = <fun>
# succ 3 ;;
- : int = 4
```

新しい演算子を定義することも可能です。整数のペアを加算する演算子++を定義します。

```
# let ( ++ ) c1 c2 = (fst c1)+(fst c2), (snd c1)+(snd c2) ;;
val ( ++ ) : int * int -> int * int -> int * int = <fun>
# let c = (2,3) ;;
val c : int * int = (2, 3)
# c ++ c ;;
- : int * int = (4, 6)
```

このような定義の可能な演算子には、重要な制限があります。*、+、@といった記号のみを含み、文字や数字を含んではいけないという制限です。元からインフィックスとして定義されているある種の関数は、この規則の例外です。それらを列挙すると、次の通りです：or mod land lor lxor lsl lsr asr。

高階関数

関数値（クロージャ）は結果として返すことができます。同様に、引数として関数に渡すこともできます。関数値を引数としてとったり結果として返したりする関数を**高階関数**と呼びます。

```
# let h = function f → function y → (f y) + y ;;
val h : (int -> int) -> int -> int = <fun>
```

注意

適用は左側に暗黙の括弧がつきますが、関数型は右側に暗黙の括弧がつきます。したがって、関数 h の型は
(int -> int) -> int -> int すなわち *(int -> int) -> (int -> int)*
と書くことができます。

高階関数を使うと、リストをエレガントに扱うことができます。たとえば関数 List.map は、リストのすべての要素に一つの関数を適用し、結果をリストにして返します。

```
# List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# let square x = string_of_int (x*x) ;;
val square : int -> string = <fun>
# List.map square [1; 2; 3; 4] ;;
- : string list = ["1"; "4"; "9"; "16"]
```

他の例として、関数 List.for_all は、リストのすべての要素が与えられた基準を満たすかどうか調査します。

```
# List.for_all ;;
- : ('a -> bool) -> 'a list -> bool = <fun>
# List.for_all (function n → n<>0) [-3; -2; -1; 1; 2; 3] ;;
- : bool = true
# List.for_all (function n → n<>0) [-3; -2; 0; 1; 2; 3] ;;
- : bool = false
```

変数のスコープ

式を評価するためには、その式に現れるすべての変数が定義されていなければなりません。これは特に宣言 **let** *p = e* における式 e に当てはまります。しかし、式 e の中で変数 p はまだ知られていないので、前の宣言で与えられた値を参照する場合しか使えません。

```
# let p = p ^ "-suffix" ;;
Characters 9-10:
  let p = p ^ "-suffix" ;;
          ^
Unbound value p
# let p = "prefix" ;;
val p : string = "prefix"
# let p = p ^ "-suffix" ;;
val p : string = "prefix-suffix"
```

Objective Caml では、変数は静的に束縛されます。クロージャを適用するときに使用される環境は、そのクロージャを宣言した時点のもの（静的スコープ）であり、適用する時点のもの（動的スコープ）ではありません。

```
# let p = 10 ;;
val p : int = 10
# let k x = (x, p, x+p) ;;
val k : int -> int * int * int = <fun>
# k p ;;
- : int * int * int = (10, 10, 20)
# let p = 1000 ;;
val p : int = 1000
# k p ;;
- : int * int * int = (1000, 10, 1010)
```

関数 k は自由変数 p を含んでいます。p はグローバル環境で定義されているので、k の定義は合法です。クロージャk の環境における、名前 p と値 10 の束縛は静的です。すなわち、それ以降の p の定義には依存しません。

再帰的定義

定義において自分自身の識別子を使用する変数宣言は**再帰的**であるといいます。この機能は主に関数のため、特に漸化式による定義を模倣するために使用します。このような定義を **let** 宣言がサポートしないことは先ほど見ました。再帰的関数を宣言するには、専用の構文要素を使用します。

構文： **let rec** *name* **=** *expr* **;;**

同様に、関数の引数を指定して関数値を定義する構文機能を使用することもできます。

構文： **let rec** *name* $p_1 \ldots p_n$ **=** *expr* **;;**

たとえば、0 から引数までの非負整数の総和を計算する関数 sigma はこのようになります。

```
# let rec sigma x = if x = 0 then 0 else x + sigma (x-1) ;;
val sigma : int -> int = <fun>
# sigma 10 ;;
- : int = 55
```

なお、この関数は引数が真に負だと停止しません。

一般に再帰的な値は関数です。コンパイラは、値が関数でない再帰的宣言を拒否します。

```
# let rec x = x + 1 ;;
Characters 13-18:
  let rec x = x + 1 ;;
              ^^^^^
This kind of expression is not allowed as right-hand side of ‘let rec’
```

ただし、ある種の場合にはこのような宣言が許されることもあります（52 ページ参照）。

let rec 宣言は **and** 構文と組み合わせて、同時に宣言を行うこともできます。この場合、同じレベルで定義したすべての関数は、互いの本体で使うことができます。これを用いると特に**相互再帰的**な関数の宣言が可能です。

```
# let rec even n = (n<>1) && ((n=0) or (odd (n-1)))
```

```
  and    odd  n = (n<>0) && ((n=1) or (even (n-1)))    ;;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>
# even 4 ;;
- : bool = true
# odd 5 ;;
- : bool = true
```

同様に、ローカルな宣言も再帰的に行うことができます。次に示す新しい sigma の定義は、引数の正当性をテストしてから、ローカル関数 sigma_rec の定義する総和を計算します。

```
# let sigma x =
    let rec sigma_rec x =  if x = 0 then 0 else x + sigma_rec (x-1) in
     if (x<0) then "error: negative argument"
     else "sigma = " ^ (string_of_int (sigma_rec x)) ;;
val sigma : int -> string = <fun>
```

注意

引数が負であろうとなかろうと同じ型の値を返す必要があるので、結果は文字列の形で与えざるを得ません。実際、引数が負の場合、sigma はどんな値を返すべきでしょうか。この問題に適切に対処する方法については後で述べます（53 ページ参照）。

多相性と型制約

ある種の関数は異なる型を持つ引数に対して同じコードを実行します。たとえば、二つの値からペアを作る際に、システムが知っている一つ一つの型について、別々の関数が必要となることはありません[6]。同様に、ペアの第一要素をアクセスする関数も、その第一要素の型によって区別する必要はありません。

```
# let make_pair a b = (a,b) ;;
val make_pair : 'a -> 'b -> 'a * 'b = <fun>
# let p = make_pair "paper" 451 ;;
val p : string * int = ("paper", 451)
# let a = make_pair 'B' 65 ;;
val a : char * int = ('B', 65)
# fst p ;;
- : string = "paper"
# fst a ;;
- : char = 'B'
```

返り値や引数の型を特定しない関数は、多相的であるといいます。Objective Caml のコンパイラに含まれる型推論器は、一つ一つの式についてもっとも一般的な型を求めます。

6. これは幸いなことです。というのは、型の数はマシンの記憶容量によってしか制限されないからです。

この場合、Objective Caml は変数（ここでは*'a* と*'b*）を用いて、そのような一般的な型を示します。これらの変数は、関数を適用するときに引数の型によって具体化されます。

Objective Caml の多相関数を用いれば、静的型付けによる実行安全性を維持しつつ、すべての型について使える一般的なコードを書くことができる、という利点が得られます。実際、make_pair は多相的ですが、(*make_pair* 'B' 65) によって作られる値は、(*make_pair* "paper" 451) とは異なる、適切に特定された型を持ちます。さらに、型の検証はコンパイルの際に行われるので、コードの一般性がプログラムの効率性を損なうことはありません。

多相的な関数や値の例

以下に、「パラメータ化された型を持つ関数」をパラメータとする多相関数の例を挙げます。

app 関数は、関数を引数に適用します。

```
# let app = function f → function x → f x ;;
val app : ('a -> 'b) -> 'a -> 'b = <fun>
```

よって、以前に定義した関数 odd に適用することが可能です。

```
# app odd 2;;
- : bool = false
```

恒等関数 (id) はパラメータをとり、そのまま返します。

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
# app id  1 ;;
- : int = 1
```

compose 関数は二つの関数と一つの値をとり、関数を合成して値に適用します。

```
# let compose f g  x = f (g x) ;;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# let add1 x = x+1 and mul5 x = x*5 in compose  mul5 add1  9 ;;
- : int = 50
```

g の結果は f の引数と同じ型でなければならないことがわかります。

関数以外の値も多相的になることがあります。たとえば、空リストの場合がそうです。

```
# let l = [] ;;
val l : 'a list = []
```

型推論は関数適用に由来する制約の解消によって行われるのであり、実行時に得られる値によって行われるわけではありません。次の例はそれを実際に証明しています。

```
# let t = List.tl [2] ;;
val t : int list = []
```

List.tl の型は*'a list -> 'a list* なので、この関数を整数のリストに適用すると、

整数のリストを返します。実行して得られるのが空リストであるという事実があっても、型はまったく変わりません。

Objective Caml は、引数の形を使わないすべての関数に対して、パラメータ化された型を与えます。このような多相性を**パラメータ的多相性**といいます[7]。

型制約

Caml の型推論器はもっとも一般的な型を与えるので、式の型を指定することが便利ないし必要な場合もあります。

型制約の構文形式は次の通りです。

構文 : (*expr* : *t*)

型推論器は、このような制約に出会うと、式の型を構成する際に考慮します。型制約を使うことにより、次のようなことができます。

- 関数のパラメータの型を見やすくする。
- 意図した文脈以外で関数を使えないようにする。
- 式の型を指定する。これは特に変更可能な値に対して有用です（66 ページ参照）。

次の例は、そのような型制約の使用法を示しています。

```
# let add (x:int) (y:int) = x + y ;;
val add : int -> int -> int = <fun>
# let make_pair_int (x:int) (y:int) = x,y;;
val make_pair_int : int -> int -> int * int = <fun>
# let compose_fn_int (f : int → int) (g : int → int) (x:int) =
    compose f g  x;;
val compose_fn_int : (int -> int) -> (int -> int) -> int -> int = <fun>
# let nil = ( []  : string list);;
val nil : string list = []
# 'H' :: nil;;
Characters 5-8:
  'H'::nil;;
       ^^^
This expression has type string list but is here used with type char list
```

このようにして多相性を制限することにより、システムが推論する型を制約し、式の型をよりよく制御することができます。次の例が示すように、型変数を含む場合も含めて、定義されているいかなる型も使用可能です。

```
# let llnil = ( []  : 'a list list) ;;
val llnil : 'a list list = []
# [1;2;3]:: llnil ;;
```

7. いくつかのあらかじめ定義された関数はこの規則にしたがいません。特に、構造等値関数（=）は多相的（型が 'a -> 'a -> bool）ですが、引数が等値かどうかテストするために、その構造を探索します。

```
- : int list list = [[1; 2; 3]]
```
記号 `llnil` は任意の型のリストのリストになります。

ここでは制約の話をしているのであって、Objective Caml の型推論を明示的な型付けで置き換えているわけではありません。特に、推論が許す範囲を越えて型を一般化することはできません。

```
# let add_general (x:'a) (y:'b) = add x y ;;
val add_general : int -> int -> int = <fun>
```

型制約はモジュールインターフェース (第 14 章参照) およびクラス定義 (第 15 章参照) でも使用されます。

例

この節では、やや複雑な関数の例をいくつか挙げます。これらの関数のほとんどは、元から Objective Caml で定義されています。「教育」のために、それらを再定義します。

ここでは、再帰関数の最終的な場合わけを条件文によって実装します。したがって、Lisp に近いプログラミングスタイルとなります。より ML 風の定義の方法については、場合わけによって関数を定義する別の方法を説明する際に述べます（33 ページ参照）。

リストの長さ

まず、リストが空かどうかをテストする関数 `null` から始めましょう。

```
# let null l = (l = [] ) ;;
val null : 'a list -> bool = <fun>
```
次に、リストの長さ（すなわち要素の数）を計算する関数 `size` を定義します。

```
# let rec size l =
    if null l then 0
    else 1 + (size (List.tl l)) ;;
val size : 'a list -> int = <fun>
# size [] ;;
- : int = 0
# size [1;2;18;22] ;;
- : int = 4
```
関数 `size` は引数が空かどうかをテストします。もし空ならば `0` を返し、空でなければリストの末尾の長さを計算し、得られた値に `1` をプラスして返します。

合成の繰り返し

式 *iterate n f* は `f` を n 回繰り返した値を計算します。

```
# let rec iterate n f =
    if n = 0 then (function x → x)
    else compose f (iterate (n-1) f) ;;
val iterate : int -> ('a -> 'a) -> 'a -> 'a = <fun>
```

関数 iterate は n が 0 かどうかテストして、もしそうならば恒等関数を返し、そうでなければ f を n-1 回繰り返した関数を f と合成します。

iterate を用いると、べき乗を乗算の繰り返しとして定義することができます。

```
# let rec power i n =
   let i_times = ( * ) i in
    iterate n i_times 1 ;;
val power : int -> int -> int = <fun>
# power 2 8 ;;
- : int = 256
```

power 関数は関数式 i_times を n 回繰り返してから、その結果を 1 に適用します。そうすれば確かに整数の n 乗が求まります。

乗算表

引数として渡された整数の乗算表を計算する関数 *multab* を書くことにします。

まず、関数 apply_fun_list を定義し、*f_list* が関数のリストならば、*apply_fun_list x f_list* は *f_list* の各要素を *x* に適用した結果のリストとなるようにします。

```
# let rec apply_fun_list x f_list =
    if null f_list then []
    else ((List.hd f_list) x) :: (apply_fun_list x (List.tl f_list)) ;;
val apply_fun_list : 'a -> ('a -> 'b) list -> 'b list = <fun>
# apply_fun_list 1 [( + ) 1;( + ) 2;( + ) 3] ;;
- : int list = [2; 3; 4]
```

関数 *mk_mult_fun_list* は、0 から n まで変化する i について、引数を i 倍する関数のリストを返します。

```
# let mk_mult_fun_list n =
     let rec mmfl_aux p =
       if p = n then [ ( * ) n ]
       else (( * ) p) :: (mmfl_aux (p+1))
     in (mmfl_aux 1) ;;
val mk_mult_fun_list : int -> (int -> int) list = <fun>
```

7 の乗算表は次のように得られます。

```
# let multab n = apply_fun_list n (mk_mult_fun_list 10) ;;
val multab : int -> int list = <fun>
# multab 7 ;;
- : int list = [7; 14; 21; 28; 35; 42; 49; 56; 63; 70]
```

リストに対する繰り返し

関数呼び出し *fold_left f a [e1; e2; ... ; en]* は *f ... (f (f a e1) e2) ... en* を返します。したがって、n 回の適用が行われます。

```
# let rec fold_left f a l =
    if null l then a
    else fold_left f ( f a (List.hd l)) (List.tl l) ;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

関数 fold_left を使うと、整数のリストの要素の和を計算する関数を簡潔に定義することができます。

```
# let sum_list = fold_left (+) 0 ;;
val sum_list : int list -> int = <fun>
# sum_list [2;4;7] ;;
- : int = 13
```

あるいは、文字列のリストの要素の連結を計算することもできます。

```
# let concat_list = fold_left (^) "" ;;
val concat_list : string list -> string = <fun>
# concat_list ["Hello "; "world" ; "!"] ;;
- : string = "Hello world!"
```

型宣言とパターンマッチング

Objective Caml で元から定義されている型を使えば、組やリストからデータを作ることはできますが、ある種のデータ構造を記述するためには、新しい型を定義する必要があります。Objective Caml では、型宣言は再帰的であり、型 *'a list* と同様の感覚で、型変数によりパラメータ化できます。型推論は新しい宣言を考慮して、式の型を提示します。新しい型の値を作るには、型を定義するときに記述した**コンストラクタ**を使います。ML 系の言語の特徴として、**パターンマッチング**があります。それにより、複雑なデータ構造の要素を簡単にアクセスすることができます。関数定義は往々にして一つの引数に対するパターンマッチングに相当し、場合わけによる関数の定義を可能にします。

我々はまず、元から定義されている型に対するパターンマッチングを示し、それから構造型を宣言したり、そのような型の値を構成したり、それらの要素をパターンマッチングでアクセスする様々な方法について説明します。

パターンマッチング

パターンは厳密には Objective Caml の式ではありません。プリミティブ型の定数 (*int*, *bool*, *char*, ...)、変数、コンストラクタ、ワイルドカードパターンと呼ばれる記号_などの要素を（文法および型の観点から）正しく並べたものです。それ以外の記号もパターンを書くのに使います。我々は必要な範囲で、それらを紹介します。

パターンマッチングは値に対して適用され、値の形を認識して、それに応じて計算を導くために使われます。このために、それぞれのパターンに対して、計算する式を関連づけます。

構文：

```
match expr with
| p1 -> expr1
⋮
| pn -> exprn
```

式 *expr* は異なるパターン $p_1, \ldots, p_n$ に対して逐次的にマッチされます。もしそれらのパターンの一つ（たとえば p_i）が *expr* の値に合致すれば、対応する計算分岐 ($expr_i$) が評価されます。これらの異なるパターン p_i は同じ型を持ちます。異なる式 $expr_i$ についても同様です。最初のパターンの前にある縦棒は省くこともできます。

例

パターンマッチングを用いて、論理的含意を実装する型 *(bool * bool) -> bool* の関数 imply を定義する方法を、二通り示します。2 つ組にマッチするパターンは (,) という形をしています。

最初の方法では、真理値表と同様に、すべての可能性を列挙します。

```
# let imply v = match v with
      (true,true)   → true
    | (true,false)  → false
    | (false,true)  → true
    | (false,false) → true;;
val imply : bool * bool -> bool = <fun>
```

変数を使って数個のケースをまとめれば、より簡潔な定義が得られます。

```
# let imply v = match v with
      (true,x)  → x
    | (false,x) → true;;
val imply : bool * bool -> bool = <fun>
```

これら二つの imply は同じ関数を計算します。すなわち、同じ入力に対しては同じ値を返します。

線形パターン

パターンは必ず**線形**でなければいけません。すなわち、どの変数も、マッチされるパターンの中に高々一回しか現われることはできません[8]。したがって、

```
# let equal c = match c with
     (x,x) → true
   | (x,y) → false;;
Characters 35-36:
    (x,x) -> true
        ^
This variable is bound several times in this matching
```

8. 訳注：バージョン 3.01 からサポートされた、OR パターンの中の変数を除く

のような書き方をしたいと思ったかもしれませんが、これはエラーになります。もしそのような書き方ができたとすると、コンパイラは等値検査のやり方を知らなければなりません。けれども、そうすると直ちに多くの問題が発生します。もし値の間の物理等値を用いたら、あまりにも弱いシステムを得ることになります。たとえば、リスト [1; 2] が二回出現しても、それらが等値であることを認識できません。かといって、もし構造等値を使うことにしたら、循環構造を永久に探索する危険をおかすことになります。たとえば再帰関数は循環構造ですが、関数以外の再帰的な（すなわち循環的な）値を構成することも可能です（52 ページ参照）。

ワイルドカードパターン

記号_はありとあらゆる値にマッチします。これはワイルドカードパターンと呼ばれます。ワイルドカードパターンは、複雑な型にマッチするように使うことができます。たとえば関数 imply の定義をさらに簡単にするために使います。

```
# let imply v = match v with
          (true,false) → false
        |   _          → true;;
val imply : bool * bool -> bool = <fun>
```

パターンマッチングによる定義は、マッチされる値として可能なすべての場合を処理しなければなりません。さもないと、コンパイラは警告メッセージを表示します。

```
# let is_zero n = match n with  0 → true ;;
Characters 17-40:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
  let is_zero n = match n with  0 -> true ;;
                  ^^^^^^^^^^^^^^^^^^^^^^^
val is_zero : int -> bool = <fun>
```

実際、もし実引数が 0 と異なっていたら、この関数は何の値を返せばよいのかわかりません。そこで、ワイルドカードパターンを使用して場合わけを完全にすることができます。

```
# let is_zero n = match n with
      0 → true
    | _ → false ;;
val is_zero : int -> bool = <fun>
```

もし実行時にどのパターンも選択されなければ、例外が起こります。したがって、次のように書くこともできます。

```
# let f x = match x with 1 → 3 ;;
Characters 11-30:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
0
```

```
  let f x = match x with 1 -> 3 ;;
            ^^^^^^^^^^^^^^^^^^^
val f : int -> int = <fun>
# f 1 ;;
- : int = 3
# f 4 ;;
Exception: Match_failure ("", 11, 30).
```

この Match_Failure 例外は f 4 の呼び出しによって起こり、もし処理されなければ、進行中の計算を停止します（53 ページ参照）。

パターンの組み合わせ

複数のパターンを組み合わせることにより、元のパターンのいずれかにしたがって値にマッチする、新しいパターンを得ることができます。その構文形式は次の通りです。

構文： p_1 **|** ... **|** p_n

これはパターン $p_1, \ldots, p_n$ を組み合わせて、新しいパターンを作ります。唯一の強い制約として、これらのパターンにおいては、すべての名前づけが禁止されています[9]。したがって、これらのパターンの一つ一つは、定数値かワイルドカードパターンしか含むことができません。次の例は、ある文字が母音かどうか確かめる方法を示しています。

```
# let is_a_vowel c = match c with
      'a' | 'e' | 'i' | 'o' | 'u' | 'y' → true
    | _ → false ;;
val is_a_vowel : char -> bool = <fun>
# is_a_vowel 'i' ;;
- : bool = true
# is_a_vowel 'j' ;;
- : bool = false
```

パラメータのパターンマッチング

場合わけによる関数の定義は、パターンマッチングの本質的な使い方の一つです。そのような定義を簡単に書くために、構文 **function** は引数のパターンマッチを許しています。

構文：

```
function | p1 -> expr1
         | p2 -> expr2
           ⋮
         | pn -> exprn
```

一つ目のパターンの前にある縦棒は、ここでも省略可能です。実際、我々はまるでジョルダン氏のように[10]、関数を定義するたびにパターンマッチングを使用します。事実、構

9. 訳注：バージョン 3.01 以降では可能

10. 原訳注：モリエールの劇 *Le Bourgeois Gentilhomme*（**町人貴族**）において、登場人物のジョルダン氏は生まれて以来、ずっと散文詩調で話していたことに気づいて驚きます。この劇は

文 function x -> expression は、一個の変数に縮退した単一のパターンを使用する、パターンマッチングによる定義です。この仕様は、

```
# let f = function (x,y) → 2*x + 3*y + 4 ;;
val f : int * int -> int = <fun>
```

のように単純なパターンで使用することもできます。

実際、構文

$$\textbf{function}\ p_1\ \text{->}\ expr_1\ \mathbf{|}\ \ldots\ \mathbf{|}\ p_n\ \text{->}\ expr_n$$

は

$$\textbf{function}\ expr\ \text{->}\ \textbf{match}\ \text{expr}\ \textbf{with}\ p_1\ \text{->}\ expr_1\ \mathbf{|}\ \ldots\ \mathbf{|}\ p_n\ \text{->}\ expr_n$$

と等価です。24 ページで触れた宣言の等価性を用いて、

```
# let f (x,y) = 2*x + 3*y + 4 ;;
val f : int * int -> int = <fun>
```

と書くこともできます。しかし、マッチされる値がコンストラクタを一つしか持たない型でないと、こういう自然な書き方はできません。さもないと、パターンマッチングが完全ではなくなってしまいます。

```
# let is_zero 0 = true ;;
Characters 13-21:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
  let is_zero 0 = true ;;
              ^^^^^^^^
val is_zero : int -> bool = <fun>
```

マッチされる値の名前づけ

パターンマッチングの際、パターンの一部や全体に名前をつけると便利なことがあります。次の構文形式は、名前をパターンに束縛するキーワード **as** を導入します。

構文 : (*p* **as** *name*)

これは、全体性を保ちつつ値を分解する必要があるときに便利です。次の例において、関数 min_rat は有理数の組を受け取り、小さいほうの有理数を与えます。それぞれの有理数は分子と分母の組により表されています。

```
# let min_rat pr = match pr with
```

リンク: http://www.site-moliere.com/pieces/bourgeoi.htm
で読むことができます。また、
リンク: http://moliere-in-english.com/bourgeois.html
には、その部分を含め、英訳の抜粋があります。

```
    ((_,0),p2) → p2
  | (p1,(_,0)) → p1
  | (((n1,d1) as r1), ((n2,d2) as r2)) →
        if (n1 * d2) < (n2 * d1) then r1 else r2;;
val min_rat : (int * int) * (int * int) -> int * int = <fun>
```

二つの有理数を比較するには、分解して分子と分母（n1, n2, d1 および d2）に名前をつける必要がありますが、元の組（r1 または r2）も返さなければなりません。**as** 構文は、このように単一の値の一部に名前をつけることを可能とします。これにより、結果として返す有理数を再構築する必要がなくなります。

ガードつきパターンマッチング

ガードつきパターンマッチングは、パターンがマッチされた直後に条件式を評価することに相当します。もし条件式が `true` を返せば、パターンに関連づけられた式が評価され、さもなければ次のパターンへパターンマッチングが続きます。

構文 :

```
match expr with
  ⋮
  | pi when condi -> expri
  ⋮
```

次の例は二つの条件を用いて、二つの有理数の等しさをテストします。

```
# let eq_rat cr = match cr with
    ((_,0),(_,0)) → true
  | ((_,0),_) → false
  | (_,(_,0)) → false
  | ((n1,1), (n2,1)) when n1 = n2 → true
  | ((n1,d1), (n2,d2)) when ((n1 * d2) = (n2 * d1)) → true
  | _ → false;;
val eq_rat : (int * int) * (int * int) -> bool = <fun>
```

もし第 4 のパターンがマッチしても、ガードが失敗したら、マッチングは第 5 のパターンに続きます。

注意

パターンマッチングが完全かどうか Objective Caml が行う検証は、ガードの条件式が偽かもしれないと仮定します。結果的に、そのようなパターンは考慮されません。ガードが充足されるかどうか実行前に知ることは不可能なためです。

次の例では、パターンマッチングが完全かどうか、検出することができません。

```
# let f = function x when x = x → true;;
Characters 10-40:
Warning: Bad style, all clauses in this pattern-matching are guarded.
   let f = function x when x = x -> true;;
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
val f : 'a -> bool = <fun>
```

文字の区間に対するパターンマッチング

文字に対するパターンマッチングを行う際に、文字区間に対応するすべてのパターンの組み合わせを作るのは面倒です。実際、ある文字が英字かどうか検査するのにさえ、少なくとも 26 個のパターンを書いて組み合わせないといけません。そこで、文字について Objective Caml では

構文： `'c1' .. 'cn'`

という形のパターンを書くことができます。これは `'c1' | 'c2' | ... | 'cn'` という組み合わせと等価です。

たとえばパターン `'0' .. '9'` はパターン `'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'` に対応します。最初の形のほうが読みやすく、かつ書きやすくなっています。

警告　この仕様は言語拡張の一部であり、将来のバージョンでは変更されるかもしれません。

パターンの組み合わせと区間を用いて、いくつかの基準にしたがって文字を分類する関数を定義します。

```
# let char_discriminate c = match c with
      'a' | 'e' | 'i' | 'o' | 'u' | 'y'
    | 'A' | 'E' | 'I' | 'O' | 'U' | 'Y'  → "Vowel"
    | 'a'..'z' | 'A'..'Z' → "Consonant"
    | '0'..'9' → "Digit"
    |   _ → "Other" ;;
val char_discriminate : char -> string = <fun>
```

パターンのグループの順番に意味があることに注意してください。実際、二番目のパターンの集まりは一番目を含みますが、一番目のチェックが済むまで二番目は調べられません。

リストに対するパターンマッチング

すでに見たように、リストは

- 空であるか（そのようなリストは `[]` という形をしています)、
- 第一要素（先頭）と部分リスト（末尾）からなります。このとき、リストは `h::t` という形をしています。

このような二つのリストの書き方は、パターンとして用いることができます。これにより、リストに対するパターンマッチングを行うことができます。

```
# let rec size x = match x with
      [] → 0
    | _::tail_x → 1 + (size tail_x) ;;
val size : 'a list -> int = <fun>
```

```
# size [];;
- : int = 0
# size [7;9;2;6];;
- : int = 4
```

したがって、たとえばリストに対する繰り返し等の以前に記述した例（31 ページ参照）を、パターンマッチングを用いて書き直すことができます。

```
# let rec fold_left f a = function
      [] → a
    | head :: tail → fold_left f (f a head) tail ;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# fold_left (+) 0 [8;4;10];;
- : int = 22
```

パターンマッチングによる値の宣言

値の宣言は、実はパターンマッチングを用いています。宣言 `let x = 18` は値 18 をパターン x にマッチさせせます。宣言の左辺には、どのようなパターンでも書くことができます。パターンの中の変数は、マッチする値に束縛されます。

```
# let (a,b,c) = (1, true, 'A');;
val a : int = 1
val b : bool = true
val c : char = 'A'
# let (d,c) = 8, 3 in d + c;;
- : int = 11
```

このようなパターン変数のスコープは、通常のローカル宣言の静的スコープです。ここでは、c が'A' に束縛されたままとなります。

```
# a + (int_of_char c);;
- : int = 66
```

他のパターンマッチングと同じように、値宣言もパターンマッチングとして完全ではないことがあります。

```
# let [x;y;z] = [1;2;3];;
Characters 5-12:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
  let [x;y;z] = [1;2;3];;
      ^^^^^^^
val x : int = 1
val y : int = 2
val z : int = 3
# let [x;y;z] = [1;2;3;4];;
Characters 4-11:
Warning: this pattern-matching is not exhaustive.
```

```
Here is an example of a value that is not matched:
[]
  let [x;y;z] = [1;2;3;4];;
      ^^^^^^^
Exception: Match_failure ("", 4, 11).
```

コンストラクタやワイルドカード、パターンの組み合わせを含めて、どのようなパターンも使えます。

```
# let head :: 2 :: _ =  [1; 2; 3] ;;
Characters 5-19:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
  let head :: 2 :: _ =  [1; 2; 3] ;;
      ^^^^^^^^^^^^^^
val head : int = 1
# let _ = 3. +. 0.14 in "PI" ;;
- : string = "PI"
```

この最後の例は、関数的世界の範囲ではほとんど意味がありません。計算された値 3.14 に名前が与えられず、失われるからです。

型宣言

Objective Caml の語句表現における、もう一つの可能な構成要素としては、型宣言があります。型宣言は、プログラムで使用する独自のデータ構造に対応する新しい型の定義を可能にします。型には大きくわけて二つの種類があります。組やレコードのための**積型**と、ユニオンのための**和型**です。

型宣言には、キーワード **type** を用います。

構文：

```
type name = typedef ;;
```

変数宣言とは対照的に、型宣言はデフォルトで再帰的です。したがって、型宣言を結合することにより相互再帰的な型の定義が可能です。

構文：

```
type  name1  =  typedef1
and   name2  =  typedef2
        ⋮
and   namen  =  typedefn ;;
```

型宣言は、型変数によってパラメータ化できます。型変数名は常にアポストロフィ（文字 **'**）で始まります。

構文：

```
type 'a name = typedef ;;
```

複数の型引数があるときは、型の名前の前で組として宣言します。

構文 : `type ('a1 ... 'an) name = typedef ;;`

宣言の左辺で定義された型引数しか、右辺にあらわれることはできません。

注意

Objective Caml の型表示器は、与えられた型の名前をつけかえます。一番目は 'a、二番目は 'b、…というように名づけられます。

すでに存在する型から新しい型を定義することは、いつでも可能です。

構文 : `type name = type expression`

これは一般的すぎる思われる型を制限するのに便利です。

```
# type 'param paired_with_integer = int * 'param ;;
type 'a paired_with_integer = int * 'a
# type specific_pair = float paired_with_integer ;;
type specific_pair = float paired_with_integer
```

型制約がなくても、推論によりもっとも一般的な型が生成されます。

```
# let x = (3, 3.14) ;;
val x : int * float = (3, 3.14)
```

しかし、型制約を使えば、望みの名前が表示されます。

```
# let (x:specific_pair) = (3, 3.14) ;;
val x : specific_pair = (3, 3.14)
```

レコード

レコードは、Pascal の *record* や C の *struct* のように、各々のフィールドに名前のついた組です。レコードは常に、新しい型の宣言に対応します。レコード型は、自身の名前と、それぞれのフィールドの名前および型によって定義されます。

構文 : `type name = { name1 : t1; ...; namen : tn } ;;`

複素数を表現する型は

```
# type complex = { re:float; im:float } ;;
type complex = { re : float; im : float; }
```

のように定義できます。

レコード型の値を作るには、それぞれのフィールドに（任意の順番で）値を与えます。

構文 : `{ name_i1 = expr_i1 ; ...; name_in = expr_in } ;;`

たとえば、実数部分 2. と虚数部分 3. を持つ複素数を作ります。

```
# let c = {re=2.;im=3.} ;;
```

```
val c : complex = {re = 2; im = 3}
# c = {im=3.;re=2.} ;;
- : bool = true
```

一部のフィールドがない場合は、次のようなエラーが出ます。

```
# let d = { im=4. } ;;
Characters 9-18:
  let d = { im=4. } ;;
          ^^^^^^^^^
Some record field labels are undefined: re
```

フィールドをアクセスするには二つの方法があります。ドット記法と、一部のフィールドに対するパターンマッチングです。

ドット記法の文法は普通です。

構文： *expr.name*

式 *expr* はフィールド *name* を含むレコード型でなければいけません。

レコードに対するパターンマッチングを用いると、複数のフィールドに束縛された値を取り出すことができます。

構文： { $name_i$ **=** p_i **;** ... **;** $name_j$ **=** p_j }

ここでのパターンは**=**記号の右側にきます（p_i, ..., p_j）。このようなパターンにおいては、レコードのすべてのフィールドが出現する必要はありません。

関数add_complexはドット記法を用いてフィールドにアクセスするのに対し、関数mult_complexはパターンマッチングを用いてフィールドにアクセスします。

```
# let add_complex c1 c2 =  {re=c1.re+.c2.re; im=c1.im+.c2.im};;
val add_complex : complex -> complex -> complex = <fun>
# add_complex c c ;;
- : complex = {re = 4; im = 6}
# let mult_complex c1 c2 = match (c1,c2) with
    ({re=x1;im=y1},{re=x2;im=y2}) → {re=x1*.x2-.y1*.y2;im=x1*.y2+.x2*.y1} ;;
val mult_complex : complex -> complex -> complex = <fun>
# mult_complex c c ;;
- : complex = {re = -5; im = 12}
```

組に対するレコードの利点は、少なくとも二つあります。

- フィールド名が説明的かつ識別的な情報を与えてくれる。特に、パターンマッチングが簡単になります。
- 名前により、どのようなレコードでも任意のフィールドに同一の方法でアクセスが可能となる。もはやフィールドの順番は関係がなく、名前のみが重要です。

次の例は、組に比べてレコードのフィールドにアクセスするのが楽であることを示しています。

```
# let a = (1,2,3) ;;
val a : int * int * int = (1, 2, 3)
# let f tr = match tr with  x,_,_ → x ;;
val f : 'a * 'b * 'c -> 'a = <fun>
# f a ;;
- : int = 1
# type triplet = {x1:int; x2:int; x3:int} ;;
type triplet = { x1 : int; x2 : int; x3 : int; }
# let b = {x1=1; x2=2; x3=3} ;;
val b : triplet = {x1 = 1; x2 = 2; x3 = 3}
# let g tr = tr.x1 ;;
val g : triplet -> int = <fun>
# g b ;;
- : int = 1
```

パターンマッチングでは、マッチされるレコードのすべてのフィールドを指示する必要はありません。よって推論される型は最後のフィールドの型となります[11]。

```
# let h tr = match tr with  {x1=x} → x;;
val h : triplet -> int = <fun>
# h b;;
- : int = 1
```

あるレコードと、いくつかのフィールドを除いて等しいレコードを作る構文があります。これは、多くのフィールドを含むレコードに対し、しばしば役に立ちます。

構文 : { *name* **with** $name_i$**=** $expr_i$ **;** ... **;** $name_j$**=**$expr_j$}

```
# let c = {b with x1=0} ;;
val c : triplet = {x1 = 0; x2 = 2; x3 = 3}
```

b の値がコピーされて、フィールド x1 だけが違う値を持つ新しいレコードが作られます。

警告 この仕様は言語拡張の一部であり、将来のバージョンでは変更されるかもしれません。

和型

デカルト積に相当する組やレコードとは対照的に、和型の宣言は集合のユニオンに相当します。異なる型（たとえば整数や文字列）が一つの型としてまとめられます。そのような和において、異なる要素は**コンストラクタ**によって区別します。コンストラクタは、一方ではその名が示す通り、その型の値を**構成**することを可能にし、他方ではパターン

11. 訳注：“The inferred type is then that of the last field.” 英文の意味が不明。仏文の確認が必要。

マッチングにより、値の構成要素に**アクセス**することを可能にします。コンストラクタを引数に適用すると、値はこのような新しい型に属することになります。

和型を宣言するには、コンストラクタと、その引数の型を与えます。

構文：

```
type  name = ...
      | Name_i ...
      | Name_j of t_j ...
      | Name_k of t_k * ...* t_l ...;;
```

コンストラクタの名前は特別な識別子です。

警告 コンストラクタの名前は、常に大文字で始まります。

定数コンストラクタ

引数を期待しないコンストラクタは、**定数コンストラクタ**と呼ばれます。定数コンストラクタは、定数のように、言語の値として以降で直接使用することができます。

```
# type coin = Heads | Tails;;
type coin = Heads | Tails
# Tails;;
- : coin = Tails
```

bool 型はこのような方法で定義することが可能です。

引数つきコンストラクタ

コンストラクタは引数をとることができます。キーワード **of** は、コンストラクタの引数の型を示します。これにより、異なる型の対象を一つの型にまとめ、それぞれを特定のコンストラクタにより導入することが可能です。

データ型を定義する古典的な例として、一つのゲーム（ここではタロット[12]）におけるトランプのカードを表現してみます。型 *suit* と *card* は次のように定義されます。

```
# type suit = Spades | Hearts | Diamonds | Clubs ;;
# type card =
      King of suit
    | Queen of suit
    | Knight of suit
    | Knave of suit
    | Minor_card of suit * int
    | Trump of int
    | Joker ;;
```

12. 原訳注：フランスのタロットのルールは、たとえば
リンク: http://www.pagat.com/tarot/frtarot.html
にあります。

型 *card* の値は、コンストラクタを適切な型の値に適用することによって生成されます。

```
# King Spades ;;
- : card = King Spades
# Minor_card(Hearts, 10) ;;
- : card = Minor_card (Hearts, 10)
# Trump 21 ;;
- : card = Trump 21
```

また、たとえば引数として渡されたマークの、すべてのカードのリストを構成する関数 all_cards は、このようになります。

```
# let rec interval a b =  if a = b then [b] else a::(interval (a+1) b) ;;
val interval : int -> int -> int list = <fun>
# let all_cards s =
      let face_cards = [ Knave s; Knight s; Queen s; King s ]
      and other_cards  = List.map (function n → Minor_card(s,n)) (interval 1 10)
      in face_cards @ other_cards ;;
val all_cards : suit -> card list = <fun>
# all_cards Hearts ;;
- : card list =
[Knave Hearts; Knight Hearts; Queen Hearts; King Hearts;
 Minor_card (Hearts, 1); Minor_card (Hearts, 2); Minor_card (Hearts, 3);
 Minor_card (Hearts, ...); ...]
```

和型の値を扱うには、パターンマッチングを使います。次の例では、型 *suit* および型 *card* の値を文字列（型 *string*）に変換する関数を構築しています。

```
# let string_of_suit = function
        Spades   → "spades"
      | Diamonds → "diamonds"
      | Hearts   → "hearts"
      | Clubs    → "clubs"  ;;
val string_of_suit : suit -> string = <fun>
# let string_of_card = function
        King c            → "king of " ^ (string_of_suit c)
      | Queen c           → "queen of " ^ (string_of_suit c)
      | Knave c           → "knave of " ^ (string_of_suit c)
      | Knight c          → "knight of " ^ (string_of_suit c)
      | Minor_card (c, n) → (string_of_int n) ^ " of "^(string_of_suit c)
      | Trump n           → (string_of_int n) ^ " of trumps"
      | Joker             → "joker" ;;
val string_of_card : card -> string = <fun>
```

パターンを一列に並べることで、これらの関数が読みやすくなります。

コンストラクタ Minor_card は**二つの**引数を持つコンストラクタとして扱われます。そのような値に対してパターンマッチングを行うには、その二つの構成要素を指定する必要があります。

```
# let is_minor_card c = match c with
```

```
     Minor_card v →  true
   | _ → false;;
Characters 41-53:
      Minor_card v ->  true
      ^^^^^^^^^^^^
The constructor Minor_card expects 2 argument(s),
but is here applied to 1 argument(s)
```

コンストラクタの一つ一つの構成要素を指定しないですませるためには、対応する組型を括弧で囲んで、一つの引数をとるようにコンストラクタを宣言します。次の二つのコンストラクタは、パターンマッチングのしかたが異なります。

```
# type t =
        C of int * bool
      | D of (int * bool) ;;
# let access v = match v with
        C (i, b) → i,b
      | D x  → x;;
val access : t -> int * bool = <fun>
```

再帰型

再帰的な型の定義は、通常のデータ構造（リスト、ヒープ、木、グラフなど）を記述するために、どのような算法言語でも必要不可欠です。そのため、値の宣言（**let**）とは対照的に、Objective Caml における型の宣言はデフォルトで再帰的です。

Objective Caml の元から定義されているリストの型は、一つだけ引数をとります。リスト構造に二つの異なる型、たとえば整数（*int*）と文字（*char*）に属する値を格納したいと思うかもしれません。その場合は、以下のように定義します。

```
# type int_or_char_list =
        Nil
      | Int_cons of int * int_or_char_list
      | Char_cons of char * int_or_char_list ;;

# let l1 =  Char_cons ( '=', Int_cons(5, Nil) )  in
      Int_cons ( 2, Char_cons ( '+', Int_cons(3, l1) ) )  ;;
- : int_or_char_list =
Int_cons (2, Char_cons ('+', Int_cons (3, Char_cons ('=', Int_cons (...)))))
```

パラメータ化された型

同様に、ユーザはパラメータをもつ型を定義することができます。これにより、二つの異なる型の値を含むリストの例を一般化することができます。

```
# type ('a, 'b) list2 =
```

```
        Nil
      | Acons of 'a * ('a, 'b) list2
      | Bcons of 'b * ('a, 'b) list2 ;;

# Acons(2, Bcons('+', Acons(3, Bcons('=', Acons(5, Nil))))) ;;
- : (int, char) list2 =
Acons (2, Bcons ('+', Acons (3, Bcons ('=', Acons (...)))))
```

当然、引数 *'a* と *'b* を同じ型で具体化することもできます。

```
# Acons(1, Bcons(2, Acons(3, Bcons(4, Nil)))) ;;
- : (int, int) list2 = Acons (1, Bcons (2, Acons (3, Bcons (4, Nil))))
```

このように型 *list2* を使えば、先の例のように、偶数と奇数を区別することができます。その方法で偶数の部分リストを抽出し、通常のリストを構築します。

```
# let rec extract_odd = function
       Nil → []
     | Acons(_, x) → extract_odd x
     | Bcons(n, x) → n :: (extract_odd x) ;;
val extract_odd : ('a, 'b) list2 -> 'b list = <fun>
```

この関数の定義は、リスト構造に格納された値の性質について、何ら手がかりを与えません。型がパラメータ化されているのは、そのためです。

宣言のスコープ

コンストラクタの名前はグローバル宣言と同じスコープ規則にしたがいます。すなわち、再定義は以前の定義を隠します。それでも、その隠された型の値はまだ存在します。Objective Caml の対話的トップレベルは出力のとき、そのような二つの型を区別しません。そのために、ある種の不明確なエラーメッセージが出ます。

下の一つ目の例では、型 *int_or_char* の定数コンストラクタ Nil が、型 *('a, 'b) list2* のコンストラクタ宣言によって隠されています。

```
# Int_cons(0, Nil) ;;
Characters 13-16:
  Int_cons(0, Nil) ;;
               ^^^
This expression has type ('a, 'b) list2 but is here used with type
  int_or_char_list
```

次の例では、少なくともはじめて見たときにはかなり不可解なエラーメッセージが発生します。下の小さなプログラムを考えましょう。

```
# type t1 = Empty | Full;;
type t1 = Empty | Full
# let empty_t1 x = match x with Empty → true | Full → false ;;
val empty_t1 : t1 -> bool = <fun>
# empty_t1 Empty;;
```

```
- : bool = true
```

そして、型 *t1* を再宣言します。

```
# type t1 = {u : int; v : int} ;;
type t1 = { u : int; v : int; }
# let y = { u=2; v=3 } ;;
val y : t1 = {u = 2; v = 3}
```

ここで、関数 empty_t1 を新しい型 *t1* の値に適用すると、次のエラーメッセージを得ます。

```
# empty_t1 y;;
Characters 10-11:
  empty_t1 y;;
           ^
This expression has type t1 but is here used with type t1
```

一つ目の *t1* は一番目に定義した型を表し、二つ目の *t1* は二番目に定義した型に対応します。

関数型

コンストラクタの引数の型は任意です。特に、関数型を含んでもまったく構いません。次の型は、最後以外の要素がすべて関数値であるようなリストを構成します。

```
# type 'a listf =
      Val of 'a
    | Fun of ('a -> 'a) * 'a listf ;;
type 'a listf = Val of 'a | Fun of ('a -> 'a) * 'a listf
```

関数値は言語によって扱うことのできる値なので、型 *listf* の値

```
# let eight_div = (/) 8 ;;
val eight_div : int -> int = <fun>
# let gl = Fun (succ, (Fun (eight_div, Val 4))) ;;
val gl : int listf = Fun (<fun>, Fun (<fun>, Val 4))
```

や、そのような値にパターンマッチする関数

```
# let rec compute = function
      Val v -> v
    | Fun(f, x) -> f (compute x) ;;
val compute : 'a listf -> 'a = <fun>
# compute gl;;
- : int = 3
```

を構成することができます。

例：木の表現

木構造はプログラミングにおいて頻繁に出現します。再帰型を利用すれば、そのような構造を簡単に定義したり操作することができます。本節では、木構造の例を二つ挙げます。

二分木 頂点が一つの型の値によってラベルづけされている二分木構造を、以下の宣言により定義します。

```
# type 'a bin_tree =
      Empty
    | Node of 'a bin_tree * 'a * 'a bin_tree ;;
```

この構造を用いて、二分探索木を使った小さな整列プログラムを定義します。二分探索木は、左の枝にあるすべての値が根の値より小さく、右の枝にあるすべての値は根の値より大きいという性質があります。図 2.5 に、整数上のそのような構造の例を挙げます。空の頂点（コンストラクタ Empty）は小さな四角により表現され、その他の頂点（コンストラクタ Node）は、格納している値が記された円で表現されています。

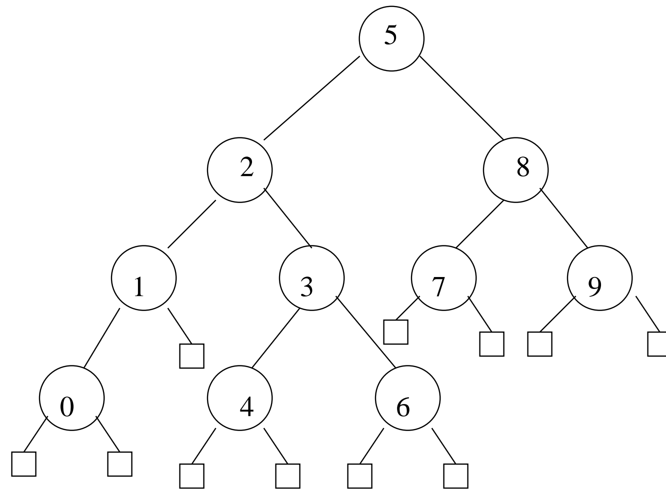


図 2.5: 二分探索木

以下の関数により、二分探索木を順番に探索して、整列リストを抽出します。

```
# let rec list_of_tree = function
     Empty → []
   | Node(lb, r, rb) → (list_of_tree lb) @ (r :: (list_of_tree rb)) ;;
val list_of_tree : 'a bin_tree -> 'a list = <fun>
```

リストから二分探査木を得るために、挿入関数を定義します。

```
# let rec insert x = function
```

```
      Empty  →  Node(Empty, x, Empty)
    | Node(lb, r, rb)  →  if x < r then Node(insert x lb, r, rb)
                          else Node(lb, r, insert x rb) ;;
val insert : 'a -> 'a bin_tree -> 'a bin_tree = <fun>
```

関数 insert を繰り返すことにより、リストを木に変換する関数が得られます。

```
# let rec tree_of_list = function
        []    →  Empty
     | h :: t  →  insert h (tree_of_list t) ;;
val tree_of_list : 'a list -> 'a bin_tree = <fun>
```

すると、整列関数は単純に関数 tree_of_list と list_of_tree の合成になります。

```
# let sort x = list_of_tree (tree_of_list x) ;;
val sort : 'a list -> 'a list = <fun>
# sort [5; 8; 2; 7; 1; 0; 3; 6; 9; 4] ;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

一般平面木　この部分では、List モジュールであらかじめ定義されている以下の関数を使用します（217 ページ参照）。

- List.map：リストのすべての要素に関数を適用し、その結果のリストを返します。
- List.fold_left：32 ページで定義した関数 fold_left と等価です。
- List.exists：リストのすべての要素に、論理値関数を適用します。もしそれらの適用が一つでも true を与えたら、結果は true になり、そうでなければ、この関数は false を返します。

一般平面木とは、枝の数があらかじめ決まっていない木のことです。個々の頂点に枝のリストが関連づけられており、その長さは様々です。

```
# type 'a tree = Empty
               | Node of 'a * 'a tree list ;;
```

空の木は値 Empty により表現されます。葉とは、Node(x,[]) あるいは縮退した Node(x, [Empty;Empty; ..]) という形をした、枝のない頂点のことです。すると、これらの木を操作する関数、たとえばある要素が木に属するかどうか判定したり、木の高さを計算する関数を書くことは、比較的容易です。

要素 e が木に属するかどうか検査するには、次のアルゴリズムを用います：もし木が空ならば、e はその木に属しません。そうでなければ、e が根のラベルに等しいか、あるいは枝の一つに属するときのみ、e は木に属します。

```
# let rec belongs e = function
      Empty → false
    | Node(v, bs) → (e=v) or (List.exists (belongs e) bs) ;;
val belongs : 'a -> 'a tree -> bool = <fun>
```

木の高さを計算するには、次の定義を用います：空の木は高さが 0 です。さもなくば、木の高さは、もっとも高い部分木の高さプラス 1 に等しくなります。

```
# let rec height =
   let max_list l = List.fold_left max 0 l in
    function
      Empty → 0
    | Node (_, bs) → 1 + (max_list (List.map height bs)) ;;
val height : 'a tree -> int = <fun>
```

関数以外の再帰的値

関数値以外の再帰的宣言により、循環データ構造を定義することができます。

以下の宣言は、一要素の循環リストを構築します。

```
# let rec l = 1 :: l ;;
val l : int list =
  [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; ...]
```

そのようなリストに再帰関数を適用すると、メモリがオーバーフローするまでループする危険があります。

```
# size l ;;
Stack overflow during evaluation (looping recursion?).
```

構造等値が依然として使用可能なのは、最初に物理等値が確認される場合のみです。

```
# l=l ;;
- : bool = true
```

要するに、もし新しいリストを定義したら、たとえ等しくても、構造等値検査を使ってはいけません。さもないと、プログラムが永久にループします。ですから、次の例を評価しようと試みることはお勧めできません。

```
let rec l2 = 1::l2 in l=l2 ;;
```

一方、物理等値は依然としていつでも判定可能です。

```
# let rec l2 = 1 :: l2 in l==l2 ;;
- : bool = false
```

述語==は即値の等しさか、構造オブジェクトの共有（値のアドレスの等しさ）を検査します。これを用いて、リストを探索するとき、すでに調べた部分リストを再探索しないように検査してみましょう。まずはじめに、物理等値によってリストの要素の存在を検証する関数 memq を定義します。これは構造等値を検査する関数 mem とは対照的な関数です。これら二つの関数は、モジュール List に属します。

```
# let rec memq a l = match l with
```

```
      []  → false
    | b :: l → (a==b) or (memq a l) ;;
val memq : 'a -> 'a list -> bool = <fun>
```

すでに調べたリストのリストを覚えておき、一つのリストが二度あらわれたら止まるように、サイズを計算する関数を再定義します。

```
# let special_size l =
    let rec size_aux previous l = match l with
        []  → 0
      |  _::l1  → if memq  l previous then 0
                         else 1 + (size_aux (l :: previous) l1)
    in size_aux [] l ;;
val special_size : 'a list -> int = <fun>
# special_size [1;2;3;4] ;;
- : int = 4
# special_size l ;;
- : int = 1
# let rec l1 = 1 :: 2 :: l2  and  l2 = 1 :: 2 :: l1 in special_size l1 ;;
- : int = 4
```

型付け、定義域、例外

関数に対して推論される型は、その関数の定義域の上位集合に対応します。関数が型 *int* の引数をとるからといって、引数として渡されるすべての整数について、値を計算する方法を知っているとは限りません。この問題は一般に、Objective Caml の例外機構により処理されます。例外が発生すると計算への割り込みが起こりますが、この割り込みはプログラムで傍受して処理することができます。そうするためには、プログラムの実行において、例外が発生する式を計算する前に、例外ハンドラを登録しておかなくてはなりません。

部分関数と例外

関数の定義域は、その関数が計算を行う値の集合に相当します。多くの数学的関数は部分的です。たとえば割り算や自然対数が挙げられます。この問題は、もっと複雑なデータ構造を操作する関数でも発生します。実際、空リストの第一要素を計算した結果は何になるというのでしょうか。同様に、`factorial` 関数を負の整数について評価したら、無限に再帰してしまいます。

プログラムの実行中には、たとえばゼロによる割り算の試みのように、例外的な状況が発生することがあります。数をゼロで割ろうとしたら、良くともプログラムが止まってしまうでしょうし、悪ければマシンの状態が不整合になるでしょう。プログラミング言語の**安全性**は、このような特別な場合に、そういう状況が発生しないという保証に由来します。例外は、そのようなケースに対応する方法の一つです。

1 を 0 で割ると、特定の例外が発生します。

```
# 1/0;;
Exception: Division_by_zero.
```

Exception: Division_by_zero というメッセージは、Division_by_zero 例外が発生したことと共に、その例外が処理されなかったことを示しています。この例外は、言語の核で宣言されているものの一つです。

パターンマッチングが完全でない、すなわち与えられた式のすべての場合にマッチせず、関数の型が定義域に一致しないことがよくあります。このような誤りを防ぐために、Objective Caml はそのような場合にメッセージを表示します。

```
# let head l = match l with h :: t → h ;;
Characters 14-36:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
  let head l = match l with h::t -> h ;;
               ^^^^^^^^^^^^^^^^^^^^^^
val head : 'a list -> 'a = <fun>
```

もしそれでもプログラマが不完全な定義を直さず、誤って部分関数を呼んだ場合は、Objective Caml は例外機構を用います。

```
# head [] ;;
Exception: Match_failure ("", 14, 36).
```

最後に、元から定義されているもう一つの例外 Failure は前に出てきました。この例外は、型 *string* の引数をとります。関数 failwith を使用してこの例外を発生させることが可能です。これを使用して、独自に head を定義することができます。

```
# let head = function
      [] → failwith "Empty list"
    | h :: t → h;;
val head : 'a list -> 'a = <fun>
# head [] ;;
Exception: Failure "Empty list".
```

例外の定義

Objective Caml では、例外は元から定義されている型 *exn* に属します。この型は**拡張可能**な和型であるという点で非常に特別です。この型の値の集合は、新しいコンストラクタを宣言することにより、拡張することができます[13]。この機能を利用して、ユーザは

13. 原訳注：Objective Caml 3.00 の新しい「多相バリアント」の機能により、今では他にも一種の和型を拡張することが可能です。

型 *exn* に新しいコンストラクタを追加することにより、独自の例外を定義することができます。

例外宣言の文法は次の通りです：

構文： **exception** *Name* **;;**

または

構文： **exception** *Name* **of** *t* **;;**

例外宣言の例をいくつか示します：

```
# exception MY_EXN;;
exception MY_EXN
# MY_EXN;;
- : exn = MY_EXN
# exception Depth of int;;
exception Depth of int
# Depth 4;;
- : exn = Depth 4
```

このように、例外は言語の値として一人前に扱われます。

警告 例外の名前はコンストラクタです。したがって必ず大文字で始まらなければいけません。

```
# exception lowercase ;;
Characters 11-20:
  exception lowercase ;;
            ^^^^^^^^^
Syntax error
```

警告 例外は単相的です。例外の引数の宣言に型変数はありません。

```
# exception Value of 'a ;;
Characters 20-22:
  exception Value of 'a ;;
                     ^^
Unbound type parameter 'a
```

もし多相的な例外があったら、任意の返り値型を持つ関数を定義することができてしまいます。これについては、57 ページでより詳しく述べます。

例外の発生

関数 raise は言語のプリミティブ関数です。これは例外を引数にとり、完全に多相的な返り値型を持ちます。

```
# raise ;;
```

```
- : exn -> 'a = <fun>
# raise MY_EXN;;
Exception: MY_EXN.
# 1+(raise MY_EXN);;
Exception: MY_EXN.
# raise (Depth 4);;
Exception: Depth 4.
```

関数 raise は Objective Caml で記述することができないので、あらかじめ定義されていなければなりません。

例外処理

そもそも例外を発生させる目的は、例外を処理して、発生した例外の値により計算の系列をコントロールすることにあります。したがって、式を評価する順序は、どの例外が発生するかを決定する場合に重要となります。我々は純粋に関数的な文脈から抜け出て、次章で議論するように（84 ページ参照）、引数を評価する順序が計算の結果を変えうる世界へ入っていきます。

次の構文は式の値を計算しますが、計算中に発生した例外の処理を可能とします。

構文 :

```
try expr with
| p1 -> expr1
⋮
| pn -> exprn
```

もし *expr* の評価が例外を発生しなければ、結果は *expr* を評価した結果になります。さもなくば、発生した例外の値がパターンマッチされ、最初にマッチしたパターンに対応する式の値が返されます。もしどのパターンも発生した例外の値に対応しなければ、その例外はプログラムの実行中に入った一つ外側の **try-with** に伝搬します。したがって、例外のパターンマッチングは常に完全であると考えられます。暗黙に、最後のパターンは | e -> raise e となります。もしマッチする例外ハンドラがプログラムの中で見つからなければ、システム自身が例外を傍受する役目をし、エラーメッセージを表示してプログラムを終了します。

例外（すなわち型 *exn* の値）を**計算**することと、例外を**発生**して計算を中断することを混同しないでください。例外は他の値と同じように、関数の結果として返すことができます。

```
# let return x = Failure x ;;
val return : string -> exn = <fun>
# return "test" ;;
- : exn = Failure "test"
# let my_raise x = raise (Failure x) ;;
val my_raise : string -> 'a = <fun>
# my_raise "test" ;;
```

```
Exception: Failure "test".
```
my_raise を適用しても値を返さないのに対し、return を適用すると型 *exn* の値が返ることに気をつけます。

例外を利用した計算

例外的な値を扱うという使い方の他に、例外は特定のプログラミングスタイルをサポートし、最適化の源となることもできます。次の例は、整数のリストのすべての要素の積を求めます。例外を利用して、値 0 に遭遇したらリストの探索を中断し、0 を返します。

```
# exception Found_zero ;;
exception Found_zero
# let rec mult_rec l = match l with
        [] → 1
     | 0 :: _ → raise Found_zero
     | n :: x → n * (mult_rec x) ;;
val mult_rec : int list -> int = <fun>
# let mult_list l =
      try mult_rec l with Found_zero → 0 ;;
val mult_list : int list -> int = <fun>
# mult_list [1;2;3;0;5;6] ;;
- : int = 0
```
待機しているすべての計算、すなわち一つ一つの再帰呼び出しに続く n による掛け算は、放棄されます。**raise** に遭遇すると、**with** で指定されたパターンマッチングから計算が再開します。

多相性と関数の返り値

Objective Caml のパラメータ的多相性を利用すれば、返り値型がまったく定まっていない関数を定義することができます。たとえば

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
```
です。

しかし、この返り値型は引数の型に依存します。したがって、関数 id が引数に適用されるとき、型推論機構には型変数 *'a* を具体化する方法がわかります。よって、各々の特定の使用については、id の型が定まります。

もし仮にそうでなかったとしたら、実行安全性の保証を得るために強い静的な型付けを用いる意味はもはやありません。実際、もし仮に *'a -> 'b* のようなまったく不定の型があったら、どんな型の変換もできてしまい、必然的に実行時エラーとなります。異なる型の値の物理的表現は、同じではないからです。

見かけ上の矛盾

しかし、Objective Caml 言語においては、引数の型に現れない型変数を返り値型が含むような関数を定義することが可能です。そのような例についていくつか考え、なぜそういう可能性が、強い静的な型付けに矛盾しないのか見ていきます。

最初の例は次の通りです：

```
# let f x = [] ;;
val f : 'a -> 'b list = <fun>
```

この関数を使えば、何からでも多相的な値を作ることができます。

```
# f () ;;
- : '_a list = []
# f "anything at all" ;;
- : '_a list = []
```

しかし、得られる値がまったく不定なわけではなく、リストになっています。よって、この値は単にどこでも使えるわけではありません。

また、恐るべき型 'a -> 'b を持つ三つの例を、次に示します。

```
# let rec f1 x = f1 x ;;
val f1 : 'a -> 'b = <fun>
# let f2 x = failwith "anything at all" ;;
val f2 : 'a -> 'b = <fun>
# let f3 x = List.hd [] ;;
val f3 : 'a -> 'b = <fun>
```

これらの関数は、実行安全性に関する限り、実際には危険ではありません。というのは、これらを用いて値を作ることはできないからです。最初の関数は永久にループしますし、後の二つは例外を発生して計算を中断します。

同様に、新しい例外コンストラクタが、変数を含む型の引数を持つことを禁じているのは、型 'a -> 'b の関数を定義できないようにするためです。

実際、もし仮に型 'a -> exn の多相的な例外 Poly_exn を宣言することができたとしたら、次のような関数を書くことができてしまいます。

```
let f = function
   0 → raise (Poly_exn false)
 | n → n+1 ;;
```

すると、関数 f は型 int -> int、例外 Poly_exn は型 'a -> exn ですから、

```
let g n = try f n with Poly_exn x → x+1 ;;
```

と定義することができます。この関数は正しく型付けされてしまい （Poly_exn の引数は任意なので)、よって (g 0) を評価したら、整数を論理値に足そうと試みることになってしまいます！

電卓

Objective Caml でプログラムを作る方法を理解するためには、何かを開発してみる必要があります。ここでは例として電卓を選ぶことにします。ただし、整数の四則演算だけ実行できる、もっとも簡単なモデルとします。

はじめに、電卓のキーを表現する型 *key* を定義します。この電卓には 15 個のキーがあります。それぞれの演算について 1 個、それぞれの数字について 1 個、そして=キーです。

```
# type key = Plus | Minus | Times | Div | Equals | Digit of int ;;
```

数字のキーは整数引数をとる単一のコンストラクタ *Digit* にまとめられていることに注意します。本当は、型 *key* には実際のキーを表現しない値もあります。たとえば、(*Digit* 32) は型 *key* の値としてはありえますが、電卓のいかなるキーも表現しません。

そこで、引数が電卓のキーに対応するかどうか確かめる関数 valid を書きます。この関数の型は *key -> bool* です。すなわち、型 *key* の値を引数としてとり、型 *bool* の値を返します。

最初のステップとして、整数が 0 から 9 の間に含まれるか確かめる関数を定義します。この関数を is_digit という名前で宣言します。

```
# let is_digit = function x → (x>=0) && (x<=9) ;;
val is_digit : int -> bool = <fun>
```

そして、型 *key* の引数についてのパターンマッチングにより、関数 valid を定義します。

```
#   let valid ky = match ky with
      Digit n → is_digit n
    | _ → true ;;
val valid : key -> bool = <fun>
```

最初のパターンは、valid の引数が *Digit* コンストラクタにより作られた値の場合に適用されます。この場合は、*Digit* の引数が関数 is_digit により検査されます。二番目のパターンはそれ以外のすべての型 *key* の値に適用されます。型付けのおかげで、マッチされる値は必ず型 *key* をもつことを思い出してください。

電卓の機構を書き始める前に、装置のキーが有効になったときの反応を形式的な観点から記述できるような**モデル**を指定します。電卓は、最後にした計算、最後に有効となったキー、最後に有効となった演算、画面に表示された数、の 4 つのレジスタを持つと考えます。これら 4 つのレジスタの集合を、電卓の状態と呼ぶことにします。この状態は、キーパッドのキーが押されるたびに変化します。この変化を遷移と呼びます。このような機構を支配するのはオートマトンの理論です。我々のプログラムでは、以下のレコード型により状態を表現します。

```
# type state = {
    lcd : int; (* 最後に行われた計算       *)
    lka : key; (* 最後に有効となったキー *)
    loa : key; (* 最後に有効となった演算 *)
    vpr : int  (* 表示されている値        *)
```

```
} ;;
```

図 2.6 に遷移の列の例を挙げます。

	state	key
	$(0,=,=,0)$	3
$\longrightarrow$	$(0,3,=,3)$	$+$
$\longrightarrow$	$(3,+,+,3)$	2
$\longrightarrow$	$(3,2,+,2)$	1
$\longrightarrow$	$(3,1,+,21)$	$\times$
$\longrightarrow$	$(24,*,*,24)$	2
$\longrightarrow$	$(24,2,*,2)$	$=$
$\longrightarrow$	$(48,=,=,48)$	

図 2.6: Transitions for $3+21*2=$.

以下においては、演算子を含んだ型 key の値と二つの整数をとり、キーに対応する演算を整数に適用した結果を返す関数 evaluate が必要になります。この関数は、型 *key* の最後の引数についてのパターンマッチングにより定義されます。

```
# let evaluate x y ky = match ky with
      Plus    → x + y
    | Minus   → x - y
    | Times   → x * y
    | Div     → x / y
    | Equals  → y
    | Digit _ → failwith "evaluate : no op";;
val evaluate : int -> int -> key -> int = <fun>
```

そして、すべての場合を列挙することにより、遷移関数を定義します。現在の状態が 4 つ組 $(a,b,\oplus,d)$ であると仮定します。

- 数字 x のキーが押されたら、二つの場合が考えられます。
  - 最後に押されたキーも数字だった。したがって、これは電卓のユーザが入力している最中の値です。よって、表示されている値に数字 x を付け加えなければなりません。つまり、値を $d\times 10+x$ で置き換えます。新しい状態は

    $$(a, (\texttt{Digit x}), \oplus, d\times 10+x)$$

    となります。
  - 最後に押されたキーが数字ではなかった。したがって、これは新たに入力される数字の始まりです。新しい状態は

    $$(a, (\texttt{Digit x}), \oplus, x)$$

    となります。

- 演算子 $\otimes$ のキーが押されたら、それにより演算の第二オペランドの入力が完了するので、電卓は演算を実行しなければいけません。最後の演算（ここでは $\oplus$）が保存されているのは、このためです。新しい状態は

$$(a \oplus d, \otimes, \otimes, a \oplus d)$$

となります。

関数 `transition` を書くためには、上述の定義の言葉を Objective Caml の言葉に翻訳すれば十分です。場合わけによる定義は、引数として渡されるキーについてのパターンマッチングによる定義となります。数字の場合は、それ自身が二つの場合から成り、ローカルな関数 `digit_transition` によって、最後に有効となったキーについてのパターンマッチングにより扱われます。

```
# let transition st ky =
    let digit_transition n = function
        Digit _ → { st with lka=ky; vpr=st.vpr*10+n }
      | _       → { st with lka=ky; vpr=n }
    in
      match ky with
          Digit p  →  digit_transition p st.lka
        | _        →  let res = evaluate st.lcd st.vpr st.loa
                      in { lcd=res; lka=ky; loa=ky; vpr=res }  ;;
val transition : state -> key -> state = <fun>
```

この関数は *state* と *key* をとり、新しい *state* を計算します。

こうして、このプログラムを先の例でテストすることができます。

```
# let initial_state = { lcd=0; lka=Equals; loa=Equals; vpr=0 } ;;
val initial_state : state = {lcd = 0; lka = Equals; loa = Equals; vpr = 0}
# let state2 = transition initial_state (Digit 3) ;;
val state2 : state = {lcd = 0; lka = Digit 3; loa = Equals; vpr = 3}
# let state3 = transition state2 Plus ;;
val state3 : state = {lcd = 3; lka = Plus; loa = Plus; vpr = 3}
# let state4 = transition state3 (Digit 2) ;;
val state4 : state = {lcd = 3; lka = Digit 2; loa = Plus; vpr = 2}
# let state5 = transition state4 (Digit 1) ;;
val state5 : state = {lcd = 3; lka = Digit 1; loa = Plus; vpr = 21}
# let state6 = transition state5 Times ;;
val state6 : state = {lcd = 24; lka = Times; loa = Times; vpr = 24}
# let state7 = transition state6 (Digit 2) ;;
val state7 : state = {lcd = 24; lka = Digit 2; loa = Times; vpr = 2}
# let state8 = transition state7 Equals ;;
val state8 : state = {lcd = 48; lka = Equals; loa = Equals; vpr = 48}
```

このような実行は、引数として渡されたキーのリストに対応する遷移の列を適用する関数を使って、より簡潔に書くことができます。

```
# let transition_list st ls = List.fold_left transition st ls ;;
val transition_list : state -> key list -> state = <fun>
# let example = [ Digit 3; Plus; Digit 2; Digit 1; Times; Digit 2; Equals ]
  in transition_list initial_state example ;;
- : state = {lcd = 48; lka = Equals; loa = Equals; vpr = 48}
```

練習問題

二つのリストをマージする

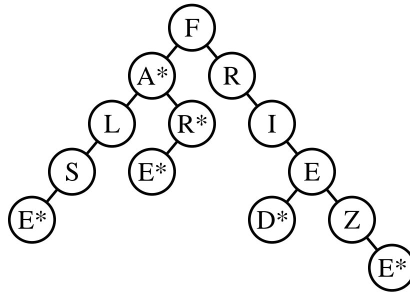
1. 昇順にソートされた二つの整数リストを入力として受け取り、それらの要素からなる新しいソートされたリストを返す関数 `merge_i` を書いてください。
2. 比較関数と、その順序によってソートされた二つのリストを受け取り、同じ順序でマージされたリストを返す一般的な関数 `merge` を書いてください。比較関数は型 *'a → 'a → bool* になります。
3. その関数を降順にソートされた二つの整数リストと、降順にソートされた二つの文字列リストにそれぞれ適用してください。
4. もしリストの一つが要求された降順になっていなかったら、何が起こるでしょうか?
5. 次の 3 つのフィールドからなる新しい *list* 型を書いてください：従来のリスト、順序関数、およびリストがその順序になっているかどうかを示す論理値。
6. その型のリストに新しい要素を加える関数 `insert` を書いてください。
7. リストの要素を挿入ソートする関数 `sort` を書いてください。
8. このリストについて新たに関数 `merge` を書いてください。

字句木

字句木（ないし**トライ**）は辞書を表現するのに用いられます。

```
# type lex_node = Letter of char * bool * lex_tree
  and  lex_tree = lex_node  list;;
# type word = string;;
```

lex_node のブール値が **true** だと、単語の終わりを表します。このような構造において、単語の列 “fa, false, far, fare, fried, frieze” は次のように格納されます。



アスタリスク (*) は単語の終わりを表します。

1. ある単語が型 *lex_tree* の辞書に属するかどうかテストする関数 `exists` を書いてください。
2. 単語と辞書を受け取って、その単語を加えた新しい辞書を返す関数 `insert` を書いてください。もしすでにその単語が辞書にあったら、挿入する必要はありません。
3. 単語のリストを受け取って、対応する辞書を構築する関数 `construct` を書いてください。
4. 単語のリストと辞書を受け取って、辞書に属さない単語のリストを返す関数 `verify` を書いてください。
5. 辞書と長さを受け取って、その長さの単語の集合を[14]返す関数 `select` を書いてください。

Graph traversal

各頂点の後続頂点からなる隣接リストにより有効グラフを表現する型 *'a graph* を定義します。

```
# type 'a graph = ( 'a * 'a list) list ;;
```

1. グラフに頂点を挿入し、新しいグラフを返す関数 `insert_vtx` を書いてください。
2. グラフに辺を追加する関数 `insert_edge` を書いてください。ただし、辺の両端の頂点はすでにグラフに含まれているものとします。
3. 与えられた頂点から直接つながっている、すべての頂点を返す関数 `has_edges_to` を書いてください。
4. 与えられた頂点へ直接つながっている、すべての頂点のリストを返す関数 `has_edges_from` を書いてください。

14. 訳注：リストとして

Summary

この章では、Objective Caml 言語の本質的特徴である、関数型プログラミングとパラメタ的多相性について解説しました。言語の関数的な核の部分の式の構文と、解説した型の構文を用いて、はじめてのプログラムをいくつか開発することができました。さらに、関数の型と定義域との深い違いについて注意しました。例外機構を導入することでこの問題を解消することができ、どのように計算が展開するか指定する新しいプログラミングスタイルが早くも示されました。

To learn more

関数型言語の計算モデルは、アロンゾ・チャーチにより 1932 年に提唱された λ 計算です。チャーチの目的は λ 定義可能性により実効的計算可能性の概念を定義することにありました。後に、このように導入された概念はチューリング（チューリング・マシン）やゲーデルとエルブラン（再帰関数）の意味での計算可能性と等価であることがわかりました。この一致から、特定の定式化によらない、普遍的な計算可能性の概念が存在すると考えられます。これがチャーチの提題です。この計算体系には、抽象と適用という二個の構成要素しかありません。データ構造（整数、ブール値、組、…）は λ 式として表されます。

関数型言語はこのモデルを実装し、より効率のよいデータ構造で拡張します。その最初の代表は Lisp です。効率のために、最初の関数型言語たちはメモリの物理的変更を実装したので、即時 (immediate) すなわち厳格 (strict) な評価戦略が必要でした。この戦略では、関数の引数は関数へ渡されるまえに評価されます。純粋な関数型言語について遅延評価（lazy あるいは call-by-need）の戦略が実装されたのは、Miranda、Haskell、あるいは LML といった他の言語であり、実は後のことです。

型推論をともなう静的型つけは、80 年代の初頭に ML 一族により促進されました。ウェブページ

リンク: http://www.pps.jussieu.fr/~cousinea/Caml/caml_history.html

では、ML の歴史について概観しています。ML の計算モデルは、λ 計算の部分集合である型つき λ 計算で、プログラムを実行している途中で型エラーが発生しないことを保証しています。しかし、「まったく正当な」プログラムが ML の型システムにより拒否されることもあります。そのようなケースは稀で、そういったプログラムは必ず型システムに合うように書き換えることができます。

もっともよく使われている関数型言語は、純粋でない関数型言語の代表である Lisp と ML の二つです。プログラミングに対する関数型のアプローチについて深く知るには、[ASS96] と [CM98] の本が、それぞれ Scheme（Lisp の方言）と Caml-Light を用いた一般的なプログラミングの道程を提示しています。

3

手続き型プログラミング

関数型プログラミングでは、関数に引数を渡してやることによって、何かの値を計算します。そこでどのように演算が実行されるのかということは気にしません。それに対して、手続き型プログラミングは計算機の表現により近くなります。そこにはメモリ状態というものがあり、プログラムの**命令** の実行によって変更されていきます。手続き型のプログラムは、命令の**列**からなります。それぞれの命令は、実行するとメモリ状態を変えることができます。入出力命令も、メモリや、ビデオメモリ、ファイルの変更であると考えることにします。

こういうプログラミングスタイルは、アセンブリプログラミングの発想からそのまま出てきたものです。また、初期の汎用言語の数々にも見ることができます（Fortran、C、Pascal など）。Objective Caml では、次の言語構成要素がこのモデルの当てはまります。

- 変更可能データ構造（配列や更新可能フィールドを持つレコードなど）
- 入出力演算
- 制御構造（ループや例外など）

アルゴリズムによっては、このプログラミングスタイルの方が簡単に書けます。例えば、行列の積の計算がそうです。たしかに、ベクトルの代わりにリストを使って、純粋な関数型に書き直すこともできますが、これは手続き型で書いたものと比べて自然でもなく、効率的でもありません。

手続き的な要素を関数型言語に取り込んだ動機は、あるアルゴリズムを書こうしたときに、手続き的スタイルが適切であるというときにはそれができるようにするということです。一方、純粋な関数型と比べ、欠点が主に２つあります。

- 言語の型システムが複雑になり、純粋関数型なら正しいように思えるプログラムも拒否されてしまうことになる。
- メモリ表現と計算順序を常に把握しておく必要が出る。

それでも、プログラムの書き方のガイドラインをいくつか用意しておけば、プログラミングスタイルの選択肢が複数あるということが、アルゴリズムを書く上で非常に大きな

柔軟性となります。それは、どんなプログラミング言語にとっても最大の目標です。それはともかくとしても、使用しているアルゴリズムに近い形で書かれたプログラムは、より単純でしょうし、正しく動く可能性が大きいでしょう（少なくとも、間違えてもすばやく訂正できるでしょう）。

このような理由から、Objective Caml 言語では、値を物理的に書き換えることのできるデータ構造が幾種類か、プログラムの実行を制御する構造、手続き的な入出力ライブラリが提供されています。

本章の流れ

この章では、前章に続いて、Objective Caml 言語の基本要素を紹介してきますが、今度は手続き的な構文に焦点を当てます。本章は5節からなります。第1節が最も重要です。ここで、いろいろな変更可能データ構造とそのメモリ表現を解説します。第2節では、この言語の基本的な入出力機能を簡単に説明します。第3節では、新しく登場する繰り返し制御構造についてお話しします。第4節では、手続き的な機能がプログラムの実行にどのような影響をおよぼすのか、特に、関数の引数の評価順序について論じます。最後の節では、前章の電卓の例に戻り、これをメモリつき計算機へと変身させていきます。

変更可能データ構造

要素の物理的変更が可能なデータ構造は、ベクトル、文字列、変更可能フィールドのあるレコード、参照です。

すでに見たように、Objective Caml の変数は、ある値に束縛されると、変数が生きている間ずっとこの値を保持し続けます。この束縛を変更するためには、変数の再定義が必要です。この場合、再定義された変数は、前と「同じ」変数であるとはあまりいえません。むしろ、同じ名前を持つ新しい変数が、前の変数を隠したのです。前の変数は、もはやアクセス不能でありながらも、同じ状態であり続けます。変更可能な値を使うと、新たな変数を再定義しなくても、変数に割り当てられている値を書き換えることができるようになります。変数の値には、書き込むためにも読み込むためにもアクセスできます。

ベクトル

ベクトル、もしくは一次元配列は、同じ型の要素を決まった数だけ持つようなデータ構造です。 ベクトルのひとつの書き方は、直接要素値を、[| と|] の間に、リスト同様セミコロンで区切って、書き並べるやりかたです。

```
# let v = [| 3.14; 6.28; 9.42 |] ;;
val v : float array = [|3.14; 6.28; 9.42|]
```

生成関数 Array.create に、ベクトルの要素数と初期値を渡して、新しいベクトルを作ることもできます。

```
# let v = Array.create 3 3.14;;
val v : float array = [|3.14; 3.14; 3.14|]
```

ある要素を参照したり書き換えたりするには、その要素のインデックスを与えてやります。

構文：$expr_1$ **.(** $expr_2$ **)**

構文：$expr_1$ **.(** $expr_2$ **) <-** $expr_3$

$expr_1$ は、ベクトル（*array* 型）で、その要素の型は $expr_3$ でなくてはなりません。式 $expr_2$ はもちろん *int* 型でなくてはなりません。上記の書き換えを行う式は *unit* 型を取ります。ベクトルの最初の要素のインデックスは 0 で、最後の要素のインデックスはベクトルの長さから 1 を引いた数です。インデックスの式は、必ず括弧で囲う必要があります。

```
# v.(1) ;;
- : float = 3.14
# v.(0) <- 100.0 ;;
- : unit = ()
# v ;;
- : float array = [|100; 3.14; 3.14|]
```

配列のインデックス範囲外のインデックスを使って要素参照を行うと、例外が発生します。

```
# v.(-1) +. 4.0;;
Exception: Invalid_argument "Array.get".
```

このチェックはプログラムの実行時に行われます。だから実行速度を落とすこともあります。それでもこれは必要不可欠なのです。そうしないと、ベクトルが割り当てられた記憶領域の外に書き込みが行われ、深刻な実行時エラーが起きるかも知れないからです。

配列操作をするための関数群は、標準ライブラリの Array モジュールの一部をなしています。これらは 8 章（217 ページ）で解説します。このあと出てくる例では、Array モジュールの中にある次の 3 つの関数を用います。

- create（与えられたサイズの配列を生成し、与えられた初期値で埋める）
- length（ベクトルの長さを返す）
- append（２つのベクトルを連結する）

ベクトル要素の値共有

ベクトルが生成されるとき、その要素に入るものは生成時に渡された値です。ということは、もしこの値が構造を持つ値である場合、その値は共有されることになります。例えば、Array モジュールの create 関数を使って、ベクトルのベクトルとして表現される行列を１つ作ってみましょう。

```
# let v = Array.create 3 0;;
val v : int array = [|0; 0; 0|]
# let m = Array.create 3 v;;
val m : int array array = [|[|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|]|]
```

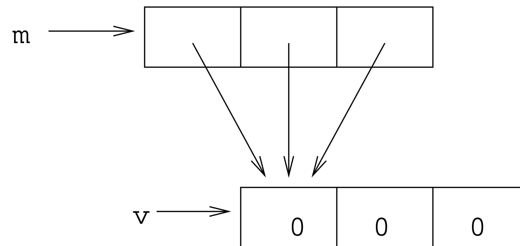


図 3.1: Memory representation of a vector sharing its elements.

今ベクトル v のひとつのフィールドを書き換えたとします。このベクトルは m の生成に使われたのですから、自動的にすべての行列の「行」をいっぺんに書き換えたことになります（図 3.1 と 3.2 を参照）。

```
# v.(0) <- 1;;
- : unit = ()
# m;;
- : int array array = [|[|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|]|]
```

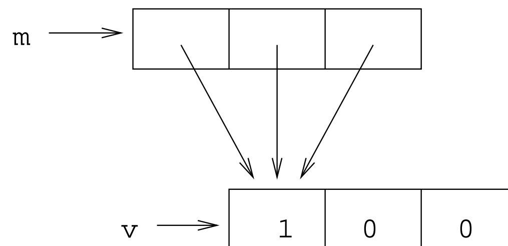


図 3.2: Modification of shared elements of a vector.

複製が起こるのは、ベクトルの初期値（Array.create の第 2 引数）が**原子的値**であるときで、共有が起こるのは、この値が構造を持つ値であるときです。

値の中で、サイズが Objective Caml の値の基準サイズ、つまりメモリワードサイズを越えないものは、原子的値と呼ばれます。整数、文字、真偽値、定数コンストラクタがそれにあたります。他の値、つまり構造を持つ値は、あるメモリ領域へのポインタとして表されます。この違いは 9 章に詳しく説明されます（249 ページ）。例外は浮動小数のベクトルです。浮動小数は構造を持つ値ですが、浮動小数ベクトルの生成では初期値はコピーされます。これは最適化のためです。12 章で、C 言語とのインタフェースについて解説するとき（317 ページ）に、この例外について説明します。

長方形でない行列

行列、すなわちベクトルのベクトルは、長方形である必要はありません。実際、ベクトルであるひとつの要素を、異なる長さのベクトルで置き換えてもいっこうに構わないのです。これは行列のサイズを小さく抑えるのに使い出があります。次に定義する値 t は、パスカルの三角形の係数を表す三角行列です。

```
# let t = [|
             [|1|];
             [|1; 1|];
             [|1; 2; 1|];
             [|1; 3; 3; 1|];
             [|1; 4; 6; 4; 1|];
             [|1; 5; 10; 10; 5; 1|]
           |] ;;
val t : int array array =
  [|[|1|]; [|1; 1|]; [|1; 2; 1|]; [|1; 3; 3; 1|]; [|1; 4; 6; 4; ...|]; ...|]
# t.(3) ;;
- : int array = [|1; 3; 3; 1|]
```

この例では、ベクトル t の i 番要素は、サイズ $i+1$ の整数ベクトルです。こういう行列を操作するためには、各要素ベクトルのサイズを計算する必要があります。

ベクトルのコピー

ベクトルをコピー、もしくはベクトルを２つ連結して得られる結果は新規のベクトルです。元のベクトルを書き換えても、複製の方は書き換えられることはありません。ただし、例によって値の共有があったりしなければの話です。

```
# let v2 = Array.copy v ;;
val v2 : int array = [|1; 0; 0|]
# let m2 = Array.copy m ;;
val m2 : int array array = [|[|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|]|]
# v.(1)<- 352;;
- : unit = ()
# v2;;
- : int array = [|1; 0; 0|]
# m2 ;;
- : int array array = [|[|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|]|]
```

この例では、m をコピーしても v へのポインタがコピーされるだけだということに注意しましょう。v の要素がひとつ書き換えられると、m2 もいっしょに書き換えられます。

連結によって作られる新しいベクトルのサイズは、元の２つのベクトルのサイズの和になります。

```
# let mm = Array.append m m ;;
val mm : int array array =
  [|[|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|];
    [|1; 352; ...|]; ...|]
# Array.length mm ;;
- : int = 6
```

```
# m.(0) <- Array.create 3 0 ;;
- : unit = ()
# m ;;
- : int array array = [|[|0; 0; 0|]; [|1; 352; 0|]; [|1; 352; 0|]|]
# mm ;;
- : int array array =
[|[|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|];
  [|1; 352; ...|]; ...|]
```

v の書き換えを行うと m と mm にも影響を及ぼします。v が m と mm に共有されている値からです。

```
# v.(1) <- 18 ;;
- : unit = ()
# mm;;
- : int array array =
[|[|1; 18; 0|]; [|1; 18; 0|]; [|1; 18; 0|]; [|1; 18; 0|]; [|1; 18; ...|];
  ...|]
```

文字列

文字列は、文字のベクトルの特殊な場合とも考えることができます。それでも、メモリの使用を効率化するために[1]、文字列の型は特別扱いになっています。要素のアクセスには特別な構文を用います。

構文 : $expr_1$ **.[** $expr_2$ **]**

文字列の要素は物理的に書き換えることができます。

構文 : $expr_1$ **.[** $expr_2$ **] <-** $expr_3$

```
# let s = "hello";;
val s : string = "hello"
# s.[2];;
- : char = 'l'
# s.[2]<-'Z';;
- : unit = ()
# s;;
- : string = "heZlo"
```

1. 32 ビットのワードに 1 文字 1 バイトとして 4 文字詰め込まれています。

レコードの変更可能フィールド

レコードのフィールドは、変更可能と宣言することができます。これをやるためにするべきことは、そのレコードの型宣言のときに、**mutable** というキーワードを使って明示することだけです。

構文： **type** *name* **=** { ... **;** **mutable** $name_i$ **:** *t* **;** ... }

次の小さな例では、平面上の点を表現するレコードの型を定義します。

```
# type point = { mutable xc : float; mutable yc : float } ;;
type point = { mutable xc : float; mutable yc : float; }
# let p = { xc = 1.0; yc = 0.0 } ;;
val p : point = {xc = 1; yc = 0}
```

そして、**mutable** と宣言されたフィールドの値は、次のような構文を用いて書き換えることができます。

構文： $expr_1$ **.** *name* **<-** $expr_2$

式 $expr_1$ は *name* フィールドを持つレコード型でなければなりません。書き換え演算の帰り値は *unit* 型の値です。

```
# p.xc <- 3.0 ;;
- : unit = ()
# p ;;
- : point = {xc = 3; yc = 0}
```

与えられた点を、その座標を書き換えて移動させる関数を書いてみましょう。副作用を伴う演算を順々に並べるのには、パターンマッチ付の局所宣言を用います。

```
# let moveto p dx dy =
    let () = p.xc <- p.xc +. dx
    in p.yc <- p.yc +. dy ;;
val moveto : point -> float -> float -> unit = <fun>
# moveto p 1.1 2.2 ;;
- : unit = ()
# p ;;
- : point = {xc = 4.1; yc = 2.2}
```

ひとつのレコードの定義で、変更可能なフィールドとそうでないフィールドを混ぜることもできます。**mutable** と指定されたフィールドだけが変更可能になります。

```
# type t = { c1 : int; mutable c2 : int } ;;
type t = { c1 : int; mutable c2 : int; }
# let r = { c1 = 0; c2 = 0 } ;;
val r : t = {c1 = 0; c2 = 0}
# r.c1 <- 1 ;;
Characters 0-9:
  r.c1 <- 1 ;;
```

```
  ^^^^^^^^^
The record field label c1 is not mutable
# r.c2 <- 1 ;;
- : unit = ()
# r ;;
- : t = {c1 = 0; c2 = 1}
```

80 ページに、変更可能フィールドと配列を用いてスタック構造を実装するという例を紹介します。

参照

Objective Caml は、多相型 ref を提供しています。これは、任意の値へのポインタを表す型と見ることができます。Objective Caml の用語では、値への**参照**と呼びます。参照値は変更可能です。*ref* 型は変更可能フィールドをひとつ持つようなレコードとして定義されています。

```
type 'a ref = {mutable contents:'a}
```

この型は省略記法として提供されています。参照先の値へと到達するには、プレフィックス関数 **(!)** を用います。参照の中身を書き換えるには、インフィックス関数 **(:=)** を用います。

```
# let x = ref 3 ;;
val x : int ref = {contents = 3}
# x ;;
- : int ref = {contents = 3}
# !x ;;
- : int = 3
# x := 4 ;;
- : unit = ()
# !x ;;
- : int = 4
# x := !x+1 ;;
- : unit = ()
# !x ;;
- : int = 5
```

多相性と書き換え可能な値

ref 型はパラメータつきの型です。だから、どんな型の値に対しても参照を作ることができるのです。しかしながら、参照される値の型には、ある制限を課す必要があります。多相型の値への参照を作っても良いようにするには、すこし警戒しなければなりません。

今、何の制限も課されていないとしましょう。すると、こういう宣言を書く人が出てくるでしょう。

```
let x = ref [] ;;
```

この変数 x は *'a list ref* 型を持つことになります。そして、この値が、Objective Caml の強い静的型付けと矛盾を来たすようなやり方で書き換えられてしまうかも知れません。

```
x := 1 :: !x ;;
x := true :: !x ;;
```

このように、ひとつの同じ変数が、あるときには *int list* 型をとり、そのあとには *bool list* 型をとるなどということが起こり得ます。

こういう状況を防ぐために、Objective Caml の型推論機構では、弱い型変数 という新しいカテゴリーの型変数を用います。文法的には、この型変数はアンダースコアを前につけて普通の型変数と区別されます。

```
# let x = ref [] ;;
val x : '_a list ref = {contents = []}
```

型変数 *'_a* は型引数ではなく、具体化されるのを待っている未知の型なのです。x の宣言のあと、それがはじめて使われるときに、*'_a* の取る値が定まり、それ以降は変わりません。（*'_a* に依存する型も同様に決定されます。）

```
# x := 0::!x ;;
- : unit = ()
# x ;;
- : int list ref = {contents = [0]}
```

これ以降は、変数 x はずっと *int list ref* 型をとります。

実は、未知のものを含む型は単相型です。その未知のものが何かが決まっていないのにも関わらずです。未知のものを多相型に具体化することはできません。

```
# let x = ref [] ;;
val x : '_a list ref = {contents = []}
# x := (function y → ())::!x ;;
- : unit = ()
# x ;;
- : ('_a -> unit) list ref = {contents = [<fun>]}
```

この例では、ある未知の型を別の型に具体化しました。その別の型は、その前の時点では多相的であった型（*'a -> unit*）でしたが、それにもかかわらず、具体化された型は新たな未知の型を含む単相型のままであり続けます。

この多相型に対する制限は、参照だけでなく、変更可能な部分を含むどんな値にも適用されます。つまり、ベクトル、**mutable** 宣言付のフィールドを少なくともひとつ含むレコード、などです。ここでの型引数は、変更可能な部分と関係ないものでも、すべて弱い型変数となります。

```
# type ('a,'b) t = { ch1 :'a list ; mutable ch2 : 'b list } ;;
type ('a, 'b) t = { ch1 : 'a list; mutable ch2 : 'b list; }
# let x = { ch1 = [] ; ch2 = [] } ;;
```

```
val x : ('_a, '_b) t = {ch1 = []; ch2 = []}
```

警告 関数適用の型付けが次のように変更されるために、純粋に関数的なプログラムにも影響が出ます。

同様に、多相関数を多相的な値に適用した場合、弱い型変数が出てきます。というのは、この関数が物理的変更が可能な値を作るかもしれない、という可能性を除外できないからです。別の言い方をすると、関数適用の結果は必ず単相型です。

```
# (function x → x) [] ;;
- : '_a list = []
```

部分適用でも、同様な結果が得られます。

```
# let f a b = a ;;
val f : 'a -> 'b -> 'a = <fun>
# let g = f 1 ;;
val g : '_a -> int = <fun>
```

もう一度多相型が欲しければ、f の第２引数を抽象化して、そしてそれを再び適用してあげる必要があります。

```
# let h x = f 1 x ;;
val h : 'a -> int = <fun>
```

事実上、h を定義している式は、関数式 **function** *x* → *f*
1 *x* です。これを評価すると関数クロージャが生成されますが、これは副作用を起こす危険はありません。それは関数の本体が評価されないからです。

一般に、「非拡張性」の式といわれるもの、つまり副作用を起こす危険が全くないと分かっている式と、そうでない「拡張性」の式は、区別されます。Objective Caml の型システムでは、式を構文の形でもって分類します。

- 「非拡張性」の式は主として、変数、変更不能な値のコンストラクタ、関数抽象を含みます。
- 「拡張性」の式は主として、関数適用、変更可能な値のコンストラクタを含みます。他にも、条件分岐、パターンマッチなどの制御構造もあります。

入出力

入出力関数は、何か値（たいがいは *unit* 型）を計算することはするのですが、その計算中には入出力装置の状態変化が起ります。それはキーボードバッファの状態変化、画面出力、ファイルへの書き込み、読み込みポインタの移動だったりします。初めから定義されている型に、*in_channel* と *out_channel* があります。それぞれ入力チャネルと出力チャネルを表します。ファイルの最後が来ると、End_of_file 例外が発生します。Unix 流の標準入力、標準出力、標準エラー出力に対応するものは、３つの定数 stdin、stdout、と stderr です。

チャネル

Objective Caml 標準ライブラリにある入出力関数が操作する対象は、**通信チャネル**、すなわち *in_channel* 型、もしくは *out_channel* 型の値です。チャネルは、最初から定義されている標準の 3 つ以外は、以下の関数のどちらかを使って作ります。

```
# open_in;;
- : string -> in_channel = <fun>
# open_out;;
- : string -> out_channel = <fun>
```

open_in は、指定ファイルを、もし存在したら[2]開きます。存在しなければ Sys_error 例外を発生させます。open_out は、指定ファイルを、もし存在しなければ作成します。存在すれば、元の内容を一旦破棄した上で開きます。

```
# let ic = open_in "koala";;
val ic : in_channel = <abstr>
# let oc = open_out "koala";;
val oc : out_channel = <abstr>
```

ファイルを閉じるための関数は、以下の通りです。

```
# close_in ;;
- : in_channel -> unit = <fun>
# close_out ;;
- : out_channel -> unit = <fun>
```

読み書き

読み書きのための最も一般的な関数は以下のものです。

```
# input_line ;;
- : in_channel -> string = <fun>
# input ;;
- : in_channel -> string -> int -> int -> int = <fun>
# output ;;
- : out_channel -> string -> int -> int -> unit = <fun>
```

- *input_line ic* は、入力チャネル ic から、最初の改行かファイル末尾までに出てくる、すべての文字を読み込み、それらを文字列で返します。(改行は含みません。)
- *input ic s p l* は、入力チャネル ic から、l 文字を読み込み、それを文字列 s の p 番目から後に保存します。実際に読み込んだ文字数が帰り値になります。
- *output oc s p l* は、出力チャネル oc へ、文字列 s の p 番目から l 文字を書き出します。

標準入出力から読み書きするには次の関数を使います。

```
# read_line ;;
- : unit -> string = <fun>
```

2. 適切な読み込み許可があった場合。

```
# print_string ;;
- : string -> unit = <fun>
# print_newline ;;
- : unit -> unit = <fun>
```

文字以外に、単純な型をもつ値で、そのまま読み込んだり書き出したりできるものもあります。それらは、文字リストに変換できるような型の値です。

局所宣言と評価順序　let x = e_1 in e_2 という形の式を用いて、印字の羅列のようなことができます。一般論として x が e_2 で使われる局所変数であるということを念頭に置くと、まず e_1 の評価が行われ、それから e_2 の番が来るということが分かるでしょう。今、これら 2 つの式として、手続き的で、() を返し、副作用も起こすような式を使えば、この評価順序がピッタリ合います。一続きの局所宣言を入れ子状に書けば、印字の羅列が実現できます。ここで、e_1 の返す値が *unit* 型の () であるということが分かっているので、局所宣言は、() に対してパターンマッチを行うようにできます。

```
# let () = print_string "and one," in
  let () = print_string " and two," in
  let () = print_string " and three" in
  print_string " zero";;
and one, and two, and three zero- : unit = ()
```

例：数当てゲーム

この例では「数当てゲーム」をやります。まずあらかじめ、ある数値が決められます。ユーザはそれを推測しなければなりません。プログラムは、毎回ユーザが数値を提示すると、その数があらかじめ決められた数値よりも大きいか小さいかを教えてくれます。

```
#  let rec hilo n =
      let () = print_string "type a number: " in
      let i = read_int ()
      in
        if i = n then
          let () = print_string "BRAVO" in
          let () = print_newline ()
          in print_newline ()
        else
          let () =
            if i < n then
              let () = print_string "Higher"
              in print_newline ()
            else
              let () = print_string "Lower"
```

```
              in print_newline ()
          in hilo n ;;
val hilo : int -> unit = <fun>
```

実行例です。

```
# hilo 64;;
type a number: 88
Lower
type a number: 44
Higher
type a number: 64
BRAVO

- : unit = ()
```

制御構造

入出力操作や値の変更操作は、副作用を起こします。これらを使うとき、ここで新しく登場する制御構造を取り入れて手続き的なプログラミングスタイルをすると楽になります。この節では、連接と繰り返しの構造を紹介します。

18 ページで条件分岐は見ましたが、それを省略した形 **if then** も手続き型の世界で良く出てきます。例えば、こういう風に書きます。

```
# let n = ref 1 ;;
val n : int ref = {contents = 1}
# if !n > 0 then n := !n - 1 ;;
- : unit = ()
```

連接

最も典型的な手続き型の構造は、**連接**です。連接は、式をセミコロンで区切って並べたもので、それらの式の評価は左から右へと行われます。

構文： $expr_1$ **;** ... **;** $expr_n$

複数の式を連接したもの自体も式です。その連接式の値となるのは、連接の最後におかれた式（この場合は $expr_n$）の値です。最終的な値に反映されないものの、最後以外の式も含めてすべての式が評価されます。副作用も実行されます。

```
# print_string "2 = "; 1+1 ;;
2 = - : int = 2
```

副作用を用いると、普通の手続き型言語に見られる構文を再現することができます。

```
# let x = ref 1 ;;
```

```
val x : int ref = {contents = 1}
# x:=!x+1 ; x:=!x*4 ; !x ;;
- : int = 8
```

セミコロンの前に出てくる式の値は破棄されるため、Objective Caml は、そういう式が *unit* 型でない場合には警告を発します。

```
# print_int 1; 2 ; 3 ;;
Characters 14-15:
Warning: this expression should have type unit.
  print_int 1; 2 ; 3 ;;
               ^
1- : int = 3
```

このメッセージが出ないようにするには、ignore 関数を使います。

```
# print_int 1; ignore 2; 3 ;;
1- : int = 3
```

値が関数型の場合、Objective Caml は関数に渡すべき引数を忘れたのではないかと疑って、違うメッセージを出します。

```
# let g x y = x := y ;;
val g : 'a ref -> 'a -> unit = <fun>
# let a = ref 10;;
val a : int ref = {contents = 10}
# let u = 1 in g a ; g a u ;;
Characters 13-16:
Warning: this function application is partial,
maybe some arguments are missing.
  let u = 1 in g a ; g a u ;;
               ^^^
- : unit = ()
# let u = !a in ignore (g a) ; g a u ;;
Characters 22-25:
Warning: this function application is partial,
maybe some arguments are missing.
  let u = !a in ignore (g a) ; g a u ;;
                        ^^^
- : unit = ()
```

一般に、連接は、その範囲を明確にするために括弧でくくった方が良いとされています。括弧の構文は２通りあります。

構文 : **(** *expr* **)**

構文 : **begin** *expr* **end**

これでようやく 76 ページでやった数当てゲームのプログラムをもっと自然な形に書き直すことができます。

```
# let rec hilo n =
    print_string "type a number: ";
    let i = read_int () in
    if i = n then print_string "BRAVO\n\n"
     else
       begin
         if i < n then print_string "Higher\n" else print_string "Lower\n" ;
         hilo n
       end ;;
val hilo : int -> unit = <fun>
```

繰り返し

繰り返し制御構造も、関数型の世界の外からきたものです。ループを続けるかどうか、もしくは抜けるかどうかを判断する条件式というものは、その値を変化させるようなメモリの物理的書き換えがない限り、意味をなしません。Objective Caml には 2 種類の繰り返し制御構造があります。ひとつは範囲限定の **for** ループ、もうひとつは範囲非限定の **while** ループです。ループ構造自身、式になります。ですから、値を返します。それは *unit* 型の定数 **()** です。

for ループは、1 ずつ上がっていく (**to**) ものと、1 ずつ下がっていく（**downto**）ものがあります。

構文：

for *name* **=** $expr_1$ **to** $expr_2$ **do** $expr_3$ **done**
for *name* **=** $expr_1$ **downto** $expr_2$ **do** $expr_3$ **done**

$expr_1$ と $expr_2$ は *int* 型です。もし $expr_3$ が *unit* 型でないと、コンパイラは警告を発します。

```
# for i=1 to 10 do  print_int i; print_string " " done; print_newline() ;;
1 2 3 4 5 6 7 8 9 10
- : unit = ()
# for i=10 downto 1 do print_int i; print_string " " done; print_newline() ;;
10 9 8 7 6 5 4 3 2 1
- : unit = ()
```

範囲非限定の **while** ループの文法は次の通りです。

構文： **while** $expr_1$ **do** $expr_2$ **done**

$expr_1$ 式は *bool* 型でなければなりません。**for** ループ同様、$expr_2$ が *unit* 型でないと、警告がでます。

```
# let r = ref 1
  in while !r < 11  do
      print_int !r ;
```

```
        print_string " " ;
        r := !r+1
      done ;;
1 2 3 4 5 6 7 8 9 10 - : unit = ()
```

ループが、前に出てきたものと同様に、*unit* 型の値 **()** を計算する式である、ということを理解しておくのは大事なことです。

```
# let f () = print_string "-- end\n" ;;
val f : unit -> unit = <fun>
# f (for i=1 to 10 do  print_int i; print_string " " done) ;;
1 2 3 4 5 6 7 8 9 10 -- end
- : unit = ()
```

整数 1 から 10 までが表示されたあとに、文字列"-- end\n"が出てくる、ということに注意してください。これから、引数（今の場合は **for** ループ）は関数に渡される前に評価されるということが分かるでしょう。

手続き型プログラミングでは、ループの本体（$expr_2$）は何か値を計算するわけではなく、副作用によって前進していきます。Objective Caml では、連接と同様に、ループの本体が *unit* 型でないと、警告がでます。

```
# let s = [5; 4; 3; 2; 1; 0] ;;
val s : int list = [5; 4; 3; 2; 1; 0]
#  for i=0 to 5 do List.tl s done ;;
Characters 17-26:
Warning: this expression should have type unit.
   for i=0 to 5 do List.tl s done ;;
                   ^^^^^^^^^
- : unit = ()
```

例：スタックの実装

データ構造 *'a stack* をレコードとして実装することにします。このレコードには、要素を保持する配列と、この配列中の一番初めの空きの位置が入っています。対応する型はこのようになります。

```
# type 'a stack = { mutable ind:int; size:int; mutable elts : 'a array } ;;
```

size フィールドには、スタックのサイズの上限が入っています。

スタックに対する操作には、スタックの初期化を行う init_stack、スタックへ要素をプッシュする push、スタックのトップを返し、それをポップする pop があります。

```
# let init_stack n = {ind=0; size=n; elts =[||]} ;;
val init_stack : int -> 'a stack = <fun>
```

この関数は、空でない配列を作るように書くことはできません。そうしたければ、その配列の初期値をこの関数が取るようにしなければならないでしょう。だから、elts フィールドは空配列で初期化するようにしています。

例外を 2 つ宣言しておきます。ひとつは空スタックからのポップをしようとするのを防ぐため、もうひとつは満杯のスタックへプッシュしようとするのを防ぐためです。これ

らは pop および push 関数で使われます。

```
# exception Stack_empty ;;
# exception Stack_full ;;

# let pop p =
    if p.ind = 0 then raise Stack_empty
    else (p.ind <- p.ind - 1; p.elts.(p.ind)) ;;
val pop : 'a stack -> 'a = <fun>
# let push e p =
    if p.elts = [||] then
      (p.elts <- Array.create p.size e;
       p.ind <- 1)
    else if p.ind >= p.size then raise Stack_full
    else (p.elts.(p.ind) <- e; p.ind <- p.ind + 1) ;;
val push : 'a -> 'a stack -> unit = <fun>
```

このデータ構造の簡単な使用例です。

```
# let p = init_stack 4 ;;
val p : '_a stack = {ind = 0; size = 4; elts = [||]}
# push 1 p ;;
- : unit = ()
# for i = 2 to 5 do push i p done ;;
Exception: Stack_full.
# p ;;
- : int stack = {ind = 4; size = 4; elts = [|1; 2; 3; 4|]}
# pop p ;;
- : int = 4
# pop p ;;
- : int = 3
```

もし、スタックへ要素をプッシュしようとしたときに Stack_full 例外が発生するのをやめさせたければ、配列の拡張をするという手があります。このとき、size フィールドも変えなければなりません。

```
# type 'a stack =
    {mutable ind:int ; mutable size:int ; mutable elts : 'a array} ;;
# let init_stack n = {ind=0; size=max n 1; elts = [||]} ;;
# let n_push e p =
    if p.elts = [||]
    then
      begin
        p.elts <- Array.create p.size e;
        p.ind <- 1
      end
    else if p.ind >= p.size then
      begin
        let nt = 2 * p.size in
```

```
        let nv = Array.create nt e in
        for j=0 to p.size-1 do nv.(j) <- p.elts.(j) done ;
        p.elts <- nv;
        p.size <- nt;
        p.ind <- p.ind + 1
      end
    else
      begin
        p.elts.(p.ind) <- e ;
        p.ind <- p.ind + 1
      end ;;
val n_push : 'a -> 'a stack -> unit = <fun>
```

上限なしに拡張できるようになりましたが、それでもなお、こういうデータ構造の扱いには注意が必要ということには変わりありません。この例では、最初のスタックが必要に応じて伸びていきます。

```
# let p = init_stack 4 ;;
val p : '_a stack = {ind = 0; size = 4; elts = [||]}
# for i = 1 to 5 do n_push i p done ;;
- : unit = ()
# p ;;
- : int stack = {ind = 5; size = 8; elts = [|1; 2; 3; 4; 5; 5; 5; 5|]}
# p.stack ;;
Characters 0-7:
  p.stack ;;
  ^^^^^^^
Unbound record field label stack
```

pop の方も、不要になったメモリを回収するためにスタックを縮めるようにすれば有用かも知れません。

例：行列計算

この例では、浮動小数の 2 次元配列として行列の型を定義し、そういう行列に対する演算をいくつか書くことにします。単相型 *mat* は、行列の次元と要素を保持するレコードです。関数 create_mat、access_mat、mod_mat は、それぞれ、行列の作成、要素の参照、要素の変更を行う関数です。

```
# type mat = { n:int; m:int; t: float array array };;
type mat = { n : int; m : int; t : float array array; }
# let create_mat n m = { n=n; m=m; t = Array.create_matrix n m 0.0 } ;;
val create_mat : int -> int -> mat = <fun>
# let access_mat m i j = m.t.(i).(j) ;;
val access_mat : mat -> int -> int -> float = <fun>
# let mod_mat m i j e = m.t.(i).(j) <- e ;;
val mod_mat : mat -> int -> int -> float -> unit = <fun>
```

```
# let a = create_mat 3 3 ;;
val a : mat = {n = 3; m = 3; t = [|[|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|]|]}
# mod_mat a 1 1 2.0; mod_mat a 1 2 1.0; mod_mat a 2 1 1.0 ;;
- : unit = ()
# a ;;
- : mat = {n = 3; m = 3; t = [|[|0; 0; 0|]; [|0; 2; 1|]; [|0; 1; 0|]|]}
```

行列 a と b の和を取ると、$c_{ij} = a_{ij} + b_{ij}$ を満たす行列 c となります。

```
# let add_mat p q =
    if p.n = q.n && p.m = q.m then
      let r = create_mat p.n p.m  in
      for i = 0 to p.n-1 do
        for j = 0 to p.m-1 do
          mod_mat r i j  (p.t.(i).(j) +. q.t.(i).(j))
        done
      done ;
      r
    else failwith "add_mat : dimensions incompatible";;
val add_mat : mat -> mat -> mat = <fun>
# add_mat  a a ;;
- : mat = {n = 3; m = 3; t = [|[|0; 0; 0|]; [|0; 4; 2|]; [|0; 2; 0|]|]}
```

行列 a と b の積を取ると、$c_{ij} = \sum_{k=1}^{k=m_a} a_{ik}.b_{kj}$ を満たす行列 c となります。

```
# let mul_mat p q =
    if p.m = q.n then
      let r = create_mat p.n q.m in
      for i = 0 to p.n-1 do
        for j = 0 to q.m-1 do
          let c = ref 0.0 in
          for k = 0 to p.m-1 do
              c := !c +.   (p.t.(i).(k) *. q.t.(k).(j))
           done;
           mod_mat r i j !c
        done
      done;
      r
    else failwith "mul_mat : dimensions incompatible" ;;
val mul_mat : mat -> mat -> mat = <fun>
# mul_mat a a;;
- : mat = {n = 3; m = 3; t = [|[|0; 0; 0|]; [|0; 5; 2|]; [|0; 2; 1|]|]}
```

引数の評価順序

純粋な関数型言語では、引数の評価順序が問題になることはありません。メモリ状態が更新されたり計算が中断されたりするすることがないので、ある引数の計算が別の引数にも影響したりすることがないわけです。一方、Objective Caml では、物理的に変更可能な値とか、例外とかがありますので、評価順序を考慮にいれないと危険です。例えば、次の結果は、Intel マシン上の Linux 用の Objective Caml バージョン 2.04 に特有のものです。

```
# let new_print_string s = print_string s; String.length s ;;
val new_print_string : string -> int = <fun>
# (+) (new_print_string "Hello ") (new_print_string "World!") ;;
World!Hello - : int = 12
```

これは２つの文字列を印字する例ですが、２番目の文字列が１番目のよりも先に出力されています。

例外についても同じことが起ります。

```
# try (failwith "function") (failwith "argument") with Failure s → s;;
- : string = "argument"
```

もし引数の評価順序を指定したければ、関数呼び出しの前に局所宣言を使って強制的に順序付けをしなければなりません。上の例だとこういう風に書き換えることができます。

```
# let e1 = (new_print_string "Hello ")
  in let e2 = (new_print_string "World!")
  in (+) e1 e2 ;;
Hello World!- : int = 12
```

Objective Caml では引数の評価順序は規定されていません。たまたま、現在の Objective Caml の実装では左から右へ引数を評価しています。それでもやはり、この実装に依存したプログラミングをしたりすると、将来のバージョンでこの実装方法が変更されたりしたときに、実は危険であったなんていうことがあるかも知れません。

あとで戻ってお話ししますが、言語デザインに関して果てしなく続いている論争があります。ある言語機能について、それは意図的に規定なしとしておくべきか？その場合、プログラマにはその機能は使わないでくれと頼むことになる。ただし、そのプログラマは、コンパイラの実装によってプログラムの結果が違ったりするということに苦しむことになる。それとも、すべては規定しておくべきか？その場合、プログラマは言語機能すべてを使ってもよいことになる。ただし、コンパイラの実装は複雑なるかも知れないし、使えない最適化もでてくるかも知れない。

メモリつき電卓

今から、前章ででてきた電卓の例をもう一度やります。ただし今回は、ユーザインターフェースを付け加え、より電卓的な使い方ができるようにします。新しい電卓プログラ

ムは、ひとつのループからなっていて、ユーザが、演算を入力し、その結果を画面上に見るということを繰り返します。また、毎回キー入力の度にいちいち遷移関数を適用しなくてもよくなります。

４つのキーを新たに追加します。画面をゼロにクリアする **C**、結果を記憶する **M**、記憶したものをよみがえらせる **m**、電卓をスイッチオフする **OFF** です。これらに対応して次の型を定義します。

```
# type key = Plus | Minus | Times | Div | Equals | Digit of int
           | Store | Recall | Clear | Off ;;
```

キーボードに打ち込まれた文字を *key* 型に変換する関数を定義する必要があります。打ち込まれた文字が電卓のどのキーにも相当しない場合には、Invalid_key 例外で対処します。文字をアスキーコードに変換するには、Char モジュールの code 関数を使います。

```
# exception Invalid_key ;;
exception Invalid_key
# let translation c = match c with
      '+' → Plus
    | '-' → Minus
    | '*' → Times
    | '/' → Div
    | '=' → Equals
    | 'C' | 'c' → Clear
    | 'M' → Store
    | 'm' → Recall
    | 'o' | 'O' → Off
    | '0'..'9' as c → Digit ((Char.code c) - (Char.code '0'))
    | _ → raise Invalid_key ;;
val translation : char -> key = <fun>
```

手続き型のスタイルでは、遷移関数は、新しい状態を計算するのではなく、電卓の状態を更新します。そのため、state 型を再定義して、フィールドを変更可能にしてやる必要があります。最後に、**OFF** キーが押されたときに対応するため、Key_off 例外を定義しておきます。

```
# type state = {
   mutable lcd : int;  (* last computation done   *)
   mutable lka : bool; (* last key activated      *)
   mutable loa : key;  (* last operator activated *)
   mutable vpr : int;  (* value printed           *)
   mutable mem : int   (* memory of calculator    *)
  };;
```

```
# exception Key_off ;;
exception Key_off
```

```
# let transition s key = match key with
      Clear →   s.vpr <- 0
    | Digit n → s.vpr <- ( if s.lka then s.vpr*10+n else n );
                s.lka <- true
    | Store  →  s.lka <- false ;
                s.mem <- s.vpr
    | Recall →  s.lka <- false ;
                s.vpr <- s.mem
    | Off → raise Key_off
    |  _  → let lcd = match s.loa with
                          Plus  → s.lcd + s.vpr
                        | Minus → s.lcd - s.vpr
                        | Times  → s.lcd * s.vpr
                        | Div   → s.lcd / s.vpr
                        | Equals  → s.vpr
                        | _ → failwith "transition: impossible match"
             in
               s.lcd <- lcd ;
               s.lka <- false ;
               s.loa <- key ;
               s.vpr <- s.lcd;;
val transition : state -> key -> unit = <fun>
```

電卓を開始させる go 関数を定義します。帰り値は () ですが、これはプログラムの実行によって引き起こされた環境への作用（開始／終了／状態の更新）だけが今の関心事だからです。引数も定数 () ですが、これはこの電卓が自立して（つまり自分で初期状態を設定する)、外界と呼応しながら（つまりキーボード入力を引数としながら計算が進む）動作するからです。状態遷移は無限ループ（**while** true **do**）の中で行われます。抜け出すために、Key_off 例外を使います。

```
# let go () =
    let state = { lcd=0; lka=false; loa=Equals; vpr=0; mem=0 }
    in try
         while true do
           try
             let input = translation (input_char stdin)
             in transition state input ;
                print_newline () ;
                print_string "result: " ;
                print_int state.vpr ;
                print_newline ()
           with
             Invalid_key → ()  (* no effect *)
         done
        with
          Key_off → () ;;
```

```
val go : unit -> unit = <fun>
```

ここで、初期状態は、パラメータとして渡すか、go 関数の中で局所的に宣言しなければならない、ということに注意しましょう。これは、この関数を適用する度に状態は初期化してやる必要があるからです。もし前章の関数型プログラムでやったように initial_state を使うようにすると、電卓はいったん停止したあと、再開したときに、前と同じ状態から始まることになります。同じプログラムで電卓を２つ使うなどということも難しくなるでしょう。

練習問題

二重リンクリスト

関数型プログラミングは、リストのようなサイクルのないデータ構造を扱いやすいようにできています。一方、サイクルのある構造を実装するのは非常に大変です。ここでは、二重リンクリストを定義しましょう。これは、各要素が、前と後の要素を保持しているようなリストのことです。

1. 変更可能フィールドを少なくとも１つもっているレコードを使って、二重リンクリストを表すパラメータつき型を定義してください。
2. 二重リンクリストへ要素を追加する add 関数、要素を削除する remove 関数を書いてください。

連立一次方程式を解く

行列の代数に関する練習問題です。与えられた連立方程式をガウスの除去法（もしくはピボット法）によって解きます。解くべき連立方程式は $A\,X = Y$ と書くことにします。ただし、A は n 次元正方行列、Y は n 次元定数ベクトル、X は同次元の未知のベクトルです。

この手法では、まず方程式 $A\,X = Y$ をある等価な方程式 $C\,X = Z$ に変形します。ただし、C は上方三角行列です。それから、C を対角化して解を求めます。

1. *vect* 型、*mat* 型、*syst* 型を定義してください。
2. ベクトルを操作する補助関数を書いてください。具体的には、方程式の画面表示、２つのベクトルの加算、ベクトルのスカラーによる乗算です。
3. 行列を操作する補助関数を書いてください。具体的には、２つの行列の乗算、行列とベクトルの乗算です。
4. 方程式を操作する補助関数を書いてください。具体的には、方程式のある行を、あるピボット（A_{ij}）で割って、２つの行を入れ替えるという関数です。
5. 方程式を対角化する関数を書いてください。これから、連立一次方程式の解を求める関数を書いてください。

6. 次の方程式について、あなたの書いた関数をテストしてください。

$$AX = \begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix} = Y$$

$$AX = \begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 32.1 \\ 22.9 \\ 33.1 \\ 30.9 \end{pmatrix} = Y$$

$$AX = \begin{pmatrix} 10 & 7 & 8.1 & 7.2 \\ 7.08 & 5.04 & 6 & 5 \\ 8 & 5.98 & 9.89 & 9 \\ 6.99 & 4.99 & 9 & 9.98 \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix} = Y$$

7. あなたの得た結果についてどう思いますか？

まとめ

この章では、手続き型プログラミングで主流の機能（変更可能な値、入出力、繰り返し制御構造）を関数型言語へと統合するということについてお話ししました。文字列、配列、レコードなどの変更可能な値だけが、物理的に書き換えをすることができます。そのほかの値は、いったん生成されると、書き換えることはできません。このようにして、関数的な部分として読み込み専用の値が、手続き的な部分として読み書き可能な値があることになるのです。

ひとつ留意しておくべきことは、もしこの言語の手続き的な機能を使わない場合、この関数的コアへの機能拡張は、関数的な部分に影響がありません。ただし、手続き的な機能を導入する上で回避しなければならなかった型付け上の問題がありましたが、それは除いての話です。

もっと知りたい方へ

手続き型プログラミングは、Fortran、C、Pascal などの初期の計算機言語の頃から最も広く使われ続けてきたプログラミングスタイルです。そういう理由で、数々のアルゴリズムはこのスタイルで記述されてきました。多くは、なんらかの疑似 Pascal が使われてきました。これらは関数的スタイルでも実装できますが、配列を使うので手続き的スタイルを使った方がやりやすいのです。古典的なアルゴリズムの教科書（例えば [AHU83] や [Sed88]）に出ているデータ構造とアルゴリズムは、この適切なスタイルでそのままもってくることができます。また、この２つのスタイルを一つの言語の中にいっしょにすると、２つを合わせた新しいプログラミングモデルを定義することができるという長所が生まれます。これがまさにつぎの章でお話しすることです。

4

Functional and Imperative Styles

Functional and imperative programming languages are primarily distinguished by the control over program execution and the data memory management.

- A functional program computes an expression. This computation results in a value. The order in which the operations needed for this computation occur does not matter, nor does the physical representation of the data manipulated, because the result is the same anyway. In this setting, deallocation of memory is managed implicitly by the language itself: it relies on an automatic garbage collector or **GC**; see chapter 9.
- An imperative program is a sequence of instructions modifying a memory state. Each execution step is enforced by rigid control structures that indicate the next instruction to be executed. Imperative programs manipulate pointers or references to values more often than the values themselves. Hence, the memory space needed to store values must be allocated and reclaimed explicitly, which sometimes leads to errors in accessing memory. Nevertheless, nothing prevents use of a GC.

Imperative languages provide greater control over execution and the memory representation of data. Being closer to the actual machine, the code can be more efficient, but loses in execution safety. Functional programming, offering a higher level of abstraction, achieves a better level of execution safety: Typing (dynamic or static) may be stricter in this case, thus avoiding operations on incoherent values. Automatic storage reclamation, in exchange for giving up efficiency, ensures the current existence of the values being manipulated.

Historically, the two programming paradigms have been seen as belonging to different universes: symbolic applications being suitable for the former, and numerical applications being suitable for the latter. But certain things have changed, especially techniques for compiling functional programming languages, and the efficiency of GCs. From another side, execution safety has become an important, sometimes the predominant criterion in the quality of an application. Also familiar is the “selling point” of

the *Java* language, according to which efficiency need not preempt assurance, especially if efficiency remains reasonably good. And this idea is spreading among software producers.

Objective Caml belongs to this class. It combines the two programming paradigms, thus enlarging its domain of application by allowing algorithms to be written in either style. It retains, nevertheless, a good degree of execution safety because of its static typing, its GC, and its exception mechanism. Exceptions are a first explicit execution control structure; they make it possible to break out of a computation or restart it. This trait is at the boundary of the two models, because although it does not replace the result of a computation, it can modify the order of execution. Introducing physically mutable data can alter the behavior of the purely functional part of the language. For instance, the order in which the arguments to a function are evaluated can be determined, if that evaluation causes side effects. For this reason, such languages are called "impure functional languages." One loses in level of abstraction, because the programmer must take account of the memory model, as well as the order of events in running the program. This is not always negative, especially for the efficiency of the code. On the other hand, the imperative aspects change the type system of the language: some functional programs, correctly typed in theory, are no longer in fact correctly typed because of the introduction of references. However, such programs can easily be rewritten.

Plan of the Chapter

This chapter provides a comparison between the functional and imperative models in the Objective Caml language, at the level both of control structure and of the memory representation of values. The mixture of these two styles allows new data structures to be created. The first section studies this comparison by example. The second section discusses the ingredients in the choice between composition of functions and sequencing of instructions, and in the choice between sharing and copying values. The third section brings out the interest of mixing these two styles to create mutable functional data, thus permitting data to be constructed without being completely evaluated. The fourth section describes *streams*, potentially infinite sequences of data, and their integration into the language via pattern-matching.

Comparison between Functional and Imperative

Character strings (of Objective Caml type `string`) and linked lists (of Objective Caml type `'a list`) will serve as examples to illustrate the differences between "functional" and "imperative."

The Functional Side

The function `map` (see page 26) is a classic ingredient in functional languages. In a purely functional style, it is written:

```
# let rec map f l = match l with
        []    →   []
    | h::q  →   (f h) :: (map f q)  ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

It recursively constructs a list by applying `f` to the elements of the list given as argument, independently specifying its head `(f h)` and its tail `(map f q)`. In particular, the program does not stipulate which of the two will be computed first.

Moreover, the physical representation of lists need not be known to the programmer to write such a function. In particular, problems of allocating and sharing data are managed implicitly by the system and not by the programmer. An example illustrating this follows:

```
# let example = [ "one" ; "two" ; "three" ] ;;
val example : string list = ["one"; "two"; "three"]
# let result = map (function x → x) example  ;;
val result : string list = ["one"; "two"; "three"]
```

The lists `example` and `result` contain equal values:

```
# example = result ;;
- : bool = true
```

These two values have exactly the same structure even though their representation in memory is different, as one learns by using the test for physical equality:

```
# example == result ;;
- : bool = false
# (List.tl example) == (List.tl result) ;;
- : bool = false
```

The Imperative Side

Let us continue the previous example, and modify a string in the list `result`.

```
# (List.hd result).[1] <- 's' ;;
- : unit = ()
# result ;;
- : string list = ["ose"; "two"; "three"]
# example ;;
- : string list = ["ose"; "two"; "three"]
```

Evidently, this operation has modified the list `example`. Hence, it is necessary to know the physical structure of the two lists being manipulated, as soon as we use imperative aspects of the language.

Let us now observe how the order of evaluating the arguments of a function can amount to a trap in an imperative program. We define a mutable list structure with primitive functions for creation, modification, and access:

```
# type 'a ilist = { mutable c : 'a list } ;;
type 'a ilist = { mutable c : 'a list; }
# let icreate () = { c = [] }
  let iempty l = (l.c = [] )
  let icons x y = y.c <- x :: y.c ; y
  let ihd x = List.hd x.c
  let itl x = x.c <- List.tl x.c ; x ;;
val icreate : unit -> 'a ilist = <fun>
val iempty : 'a ilist -> bool = <fun>
val icons : 'a -> 'a ilist -> 'a ilist = <fun>
val ihd : 'a ilist -> 'a = <fun>
val itl : 'a ilist -> 'a ilist = <fun>
# let rec imap f l =
    if iempty l then icreate()
    else icons (f (ihd l))  (imap f (itl l)) ;;
val imap : ('a -> 'b) -> 'a ilist -> 'b ilist = <fun>
```

Despite having reproduced the general form of the `map` of the previous paragraph, with `imap` we get a distinctly different result:

```
# let example = icons "one" (icons "two" (icons "three" (icreate()))) ;;
val example : string ilist = {c = ["one"; "two"; "three"]}
# imap (function x → x) example ;;
Exception: Failure "hd".
```

What has happened? Just that the evaluation of `(itl l)` has taken place before the evaluation of `(ihd l)`, so that on the last iteration of `imap`, the list referenced by `l` became the empty list before we examined its head. The list `example` is henceforth definitely empty even though we have not obtained any result:

```
# example ;;
- : string ilist = {c = []}
```

The flaw in the function `imap` arises from a mixing of the genres that has not been controlled carefully enough. The choice of order of evaluation has been left to the system. We can reformulate the function `imap`, making explicit the order of evaluation, by using the syntactic construction **`let .. in ..`**

```
# let rec imap f l =
    if iempty l then icreate()
    else let h = ihd l  in  icons (f h)  (imap f (itl l)) ;;
val imap : ('a -> 'b) -> 'a ilist -> 'b ilist = <fun>
# let example = icons "one" (icons "two" (icons "three" (icreate()))) ;;
val example : string ilist = {c = ["one"; "two"; "three"]}
# imap (function x → x) example ;;
```

```
- : string ilist = {c = ["one"; "two"; "three"]}
```

However, the original list has still been lost:

```
# example ;;
- : string ilist = {c = []}
```

Another way to make the order of evaluation explicit is to use the sequencing operator and a looping structure.

```
# let imap f l =
     let l_res = icreate ()
     in while not (iempty l) do
          ignore (icons (f (ihd l)) l_res) ;
          ignore (itl l)
        done ;
        { l_res with c = List.rev l_res.c } ;;
val imap : ('a -> 'b) -> 'a ilist -> 'b ilist = <fun>
# let example = icons "one" (icons "two" (icons "three" (icreate()))) ;;
val example : string ilist = {c = ["one"; "two"; "three"]}
# imap (function x → x) example ;;
- : string ilist = {c = ["one"; "two"; "three"]}
```

The presence of ignore emphasizes the fact that it is not the result of the functions that counts here, but their side effects on their argument. In addition, we had to put the elements of the result back in the right order (using the function List.rev).

Recursive or Iterative

People often mistakenly associate recursive with functional and iterative with imperative. A purely functional program cannot be iterative because the value of the condition of a loop never varies. By contrast, an imperative program may be recursive: the original version of the function imap is an example.

Calling a function conserves the values of its arguments during its computation. If it calls another function, the latter conserves its own arguments in addition. These values are conserved on the **execution stack**. When the call returns, these values are popped from the stack. The memory space available for the stack being bounded, it is possible to encounter the limit when using a recursive function with calls too deeply nested. In this case, Objective Caml raises the exception Stack_overflow.

```
# let rec succ n = if n = 0 then 1 else 1 + succ (n-1) ;;
val succ : int -> int = <fun>
# succ 100000 ;;
Stack overflow during evaluation (looping recursion?).
```

In the iterative version `succ_iter`, the stack space needed for a call does not depend on its argument.

```
# let succ_iter n =
    let i = ref 0 in
      for j=0 to n do incr i done ;
      !i ;;
val succ_iter : int -> int = <fun>
# succ_iter 100000 ;;
- : int = 100001
```

The following recursive version has *a priori* the same depth of calls, yet it executes successfully with the same argument.

```
# let succ_tr n =
    let rec succ_aux n accu =
      if n = 0 then accu  else succ_aux (n-1) (accu+1)
    in
      succ_aux 1 n ;;
val succ_tr : int -> int = <fun>
# succ_tr 100000 ;;
- : int = 100001
```

This function has a special form of recursive call, called **tail recursion**, in which the result of this call will be the result of the function without further computation. It is therefore unnecessary to have stored the values of the arguments to the function while computing the recursive call. When Objective Caml can observe that a call is tail recursive, it frees the arguments on the stack before making the recursive call. This optimization allows recursive functions that do not increase the size of the stack.

Many languages detect tail recursive calls, but it is indispensable in a functional language, where naturally many tail recursive calls are used.

Which Style to Choose?

This is no matter of religion or esthetics; *a priori* neither style is prettier or holier than the other. On the contrary, one style may be more adequate than the other depending on the problem to be solved.

The first rule to apply is the rule of simplicity. Whether the algorithm to use implemented is written in a book, or whether its seed is in the mind of the programmer, the algorithm is itself described in a certain style. It is natural to use the same style when implementing it.

The second criterion of choice is the efficiency of the program. One may say that an imperative program (if well written) is more efficient that its functional analogue, but in very many cases the difference is not enough to justify complicating the code to

adopt an imperative style where the functional style would be natural. The function `map` in the previous section is a good example of a problem naturally expressed in the functional style, which gains nothing from being written in the imperative style.

Sequence or Composition of Functions

We have seen that as soon as a program causes side effects, it is necessary to determine precisely the order of evaluation for the elements of the program. This can be done in both styles:

functional: using the fact that Objective Caml is a **strict language**, which means that the argument is evaluated before applying the function. The expression `(f (g x))` is computed by first evaluating `(g x)`, and then passing the result as argument to `f`. With more complex expressions, we can name an intermediate result with the **`let in`** construction, but the idea remains the same: `let aux=(g x) in (f aux)`.

imperative: using sequences or other control structures (loops). In this case, the result is not the value returned by a function, but its side effects on memory: `aux:=(g x) ; (f !aux)`.

Let us examine this choice of style on an example. The **quick sort** algorithm, applied to a vector, is described recursively as follows:

1. Choose a pivot: This is the index of an element of the vector;
2. Permute around the pivot: Permute the elements of the vector so elements less than the value at the pivot have indices less than the pivot, and vice versa;
3. sort the subvectors obtained on each side of the pivot, using the same algorithm: The subvector preceding the pivot and the subvector following the pivot.

The choice of algorithm, namely to modify a vector so that its elements are sorted, incites us to use an imperative style at least to manipulate the data.

First, we define a function to permute two elements of a vector:

```
# let permute_element vec n p =
    let aux = vec.(n) in vec.(n) <- vec.(p) ; vec.(p) <- aux  ;;
val permute_element : 'a array -> int -> int -> unit = <fun>
```

The choice of a good pivot determines the efficiency of the algorithm, but we will use the simplest possible choice here: return the index of the first element of the (sub)vector.

```
# let choose_pivot vec start finish = start ;;
val choose_pivot : 'a -> 'b -> 'c -> 'b = <fun>
```

Let us write the algorithm that we would like to use to permute the elements of the vector around the pivot.

1. Place the pivot at the beginning of the vector to be permuted;
2. Initialize i to the index of the second element of the vector;
3. Initialize j to the index of the last element of the vector;
4. If the element at index j is greater than the pivot, permute it with the element at index i and increment i; otherwise, decrement j;
5. While $i < j$, repeat the previous operation;
6. At this stage, every element with index $< i$ (or equivalently, j) is less than the pivot, and all others are greater; if the element with index i is less than the pivot, permute it with the pivot; otherwise, permute its predecessor with the pivot.

In implementing this algorithm, it is natural to adopt imperative control structures.

```
# let permute_pivot vec start finish ind_pivot =
      permute_element vec start ind_pivot ;
      let i = ref (start+1) and j = ref finish and pivot = vec.(start) in
      while !i < !j do
        if vec.(!j) >= pivot then decr j
        else
          begin
            permute_element vec !i !j ;
            incr i
          end
      done ;
      if vec.(!i) > pivot then decr i ;
      permute_element vec start !i ;
      !i
  ;;
val permute_pivot : 'a array -> int -> int -> int -> int = <fun>
```

In addition to its effects on the vector, this function returns the index of the pivot as its result.

All that remains is to put together the different stages and add the recursion on the sub-vectors.

```
# let rec quick vec start finish =
      if start < finish
      then
        let pivot = choose_pivot vec start finish in
        let place_pivot = permute_pivot vec start finish pivot in
        quick (quick vec start (place_pivot-1)) (place_pivot+1) finish
      else vec ;;
val quick : 'a array -> int -> int -> 'a array = <fun>
```

We have used the two styles here. The chosen pivot serves as argument to the permutation around this pivot, and the index of the pivot after the permutation is an argument to the recursive call. By contrast, the vector obtained after the permutation is not returned by the `permute_pivot` function; instead, this result is produced by side

effect. However, the `quick` function returns a vector, and the sorting of sub-vectors is obtained by composition of recursive calls.

The main function is:

```
# let quicksort vec = quick vec 0 ((Array.length vec)-1) ;;
val quicksort : 'a array -> 'a array = <fun>
```

It is a polymorphic function because the order relation `<` on vector elements is itself polymorphic.

```
# let t1 = [|4;8;1;12;7;3;1;9|] ;;
val t1 : int array = [|4; 8; 1; 12; 7; 3; 1; 9|]
# quicksort t1 ;;
- : int array = [|1; 1; 3; 4; 7; 8; 9; 12|]
# t1 ;;
- : int array = [|1; 1; 3; 4; 7; 8; 9; 12|]
# let t2 = [|"the"; "little"; "cat"; "is"; "dead"|] ;;
val t2 : string array = [|"the"; "little"; "cat"; "is"; "dead"|]
# quicksort t2 ;;
- : string array = [|"cat"; "dead"; "is"; "little"; "the"|]
# t2 ;;
- : string array = [|"cat"; "dead"; "is"; "little"; "the"|]
```

Shared or Copy Values

When the values that we manipulate are not mutable, it does not matter whether they are shared or not.

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
# let a = [ 1; 2; 3 ] ;;
val a : int list = [1; 2; 3]
# let b = id a ;;
val b : int list = [1; 2; 3]
```

Whether `b` is a copy of the list `a` or the very same list makes no difference, because these are intangible values anyway. But if we put modifiable values in place of integers, we need to know whether modifying one value causes a change in the other.

The implementation of polymorphism in Objective Caml causes immediate values to be copied, and structured values to be shared. Even though arguments are always passed by value, only the pointer to a structured value is copied. This is the case even in the function `id`:

```
# let a = [| 1 ; 2 ; 3 |] ;;
val a : int array = [|1; 2; 3|]
# let b = id a ;;
val b : int array = [|1; 2; 3|]
# a.(1) <- 4 ;;
- : unit = ()
# a ;;
```

```
- : int array = [|1; 4; 3|]
# b ;;
- : int array = [|1; 4; 3|]
```

We have here a genuine programming choice to decide which is the most efficient way to represent a data structure. On one hand, using mutable values allows manipulations in place, which means without allocation, but requires us to make copies sometimes when immutable data would have allowed sharing. We illustrate this here with two ways to implement lists.

```
# type 'a list_immutable = LInil | LIcons of 'a * 'a list_immutable ;;
# type 'a list_mutable = LMnil | LMcons of 'a * 'a list_mutable ref ;;
```

The immutable lists are strictly equivalent to lists built into Objective Caml, while the mutable lists are closer to the style of C, in which a cell is a value together with a reference to the following cell.

With immutable lists, there is only one way to write concatenation, and it requires duplicating the structure of the first list; by contrast, the second list may be shared with the result.

```
# let rec concat l1 l2 = match l1 with
      LInil → l2
    | LIcons (a,l11) → LIcons(a, (concat l11 l2)) ;;
val concat : 'a list_immutable -> 'a list_immutable -> 'a list_immutable =
  <fun>

# let li1 = LIcons(1, LIcons(2, LInil))
  and li2 =  LIcons(3, LIcons(4, LInil)) ;;
val li1 : int list_immutable = LIcons (1, LIcons (2, LInil))
val li2 : int list_immutable = LIcons (3, LIcons (4, LInil))
# let li3 = concat li1 li2 ;;
val li3 : int list_immutable =
  LIcons (1, LIcons (2, LIcons (3, LIcons (4, LInil))))
# li1==li3 ;;
- : bool = false
# let tlLI l =  match l with
    LInil       → failwith "Liste vide"
  | LIcons(_,x) → x ;;
val tlLI : 'a list_immutable -> 'a list_immutable = <fun>
# tlLI(tlLI(li3)) == li2 ;;
- : bool = true
```

From these examples, we see that the first cells of `li1` and `li3` are distinct, while the second half of `li3` is exactly `li2`.

With mutable lists, we have a choice between modifying arguments (function `concat_share`) and creating a new value (function `concat_copy`).

```
# let rec concat_copy l1 l2 =  match l1 with
      LMnil  → l2
    | LMcons (x,l11) → LMcons(x, ref (concat_copy !l11 l2)) ;;
```

```
val concat_copy : 'a list_mutable -> 'a list_mutable -> 'a list_mutable =
  <fun>
```

This first solution, concat_copy, gives a result similar to the previous function, concat. A second solution shares its arguments with its result fully:

```
# let concat_share l1 l2 =
   match l1 with
     LMnil → l2
   | _     → let rec set_last = function
                 LMnil       → failwith "concat_share : impossible case!!"
               | LMcons(_,l) → if !l=LMnil then l:=l2 else set_last !l
             in
               set_last l1 ;
               l1 ;;
val concat_share : 'a list_mutable -> 'a list_mutable -> 'a list_mutable =
  <fun>
```

Concatenation with sharing does not require any allocation, and therefore does not use the constructor LMcons. Instead, it suffices to cause the last cell of the first list to point to the second list. However, this version of concatenation has the potential weakness that it alters arguments passed to it.

```
# let lm1 = LMcons(1, ref (LMcons(2, ref LMnil)))
  and lm2 = LMcons(3, ref (LMcons(4, ref LMnil))) ;;
val lm1 : int list_mutable =
  LMcons (1, {contents = LMcons (2, {contents = LMnil})})
val lm2 : int list_mutable =
  LMcons (3, {contents = LMcons (4, {contents = LMnil})})
# let lm3 = concat_share lm1 lm2 ;;
val lm3 : int list_mutable =
  LMcons (1, {contents = LMcons (2, {contents = LMcons (...)})})
```

We do indeed obtain the expected result for lm3. However, the value bound to lm1 has been modified.

```
# lm1 ;;
- : int list_mutable =
LMcons (1, {contents = LMcons (2, {contents = LMcons (...)})})
```

This may therefore have consequences on the rest of the program.

How to Choose your Style

In a purely functional program, side effects are forbidden, and this excludes mutable data structures, exceptions, and input/output. We prefer, though, a less restrictive definition of the functional style, saying that functions that do not modify their global environment may be used in a functional style. Such a function may manipulate mutable values locally, and may therefore be written in an imperative style, but must not modify global variables, nor its arguments. We permit them to raise exceptions in addition. Viewed from outside, these functions may be considered “black boxes.” Their behavior matches a function written in a purely functional style, apart from being able of breaking control flow by raising an exception. In the same spirit, a mutable value which can no longer be modified after initialization may be used in a functional style.

On the other hand, a program written in an imperative style still benefits from the advantages provided by Objective Caml: static type safety, automatic memory management, the exception mechanism, parametric polymorphism, and type inference.

The choice between the imperative and functional styles depends on the application to be developed. We may nevertheless suggest some guidelines based on the character of the application, and the criteria considered important in the development process.

- **choice of data structures**: The choice whether to use mutable data structures follows from the style of programming adopted. Indeed, the functional style is essentially incompatible with modifying mutable values. By contrast, constructing and traversing objects are the same whatever their status. This touches the same issue as "modification in place *vs* copying" on page 97; we return to it again in discussing criteria of efficiency.
- **required data structures**: If a program must modify mutable data structures, then the imperative style is the only one possible. If, on the other hand, you just have to traverse values, then adopting the functional style guarantees the integrity of the data.
  Using recursive data structures requires the use of functions that are themselves recursive. Recursive functions may be defined using either of the two styles, but it is often easier to understand the creation of a value following a recursive definition, which corresponds to a functional approach, than to repeat the recursive processing on this element. The functional style allows us to define generic iterators over the structure of data, which factors out the work of development and makes it faster.
- **criteria of efficiency**: Modification in place is far more efficient than creating a value. When code efficiency is the preponderant criterion, it will usually tip the balance in favor of the imperative style. We note however that the need to avoid sharing values may turn out to be a very hard task, and in the end costlier than copying the values to begin with.
  Being purely functional has a cost. Partial application and using functions passed as arguments from other functions has an execution cost greater than total application of a function whose declaration is visible. Using this eminently functional feature must thus be avoided in those portions of a program where efficiency is crucial.
- **development criteria**: the higher level of abstraction of functional programs permits them to be written more quickly, leading to code that is more compact and contains fewer errors than the equivalent imperative code, which is generally more verbose. The functional style is better suited to the constraints imposed by developing substantial applications. Since each function is not dependent upon its evaluation context, functional can be easily divided into small units that can be examined separately; as a consequence, the code is easier to read.
  Programs written using the functional style are more easily reusable because of its better modularity, and because functions may be passed as arguments to other functions.

These remarks show that it is often a good idea to mix the two programming styles within the same application. The functional programming style is faster to develop and confers a simpler organization to an application. However, portions whose execution time is critical repay being developed in a more efficient imperative style.

Mixing Styles

As we have mentioned, a language offering both functional and imperative characteristics allows the programmer to choose the more appropriate style for each part of the implementation of an algorithm. One can indeed use both aspects in the same function. This is what we will now illustrate.

Closures and Side Effects

The convention, when a function causes a side effect, is to treat it as a procedure and to return the value `()`, of type *unit*. Nevertheless, in some cases, it can be useful to cause the side effect within a function that returns a useful value. We have already used this mixture of the styles in the function `permute_pivot` of quicksort.

The next example is a symbol generator that creates a new symbol each time that it is called. It simply uses a counter that is incremented at every call.

```
# let c = ref 0;;
val c : int ref = {contents = 0}
# let reset_symb = function () → c:=0 ;;
val reset_symb : unit -> unit = <fun>
# let new_symb = function s →  c:=!c+1 ; s^(string_of_int !c) ;;
val new_symb : string -> string = <fun>
# new_symb "VAR" ;;
- : string = "VAR1"
# new_symb "VAR" ;;
- : string = "VAR2"
# reset_symb () ;;
- : unit = ()
# new_symb "WAR" ;;
- : string = "WAR1"
# new_symb "WAR" ;;
- : string = "WAR2"
```

The reference `c` may be hidden from the rest of the program by writing:

```
# let (reset_s , new_s) =
    let c = ref 0
    in let f1 () = c := 0
       and f2 s  = c := !c+1 ; s^(string_of_int !c)
    in (f1,f2) ;;
```

```
val reset_s : unit -> unit = <fun>
val new_s : string -> string = <fun>
```

This declaration creates a pair of functions that share the variable `c`, which is local to this declaration. Using these two functions produces the same behavior as the previous definitions.

```
# new_s "VAR";;
- : string = "VAR1"
# new_s "VAR";;
- : string = "VAR2"
# reset_s();;
- : unit = ()
# new_s "WAR";;
- : string = "WAR1"
# new_s "WAR";;
- : string = "WAR2"
```

This example permits us to illustrate the way that closures are represented. A closure may be considered as a pair containing the code (that is, the **function** part) as one component and the local envoronment containing the values of the free variables of the function. Figure 4.1 shows the memory representation of the closures `reset_s` and `new_s`.

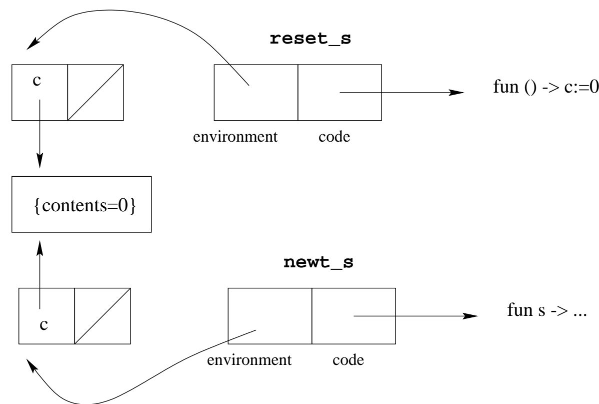


図 4.1: Memory representation of closures.

These two closures share the same environment, containing the value of `c`. When either one modifies the reference `c`, it modifies the contents of an area of memory that is shared with the other closure.

Physical Modifications and Exceptions

Exceptions make it possible to escape from situations in which the computation cannot proceed. In this case, an exception handler allows the calculation to continue, knowing that one branch has failed. The problem with side effects comes from the state of the modifiable data when the exception was raised. One cannot be sure of this state if there have been physical modifications in the branch of the calculation that has failed.

Let us define the increment function (++) analogous to the operator in C:

```
# let (++) x = x:=!x+1; x;;
val ( ++ ) : int ref -> int ref = <fun>
```

The following example shows a little computation where division by zero occurs together with

```
# let x = ref 2;;
val x : int ref = {contents = 2}
(* 1 *)
# !((++) x) * (1/0) ;;
Exception: Division_by_zero.
# x;;
- : int ref = {contents = 2}
(* 2 *)
# (1/0) * !((++) x) ;;
Exception: Division_by_zero.
# x;;
- : int ref = {contents = 3}
```

The variable x is not modified during the computation of the expression in (∗1∗), while it is modified in the computation of (∗2∗). Unless one saves the initial values, the form **try .. with ..** must not have a **with ..** part that depends on modifiable variables implicated in the expression that raised the exception.

Modifiable Functional Data Structures

In functional programming a program (in particular, a function expression) may also serve as a data object that may be manipulated, and one way to see this is to write association lists in the form of function expressions. In fact, one may view association lists of type *('a * 'b) list* as partial functions taking a key chosen from the set *'a* and returning a value in the set of associated values *'b*. Each association list is then a function of type *'a -> 'b*.

The empty list is the everywhere undefined function, which one simulates by raising an exception:

```
# let nil_assoc = function x → raise Not_found ;;
val nil_assoc : 'a -> 'b = <fun>
```

We next write the function add_assoc which adds an element to a list, meaning that it extends the function for a new entry:

```
# let add_assoc (k,v) l = function x → if x = k then v else l x ;;
val add_assoc : 'a * 'b -> ('a -> 'b) -> 'a -> 'b = <fun>
# let l = add_assoc ('1', 1) (add_assoc ('2', 2) nil_assoc) ;;
val l : char -> int = <fun>
# l '2' ;;
- : int = 2
# l 'x' ;;
Exception: Not_found.
```

We may now re-write the function mem_assoc:

```
# let mem_assoc k l = try  (l k) ; true  with  Not_found → false ;;
val mem_assoc : 'a -> ('a -> 'b) -> bool = <fun>
# mem_assoc '2' l ;;
- : bool = true
# mem_assoc 'x' l ;;
- : bool = false
```

By contrast, writing a function to remove an element from a list is not trivial, because one no longer has access to the values captured by the closures. To accomplish the same purpose we mask the former value by raising the exception Not_found.

```
# let rem_assoc k l = function x → if x=k then raise Not_found else l x ;;
val rem_assoc : 'a -> ('a -> 'b) -> 'a -> 'b = <fun>
# let l = rem_assoc '2' l ;;
val l : char -> int = <fun>
# l '2' ;;
Exception: Not_found.
```

Clearly, one may also create references and work by side effect on such values. However, one must take some care.

```
# let add_assoc_again (k,v) l =  l := (function x → if x=k then v else !l x) ;;
val add_assoc_again : 'a * 'b -> ('a -> 'b) ref -> unit = <fun>
```

The resulting value for l is a function that points at itself and therefore loops. This annoying side effect is due to the fact that the dereferencing `!l` is within the scope of the closure **function** *x* →. The value of `!l` is not evaluated during compilation, but at run-time. At that time, l points to the value that has already been modified by add_assoc. We must therefore correct our definition using the closure created by our original definition of add_assoc:

```
# let add_assoc_again (k, v) l =  l := add_assoc (k, v) !l ;;
val add_assoc_again : 'a * 'b -> ('a -> 'b) ref -> unit = <fun>
# let l = ref nil_assoc ;;
val l : ('_a -> '_b) ref = {contents = <fun>}
# add_assoc_again ('1',1) l ;;
- : unit = ()
```

```
# add_assoc_again ('2',2) l ;;
- : unit = ()
# !l '1' ;;
- : int = 1
# !l 'x' ;;
Exception: Not_found.
```

Lazy Modifiable Data Structures

Combining imperative characteristics with a functional language produces good tools for implementing computer languages. In this subsection, we will illustrate this idea by implementing data structures with deferred evaluation. A data structure of this kind is not completely evaluated. Its evaluation progresses according to the use made of it.

Deferred evaluation, which is often used in purely functional languages, is simulated using function values, possibly modifiable. There are at least two purposes for manipulating incompletely evaluated data structures: first, so as to calculate only what is effectively needed in the computation; and second, to be able to work with potentially infinite data structures.

We define the type `vm`, whose members contain either an already calculated value (constructor `Imm`) or else a value to be calculated (constructor `Deferred`):

```
# type 'a v =
      Imm of 'a
    | Deferred of (unit → 'a);;
# type 'a vm = {mutable c : 'a v };;
```

A computation is deferred by encapsulating it in a closure. The evaluation function for deferred values must return the value if it has already been calculated, and otherwise, if the value is not already calculated, it must evaluate it and then store the result.

```
# let eval e = match e.c with
    Imm a → a
  | Deferred f → let u = f () in e.c <- Imm u ; u ;;
val eval : 'a vm -> 'a = <fun>
```

The operations of deferring evaluation and activating it are also called **freezing** and **thawing** a value.

We could also write the conditional control structure in the form of a function:

```
# let if_deferred c e1 e2 =
    if eval c then eval e1 else eval e2;;
val if_deferred : bool vm -> 'a vm -> 'a vm -> 'a = <fun>
```

Here is how to use it in a recursive function such as factorial:

```
# let rec facr n =
    if_deferred
           {c=Deferred(fun () → n = 0)}
           {c=Deferred(fun () →  1)}
           {c=Deferred(fun () → n*(facr(n-1)))};;
val facr : int -> int = <fun>
# facr 5;;
- : int = 120
```

The classic form of **if** can not be written in the form of a function. In fact, if we define a function `if_function` this way:

```
# let if_function c e1 e2 = if c then e1 else e2;;
val if_function : bool -> 'a -> 'a -> 'a = <fun>
```

then the three arguments of `if_function` are evaluated at the time they are passed to the function. So the function `fact` loops, because the recursive call `fact(n-1)` is always evaluated, even when `n` has the value `0`.

```
# let rec fact n = if_function (n=0) 1 (n*fact(n-1)) ;;
val fact : int -> int = <fun>
# fact 5 ;;
Stack overflow during evaluation (looping recursion?).
```

Module Lazy

The implementation difficulty for frozen values is due to the conflict between the eager evaluation strategy of Objective Caml and the need to leave expressions unevaluated. Our attempt to redefine the conditional illustrated this. More generally, it is impossible to write a function that freezes a value in producing an object of type *vm*:

```
# let freeze e = { c = Deferred (fun () → e) };;
val freeze : 'a -> 'a vm = <fun>
```

When this function is applied to arguments, the Objective Caml evaluation strategy evaluates the expression `e` passed as argument before constructing the closure **fun** () → *e*. The next example shows this:

```
# freeze (print_string "trace"; print_newline(); 4*5);;
trace
- : int vm = {c = Deferred <fun>}
```

This is why the following syntactic form was introduced.

構文 : **lazy** *expr*

警告 This form is a language extension that may evolve in future versions.

When the keyword **lazy** is applied to an expression, it constructs a value of a type declared in the module `Lazy`:

```
# let x = lazy (print_string "Hello"; 3*4) ;;
val x : int lazy_t = <lazy>
```

The expression `(print_string "Hello")` has not been evaluated, because no message has been printed. The function `force` of module `Lazy` allows one to force evaluation:

```
# Lazy.force x ;;
Hello- : int = 12
```

Now the value `x` has altered:

```
# x ;;
- : int lazy_t = lazy 12
```

It has become the value of the expression that had been frozen, namely `12`.

For another call to the function `force`, it's enough to return the value already calculated:

```
# Lazy.force x ;;
- : int = 12
```

The string `"Hello"` is no longer prefixed.

"Infinite" Data Structures

The second reason to defer evaluation is to be able to construct potentially infinite data structures such as the set of natural numbers. Because it might take a long time to construct them all, the idea here is to compute only the first one and to know how to pass to the next element.

We define a generic data structure *'a enum* which will allow us to enumerate the elements of a set.

```
# type 'a enum = { mutable i : 'a; f :'a → 'a } ;;
type 'a enum = { mutable i : 'a; f : 'a -> 'a; }
# let next e = let x = e.i in e.i <- (e.f e.i) ; x ;;
val next : 'a enum -> 'a = <fun>
```

Now we can get the set of natural numbers by instantiating the fields of this structure:

```
# let nat = { i=0; f=fun x → x + 1 };;
val nat : int enum = {i = 0; f = <fun>}
# next nat;;
- : int = 0
# next nat;;
- : int = 1
# next nat;;
- : int = 2
```

Another example gives the elements of the Fibonnacci sequence, which has the defini-

tion:

$$\begin{cases} u_0 = 1 \\ u_1 = 1 \\ u_{n+2} = u_n + u_{n+1} \end{cases}$$

The function to compute the successor must take account of the current value, (u_{n-1}), but also of the preceding one (u_{n-2}). For this, we use the state `c` in the following closure:

```
# let fib = let fx = let c = ref 0 in fun v → let r = !c + v in c:=v ; r
            in { i=1 ; f=fx } ;;
val fib : int enum = {i = 1; f = <fun>}
# for i=0 to 10 do  print_int (next fib); print_string "  " done ;;
1  1  2  3  5  8  13  21  34  55  89  - : unit = ()
```

Streams of Data

Streams are (potentially infinite) sequences containing elements of the same kind. The evaluation of a part of a stream is done on demand, whenever it is needed by the current computation. A stream is therefore a **lazy** data structure.

The *stream* type is an abstract data type; one does not need to know how it is implemented. We manipulate objects of this type using constructor functions and destructor (or selector) functions. For the convenience of the user, Objective Caml has simple syntactic constructs to construct streams and to access their elements.

警告 Streams are an extension of the language, not part of the stable core of Objective Caml.

Construction

The syntactic sugar to construct streams is inspired by that for lists and arrays. The empty stream is written:

```
# [< >] ;;
Characters 1-3:
  [< >] ;;
  ^^
Syntax error
```

One may construct a stream by enumerating its elements, preceding each one with an with a single quote (character '):

```
# [< '0; '2; '4 >] ;;
Characters 1-3:
  [< '0; '2; '4 >] ;;
  ^^
```

```
Syntax error
```

Expressions not preceded by an apostrophe are considered to be sub-streams:

```
# [< '0; [< '1; '2; '3 >]; '4 >] ;;
Characters 1-3:
  [< '0; [< '1; '2; '3 >]; '4 >] ;;
  ^^
Syntax error
# let s1 = [< '1; '2; '3 >] in [< s1; '4 >] ;;
Characters 9-11:
  let s1 = [< '1; '2; '3 >] in [< s1; '4 >] ;;
           ^^
Syntax error
# let concat_stream a b = [< a ; b >] ;;
Characters 24-26:
  let concat_stream a b = [< a ; b >] ;;
                          ^^
Syntax error
# concat_stream [< '"if"; '"c";'"then";'"1" >] [< '"else";'"2" >] ;;
Characters 14-16:
  concat_stream [< '"if"; '"c";'"then";'"1" >] [< '"else";'"2" >] ;;
                ^^
Syntax error
```

The Stream module also provides other construction functions. For instance, the functions of_channel and of_string return a stream containing a sequence of characters, received from an input stream or a string.

```
# Stream.of_channel ;;
- : in_channel -> char Stream.t = <fun>
# Stream.of_string ;;
- : string -> char Stream.t = <fun>
```

The deferred computation of streams makes it possible to manipulate infinite data structures in a way similar to the type *'a enum* defined on page 107. We define the stream of natural numbers by its first element and a function calculating the stream of elements to follow:

```
# let rec nat_stream n = [< 'n ; nat_stream (n+1) >] ;;
Characters 24-26:
  let rec nat_stream n = [< 'n ; nat_stream (n+1) >] ;;
                         ^^
Syntax error
# let nat = nat_stream 0 ;;
Characters 10-20:
  let nat = nat_stream 0 ;;
            ^^^^^^^^^^
```

```
Unbound value nat_stream
```

Destruction and Matching of Streams

The primitive `next` permits us to evaluate, retrieve, and remove the first element of a stream, all at once:

```
# for i=0 to 10 do
     print_int (Stream.next nat) ;
     print_string "  "
     done ;;
Characters 44-47:
    print_int (Stream.next nat) ;
                           ^^^
This expression has type int enum but is here used with type 'a Stream.t
# Stream.next nat ;;
Characters 12-15:
  Stream.next nat ;;
              ^^^
This expression has type int enum but is here used with type 'a Stream.t
```

When the stream is exhausted, an exception is raised.

```
# Stream.next [< >] ;;
Characters 13-15:
  Stream.next [< >] ;;
              ^^
Syntax error
```

To manipulate streams, Objective Caml offers a special-purpose matching construct called **destructive matching**. The value matched is calculated and removed from the stream. There is no notion of exhaustive match for streams, and, since the data type is lazy and potentially infinite, one may match less than the whole stream. The syntax for matching is:

構文 : **match** *expr* **with parser [< '**p_1 ...**>] ->** $expr_1$ **|** ...

The function `next` could be written:

```
# let next s = match s with parser [< 'x >] → x ;;
Characters 34-36:
  let next s = match s with parser [< 'x >] -> x ;;
                                    ^^
Syntax error
# next nat;;
- : int = 3
```

Note that the enumeration of natural numbers picks up where we left it previously.

As with function abstraction, there is a syntactic form matching a function parameter of type *Stream.t*.

構文 : `parser p -¿ ...`

The function `next` can thus be rewritten:

```
# let next = parser [<'x>] → x ;;
Characters 19-21:
  let next = parser [<'x>] -> x ;;
                    ^^
Syntax error
# next nat ;;
- : int = 4
```

It is possible to match the empty stream, but take care: the stream pattern `[<>]` matches every stream. In fact, a stream `s` is always equal to the stream `[< [<>]; s >]`. For this reason, one must reverse the usual order of matching:

```
# let rec it_stream f s =
   match s with parser
     [< 'x ; ss >] → f x ; it_stream f ss
   | [<>] → () ;;
Characters 49-51:
     [< 'x ; ss >] -> f x ; it_stream f ss
     ^^
Syntax error
# let print_int1 n = print_int n ; print_string" " ;;
val print_int1 : int -> unit = <fun>
# it_stream print_int1 [<'1; '2; '3>] ;;
Characters 21-23:
  it_stream print_int1 [<'1; '2; '3>] ;;
                       ^^
Syntax error
```

Since matching is destructive, one can equivalently write:

```
# let rec it_stream f s =
   match s with parser
     [< 'x >] → f x ; it_stream f s
   | [<>] → () ;;
Characters 49-51:
     [< 'x >] -> f x ; it_stream f s
     ^^
Syntax error
# it_stream print_int1 [<'1; '2; '3>] ;;
Characters 21-23:
  it_stream print_int1 [<'1; '2; '3>] ;;
                       ^^
Syntax error
```

Although streams are lazy, they want to be helpful, and never refuse to furnish a first element; when it has been supplied once it is lost. This has consequences for matching.

The following function is an attempt (destined to fail) to display pairs from a stream of integers, except possibly for the last element.

```
# let print_int2 n1
   n2 =
    print_string "(" ; print_int n1 ; print_string "," ;
    print_int n2 ; print_string ")" ;;
val print_int2 : int -> int -> unit = <fun>
# let rec print_stream s =
    match s with parser
      [< 'x; 'y >] → print_int2 x y; print_stream s
    | [< 'z >] → print_int1 z; print_stream s
    | [<>] → print_newline() ;;
Characters 49-51:
      [< 'x; 'y >] -> print_int2 x y; print_stream s
      ^^
Syntax error
# print_stream [<'1; '2; '3>];;
Characters 13-15:
  print_stream [<'1; '2; '3>];;
               ^^
Syntax error
```

The first two two members of the stream were displayed properly, but during the evaluation of the recursive call (`print_stream [<3>]`), the first pattern found a value for `x`, which was thereby consumed. There remained nothing more for `y`. This was what caused the error. In fact, the second pattern is useless, because if the stream is not empty, then first pattern always begins evaluation.

To obtain the desired result, we must sequentialize the matching:

```
# let rec print_stream s =
    match s with parser
      [< 'x >]
       → (match s with parser
            [< 'y >] → print_int2 x y; print_stream s
          | [<>] → print_int1 x; print_stream s)
    | [<>] → print_newline() ;;
Characters 50-52:
      [< 'x >]
      ^^
Syntax error
# print_stream [<'1; '2; '3>];;
Characters 13-15:
  print_stream [<'1; '2; '3>];;
               ^^
Syntax error
```

If matching fails on the first element of a pattern however, then we again have the familiar behavior of matching:

```
# let rec print_stream s =
    match s with parser
      [< '1; 'y >] → print_int2 1 y; print_stream s
    | [< 'z >] → print_int1 z; print_stream s
    | [<>] → print_newline() ;;
Characters 50-52:
     [< '1; 'y >] -> print_int2 1 y; print_stream s
     ^^
Syntax error
# print_stream [<'1; '2; '3>] ;;
Characters 13-15:
  print_stream [<'1; '2; '3>] ;;
               ^^
Syntax error
```

The Limits of Matching

Because it is destructive, matching streams differs from matching on sum types. We will now illustrate how radically different it can be.

We can quite naturally write a function to compute the sum of the elements of a stream:

```
# let rec sum s =
    match s with parser
      [< 'n; ss >] → n+(sum ss)
    | [<>] → 0 ;;
Characters 41-43:
     [< 'n; ss >] -> n+(sum ss)
     ^^
Syntax error
# sum [<'1; '2; '3; '4>] ;;
Characters 4-6:
  sum [<'1; '2; '3; '4>] ;;
      ^^
Syntax error
```

However, we can just as easily consume the stream from the inside, naming the partial result:

```
# let rec sum s =
    match s with parser
      [< 'n; r = sum >] → n+r
    | [<>] → 0 ;;
Characters 41-43:
     [< 'n; r = sum >] -> n+r
     ^^
Syntax error
```

```
# sum [<'1; '2; '3; '4>] ;;
Characters 4-6:
  sum [<'1; '2; '3; '4>] ;;
      ^^
Syntax error
```

We will examine some other important uses of streams in chapter 11, which is devoted to lexical and syntactic analysis. In particular, we will see how consuming a stream from the inside may be profitably used.

Exercises

Binary Trees

We represent binary trees in the form of vectors. If a tree a has height h, then the length of the vector will be $2^{(h+1)} - 1$. If a node has position i, then the left subtree of this node lies in the interval of indices $[i+1 \ , \ i+1+2^h]$, and its right subtree lies in the interval $[i+1+2^h+1 \ , \ 2^{(h+1)}-1]$. This representation is useful when the tree is almost completely filled. The type *'a* of labels for nodes in the tree is assumed to contain a special value indicating that the node does not exist. Thus, we represent labeled trees by the by vectors of type *'a array*.

1. Write a function , taking as input a binary tree of type *'a bin_tree* (defined on page 50) and an array (which one assumes to be large enough). The function stores the labels contained in the tree in the array, located according to the discipline described above.
2. Write a function to create a leaf (tree of height 0).
3. Write a function to construct a new tree from a label and two other trees.
4. Write a conversion function from the type *'a bin_tree* to an array.
5. Define an infix traversal function for these trees.
6. Use it to display the tree.
7. What can you say about prefix traversal of these trees?

Spelling Corrector

The exercise uses the lexical tree , from the exercise of chapter 2, page 62, to build a spelling corrector.

1. Construct a dictionary from a file in ASCII in which each line contains one word. For this, one will write a function which takes a file name as argument and returns the corresponding dictionary.

2. Write a function `words` that takes a character string and constructs the list of words in this string. The word separators are space, tab, apostrophe, and quotation marks.
3. Write a function `verify` that takes a dictionary and a list of words, and returns the list of words that do not occur in the dictionary.
4. Write a function `occurrences` that takes a list of words and returns a list of pairs associating each word with the number of its occurrences.
5. Write a function `spellcheck` that takes a dictionary and the name of a file containing the text to analyze. It should return the list of incorrect words, together with their number of occurrences.

Set of Prime Numbers

We would like now to construct the infinite set of prime numbers (without calculating it completely) using lazy data structures.

1. Define the predicate `divisible` which takes an integer and an initial list of prime numbers, and determines whether the number is divisible by one of the integers on the list.
2. Given an initial list of prime numbers, write the function `next` that returns the smallest number not on the list.
3. Define the value *`setprime`* representing the set of prime numbers, in the style of the type *`'a enum`* on page 107. It will be useful for this set to retain the integers already found to be prime.

Summary

This chapter has compared the functional and imperative programming styles. They differ mainly in the control of execution (implicit in functional and explicit in imperative programming), and in the representation in memory of data (sharing or explicitly copied in the imperative case, irrelevant in the functional case). The implementation of algorithms must take account of these differences. The choice between the two styles leads in fact to mixing them. This mixture allows us to clarify the representation of closures, to optimize crucial parts of applications, and to create mutable functional data. Physical modification of values in the environment of a closure permits us to better understand what a functional value is. The mixture of the two styles gives powerful implementation tools. We used them to construct potentially infinite values.

To Learn More

The principal consequences of adding imperative traits to a functional language are:

- To determine the evaluation strategy (strict evaluation);

- to add implementation constraints, especially for the GC (see Chapter 9);
- For statically typed languages, to make their type system more complex;
- To offer different styles of programming in the same language, permitting us to program in the style appropriate to the algorithm at hand, or possibly in a mixed style.

This last point is important in Objective Caml where we need the same parametric polymorphism for functions written in either style. For this, certain purely functional programs are no longer typable after the addition. Wright's article ([Wri95]) explains the difficulties of polymorphism in languages with imperative aspects. Objective Caml adopts the solution that he advocates. The classification of different kinds of polymorphism in the presence of physical modification is described well in the thesis of Emmanuel Engel ([Eng98]).

These consequences make the job of programming a bit harder, and learning the language a bit more difficult. But because the language is richer for this reason and above all offers the choice of style, the game is worth the candle. For example, strict evaluation is the rule, but it is possible to implement basic mechanisms for lazy evaluation, thanks to the mixture of the two styles. Most purely functional languages use a lazy evaluation style. Among languages close to ML, we would mention Miranda, LazyML, and Haskell. The first two are used at universities for teaching and research. By contrast, there are significant applications written in Haskell. The absence of controllable side effects necessitates an additional abstraction for input/output called **monads**. One can read works on Haskell (such as [Tho99]) to learn more about this subject. Streams are a good example of the mixture of functional and imperative styles. Their use in lexical and syntactic analysis is described in Chapter 11.

5

The Graphics Interface

This chapter presents the `Graphics` library, which is included in the distribution of the Objective Caml-language. This library is designed in such a way that it works identically under the main graphical interfaces of the most commonly used operating systems: Windows, MacOS, Unix with X-Windows. `Graphics` permits the realization of drawings which may contain text and images, and it handles basic events like mouse clicks or pressed keys.

The model of programming graphics applied is the "painter's model:" the last touch of color erases the preceding one. This is an imperative model where the graphics window is a table of points which is physically modified by each graphics primitive. The interactions with the mouse and the keyboard are a model of event-driven programming: the primary function of the program is an infinite loop waiting for user interaction. An event starts execution of a special handler, which then returns to the main loop to wait for the next event.

Although the `Graphics` library is very simple, it is sufficient for introducing basic concepts of graphical interfaces, and it also contains basic elements for developing graphical interfaces that are rich and easy to use by the programmer.

Chapter overview

The first section explains how to make use of this library on different systems. The second section introduces the basic notions of graphics programming: reference point, plotting, filling, colors, bitmaps. The third section illustrates these concepts by describing and implementing functions for creating and drawing "boxes." The fourth section demonstrates the animation of graphical objects and their interaction with the background of the screen or other animated objects. The fifth section presents event-driven programming, in other terms the skeleton of all graphical interfaces. Finally, the last

section uses the library `Graphics` to construct a graphical interface for a calculator (see page 84).

Using the `Graphics` *Module*

Utilization of the library `Graphics` differs depending on the system and the compilation mode used. We will not cover applications other than usable under the interactive toplevel of Objective Caml. Under the Windows and MacOS systems the interactive working environment already preloads this library. To make it available under Unix, it is necessary to create a new toplevel. This depends on the location of the X11 library. If this library is placed in one of the usual search paths for C language libraries, the command line is the following:

```
ocamlmktop -custom -o mytoplevel graphics.cma -cclib -lX11
```

It generates a new executable`mytoplevel` into which the library `Graphics` is integrated. Starting the executable works as follows:

```
./mytoplevel
```

If, however, as under Linux, the library X11 is placed in another directory, this has to be indicated to the command `ocamlmktop`:

```
ocamlmktop -custom -o mytoplevel graphics.cma -cclib \
           -L/usr/X11/lib -cclib -lX11
```

In this example, the file `libX11.a` is searched in the directory `/usr/X11/lib`.

A complete description of the command `ocamlmktop` can be found in chapter 7.

Basic notions

Graphics programming is tightly bound to the technological evolution of hardware, in particular to that of screens and graphics cards. In order to render images in sufficient quality, it is necessary that the drawing be refreshed (redrawn) at regular and short intervals, somewhat like in a cinema. There are basically two techniques for drawing on the screen: the first makes use of a list of visible segments where only the useful part of the drawing is drawn, the second displays all points of the screen (bitmap screen). It is the last technique which is used on ordinary computers.

Bitmap screens can be seen as rectangles of accessible, in other terms, displayable points. These points are called **pixels**, a word derived from *picture element*. They are the basic elements for constructing images. The height and width of the main bitmap

is the resolution of the screen. The size of this bitmap therefore depends on the size of each pixel. In monochrome (black/white) displays, a pixel can be encoded in one bit. For screens that allow gray scales or for color displays, the size of a pixel depends on the number of different colors and shades that a pixel may take. In a bitmap of 320x640 pixels with 256 colors per pixel, it is therefore necessary to encode a pixel in 8 bits, which requires video memory of: $480 * 640$ bytes $= 307200$ bytes $\simeq$ 300KB. This resolution is still used by certain MS-DOS programs.

The basic operations on bitmaps which one can find in the `Graphics` library are:

- coloration of pixels,
- drawing of pixels,
- drawing of forms: rectangles, ellipses,
- filling of closed forms: rectangles, ellipses, polygons,
- displaying text: as bitmap or as vector,
- manipulation or displacement of parts of the image.

All these operations take place at a **reference point**, the one of the bitmap. A certain number of characteristics of these graphical operations like the width of strokes, the joints of lines, the choice of the character font, the style and the motive of filling define what we call a **graphical context**. A graphical operation always happens in a particular graphical context, and its result depends on it. The graphical context of the `Graphics` library does not contain anything except for the current point, the current color, the current font and the size of the image.

Graphical display

The elements of the graphical display are: the reference point and the graphical context, the colors, the drawings, the filling pattern of closed forms, the texts and the bitmaps.

Reference point and graphical context

The `Graphics` library manages a unique main window. The coordinates of the reference point of the window range from point $(0,0)$ at the bottom left to the upper right corner of the window. The main functions on this window are:

- `open_graph`, of type *`string -> unit`*, which opens a window;
- `close_graph`, of type *`unit -> unit`*, which closes it;
- `clear_graph`, of type *`unit -> unit`*, which clears it.

The dimensions of the graphical window are given by the functions `size_x` and `size_y`.

The string argument of the function `open_graph` depends on the window system of the machine on which the program is executed and is therefore not platform independent. The empty string, however, opens a window with default settings. It is possible to

specify the size of the window: under X-Windows, `" 200x300"` yields a window which is 200 pixels wide and 300 pixels high. Beware, the space at the beginning of the string `" 200x300"` is required!

The graphical context contains a certain number of readable and/or modifiable parameters:

the current point:	`current_point` : *unit -> int * int*
	`moveto` : *int -> int -> unit*
the current color:	`set_color` : *color -> unit*
the width of lines:	`set_line_width` : *int -> unit*
the current character font:	`set_font` : *string -> unit*
the size of characters:	`set_text_size` : *int -> unit*

Colors

Colors are represented by three bytes: each stands for the intensity value of a main color in the RGB-model (red, green, blue), ranging from a minimum of 0 to a maximum of 255. The function `rgb` (of type *int -> int -> int -> color*) allows the generation of a new color from these three components. If the three components are identical, the resulting color is a gray which is more or less intense depending on the intensity value. Black corresponds to the minimum intensity of each component (`0 0 0`) and white is the maximum (`255 255 255`). Certain colors are predefined: `black`, `white`, `red`, `green`, `blue`, `yellow`, `cyan` and `magenta`.

The variables `foreground` and `background` correspond to the color of the fore- and the background respectively. Clearing the screen is equivalent to filling the screen with the background color.

A color (a value of type *color*) is in fact an integer which can be manipulated to, for example, decompose the color into its three components (`from_rgb`) or to apply a function to it that inverts it (`inv_color`).

```
(* color == R * 256 * 256  +  G * 256  +  B  *)
# let from_rgb (c : Graphics.color)  =
    let r = c / 65536  and  g = c / 256 mod 256  and  b = c mod 256
    in (r,g,b);;
val from_rgb : Graphics.color -> int * int * int = <fun>
# let inv_color (c : Graphics.color) =
     let (r,g,b) = from_rgb c
     in Graphics.rgb (255-r) (255-g) (255-b);;
val inv_color : Graphics.color -> Graphics.color = <fun>
```

The function `point_color`, of type *int -> int -> color*, returns the color of a point when given its coordinates.

Drawing and filling

A drawing function draws a line on the screen. The line is of the current width and color. A filling function fills a closed form with the current color. The various line- and filling functions are presented in figure 5.1.

drawing	filling	type
plot		*int -> int -> unit*
lineto		*int -> int -> unit*
	fill_rect	*int -> int -> int -> int -> unit*
	fill_poly	*(int * int) array -> unit*
draw_arc	fill_arc	*int -> int -> int -> int -> int -> unit*
draw_ellipse	fill_ellipse	*int -> int -> int -> int -> unit*
draw_circle	fill_circle	*int -> int -> int -> unit*

図 5.1: Drawing- and filling functions.

Beware, the function `lineto` changes the position of the current point to make drawing of vertices more convenient.

Drawing polygons To give an example, we add drawing primitives which are not predefined. A polygon is described by a table of its vertices.

```
# let draw_rect x0 y0 w h =
    let (a,b) = Graphics.current_point()
    and x1 = x0+w and y1 = y0+h
    in
      Graphics.moveto x0 y0;
      Graphics.lineto x0 y1; Graphics.lineto x1 y1;
      Graphics.lineto x1 y0; Graphics.lineto x0 y0;
      Graphics.moveto a b;;
val draw_rect : int -> int -> int -> int -> unit = <fun>

# let draw_poly r =
    let (a,b) = Graphics.current_point () in
    let (x0,y0) = r.(0) in Graphics.moveto x0 y0;
      for i = 1 to (Array.length r)-1 do
        let (x,y) = r.(i) in Graphics.lineto x y
      done;
      Graphics.lineto x0 y0;
      Graphics.moveto a b;;
val draw_poly : (int * int) array -> unit = <fun>
```

Please note that these functions take the same arguments as the predefined ones for filling forms. Like the other functions for drawing forms, they do not change the current point.

Illustrations in the painter's model This example generates an illustration of a token ring network (figure 5.2). Each machine is represented by a small circle. We place the set of machines on a big circle and draw a line between the connected machines. The current position of the token in the network is indicated by a small black disk.

The function `net_points` generates the coordinates of the machines in the network. The resulting data is stored in a table.

```
# let pi = 3.1415927;;
val pi : float = 3.1415927
# let net_points (x,y) l n =
    let a = 2. *. pi /. (float n) in
    let rec aux (xa,ya) i =
      if i > n then []
      else
        let na = (float i) *. a in
        let x1 = xa + (int_of_float ( cos(na) *. l))
        and y1 = ya + (int_of_float ( sin(na) *. l)) in
        let np = (x1,y1) in
          np :: (aux np (i+1))
    in Array.of_list (aux (x,y) 1);;
val net_points : int * int -> float -> int -> (int * int) array = <fun>
```

The function `draw_net` displays the connections, the machines and the token.

```
# let draw_net (x,y) l n sc st =
   let r = net_points (x,y) l n in
     draw_poly r;
     let draw_machine (x,y) =
       Graphics.set_color Graphics.background;
       Graphics.fill_circle x y sc;
       Graphics.set_color Graphics.foreground;
       Graphics.draw_circle x y sc
     in
       Array.iter draw_machine r;
       Graphics.fill_circle x y st;;
val draw_net : int * int -> float -> int -> int -> int -> unit = <fun>
```

The following function call corresponds to the left drawing in figure 5.2.

```
# draw_net (140,20) 60.0 10 10 3;;
Exception: Graphics.Graphic_failure "graphic screen not opened".

# save_screen "IMAGES/tokenring.caa";;
```

```
- : unit = ()
```

We note that the order of drawing objects is important. We first plot the connections

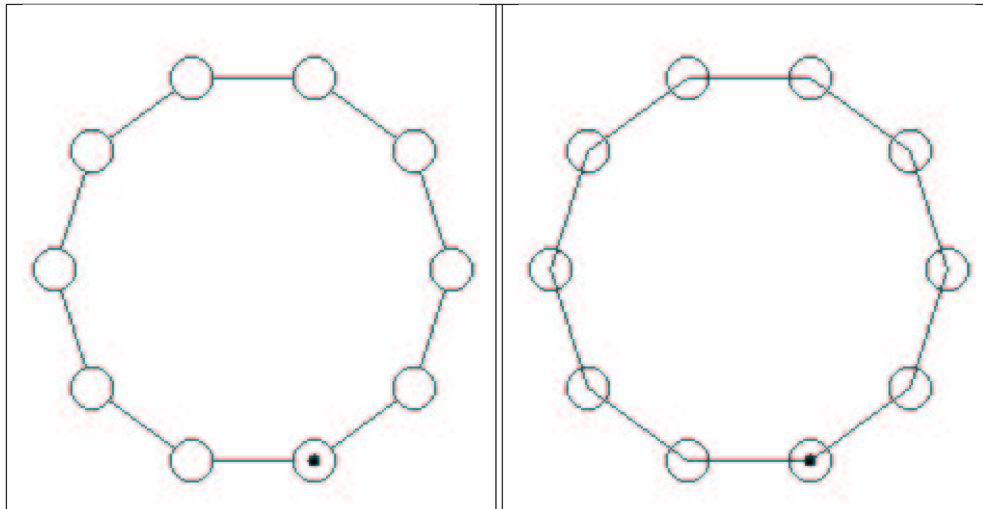


図 5.2: Tokenring network.

then the nodes. The drawing of network nodes erases some part of the connecting lines. Therefore, there is no need to calculate the point of intersection between the connection segments and the circles of the vertices. The right illustration of figure 5.2 inverts the order in which the objects are displayed. We see that the segments appear inside of the circles representing the nodes.

Text

The functions for displaying texts are rather simple. The two functions `draw_char` (of type *char -> unit*) and `draw_string` (of type *string -> unit*) display a character and a character string respectively at the current point. After displaying, the latter is modified. These functions do not change the current font and its current size.

注意

The displaying of strings may differ depending on the graphical interface.

The function `text_size` takes a string as input and returns a pair of integers that correspond to the dimensions of this string when it is displayed in the current font and size.

Displaying strings vertically This example describes the function `draw_string_v`, which displays a character string vertically at the current point. It is used in figure 5.3. Each letter is displayed separately by changing the vertical coordinate.

```
# let draw_string_v  s =
```

```
    let (xi,yi) = Graphics.current_point()
    and l = String.length s
    and (_,h) = Graphics.text_size s
    in
       Graphics.draw_char s.[0];
       for i=1 to l-1 do
         let (_,b) = Graphics.current_point()
         in Graphics.moveto xi (b-h);
            Graphics.draw_char s.[i]
        done;
      let (a,_) = Graphics.current_point() in Graphics.moveto a yi;;
val draw_string_v : string -> unit = <fun>
```

This function modifies the current point. After displaying, the point is placed at the initial position offset by the width of one character.

The following program permits displaying a legend around the axes (figure 5.3)

```
#
  Graphics.moveto 0 150; Graphics.lineto 300 150;
  Graphics.moveto 2 130; Graphics.draw_string "abscissa";
  Graphics.moveto 150 0; Graphics.lineto 150 300;
  Graphics.moveto 135 280; draw_string_v "ordinate";;
Exception: Graphics.Graphic_failure "graphic screen not opened".
```

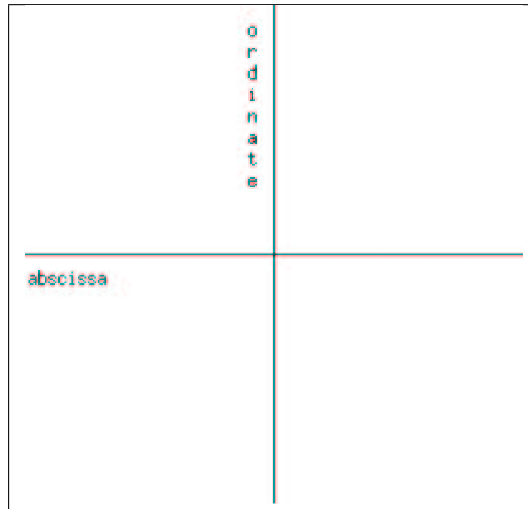


図 5.3: Legend around axes.

If we wish to realize vertical displaying of text, it is necessary to account for the fact that the current point is modified by the function `draw_string_v`. To do this, we define the function `draw_text_v`, which accepts the spacing between columns and a list of words as parameters.

```
# let draw_text_v n l =
    let f s = let (a,b) = Graphics.current_point()
              in draw_string_v s;
                 Graphics.moveto (a+n) b
    in List.iter f l;;
val draw_text_v : int -> string list -> unit = <fun>
```

If we need further text transformations like, for example, rotation, we will have to take the *bitmap* of each letter and perform the rotation on this set of pixels.

Bitmaps

A bitmap may be represented by either a color matrix (*color array array*) or a value of abstract type [1] *image*, which is declared in library `Graphics`. The names and types of the functions for manipulating bitmaps are given in figure 5.4.

function	type
`make_image`	*color array array -> image*
`dump_image`	*image -> color array array*
`draw_image`	*image -> int -> int -> unit*
`get_image`	*int -> int -> int -> int -> image*
`blit_image`	*image -> int -> int -> unit*
`create_image`	*int -> int -> image*

図 5.4: Functions for manipulating bitmaps.

The functions `make_image` and `dump_image` are conversion functions between types *image* and *color array array*. The function `draw_image` displays a bitmap starting at the coordinates of its bottom left corner.

The other way round, one can capture a rectangular part of the screen to create an image using the function `get_image` and by indicating the bottom left corner and the upper right one of the area to be captured. The function `blit_image` modifies its first parameter (of type *image*) and captures the region of the screen where the lower left corner is given by the point passed as parameter. The size of the captured region is the one of the image argument. The function `create_image` allows initializing images by specifying their size to use them with `blit_image`.

The predefined color `transp` can be used to create transparent points in an image. This makes it possible to display an image within a rectangular area only; the transparent points do not modify the initial screen.

1. Abstract types hide the internal representation of their values. The declaration of such types will be presented in chapter 14.

Polarization of Jussieu This example inverts the color of points of a bitmap. To do this, we use the function for color inversion presented on page 120, applying it to each pixel of a bitmap.

```
# let inv_image i =
    let inv_vec = Array.map (fun c → inv_color c) in
    let inv_mat = Array.map inv_vec in
    let inverted_matrix = inv_mat (Graphics.dump_image i) in
    Graphics.make_image inverted_matrix;;
val inv_image : Graphics.image -> Graphics.image = <fun>
```

Given the bitmap `jussieu`, which is displayed in the left half of figure 5.5, we use the function `inv_image` and obtain a new "solarized" bitmap, which is displayed in the right half of the same figure.

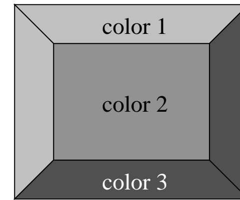
```
# let f_jussieu2 () = inv_image jussieu1;;
Characters 32-40:
  let f_jussieu2 () = inv_image jussieu1;;
                                ^^^^^^^^
Unbound value jussieu1
```

図 5.5: Inversion of Jussieu.

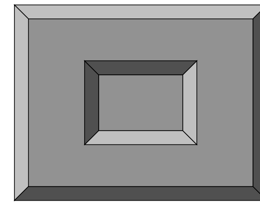
Example: drawing of boxes with relief patterns

In this example we will define a few utility functions for drawing boxes that carry relief patterns. A box is a generic object that is useful in many cases. It is inscribed in a rectangle which is characterized by a point of origin, a height and a width.

To give an impression of a box with a relief pattern, it is sufficient to surround it with two trapezoids in a light color and two others in a somewhat darker shade.



Inverting the colors, one can give the impression that the boxes are on top or at the bottom.



Implementation We add the border width, the display mode (top, bottom, flat) and the colors of its edges and of its bottom. This information is collected in a record.

```
# type relief = Top | Bot | Flat;;
# type box_config =
    { x:int; y:int; w:int; h:int; bw:int; mutable r:relief;
      b1_col : Graphics.color;
      b2_col : Graphics.color;
      b_col : Graphics.color};;
```

Only field `r` can be modified. We use the function `draw_rect` defined at page 121, which draws a rectangle.

For convenience, we define a function for drawing the outline of a box.

```
# let draw_box_outline bcf col =
    Graphics.set_color col;
    draw_rect bcf.x bcf.y bcf.w  bcf.h;;
val draw_box_outline : box_config -> Graphics.color -> unit = <fun>
```

The function of displaying a box consists of three parts: drawing the first edge, drawing the second edge and drawing the interior of the box.

```
# let draw_box bcf =
     let x1 = bcf.x and y1 = bcf.y in
     let x2 = x1+bcf.w and y2 = y1+bcf.h in
     let ix1 = x1+bcf.bw and ix2 = x2-bcf.bw
     and iy1 = y1+bcf.bw and iy2 = y2-bcf.bw in
     let border1 g =
       Graphics.set_color g;
       Graphics.fill_poly
         [| (x1,y1);(ix1,iy1);(ix2,iy1);(ix2,iy2);(x2,y2);(x2,y1) |]
```

```
    in
    let border2 g =
      Graphics.set_color g;
      Graphics.fill_poly
        [| (x1,y1);(ix1,iy1);(ix1,iy2);(ix2,iy2);(x2,y2);(x1,y2) |]
    in
    Graphics.set_color bcf.b_col;
    ( match bcf.r with
          Top  →
            Graphics.fill_rect ix1 iy1 (ix2-ix1) (iy2-iy1);
            border1 bcf.b1_col;
            border2 bcf.b2_col
        | Bot  →
            Graphics.fill_rect ix1 iy1 (ix2-ix1) (iy2-iy1);
            border1 bcf.b2_col;
            border2 bcf.b1_col
        | Flat →
            Graphics.fill_rect x1 y1 bcf.w bcf.h );
    draw_box_outline bcf Graphics.black;;
val draw_box : box_config -> unit = <fun>
```

The outline of boxes is highlighted in black. Erasing a box fills the area it covers with the background color.

```
# let erase_box bcf =
     Graphics.set_color bcf.b_col;
     Graphics.fill_rect (bcf.x+bcf.bw) (bcf.y+bcf.bw)
                        (bcf.w-(2*bcf.bw)) (bcf.h-(2*bcf.bw));;
val erase_box : box_config -> unit = <fun>
```

Finally, we define a function for displaying a character string at the left, right or in the middle of the box. We use the type *position* to describe the placement of the string.

```
# type position = Left | Center | Right;;
type position = Left | Center | Right
# let draw_string_in_box pos str bcf col =
      let (w, h) = Graphics.text_size str in
      let ty = bcf.y + (bcf.h-h)/2 in
      ( match pos with
            Center → Graphics.moveto (bcf.x + (bcf.w-w)/2) ty
          | Right  → let tx = bcf.x + bcf.w - w - bcf.bw - 1 in
                     Graphics.moveto tx ty
          | Left   → let tx = bcf.x + bcf.bw + 1 in Graphics.moveto tx ty  );
      Graphics.set_color col;
      Graphics.draw_string str;;
val draw_string_in_box :
  position -> string -> box_config -> Graphics.color -> unit = <fun>
```

Example: drawing of a game We illustrate the use of boxes by displaying the position of a game of type "tic-tac-toe" as shown in figure 5.6. To simplify the creation of boxes, we predefine colors.

```
# let set_gray x =  (Graphics.rgb x x x);;
val set_gray : int -> Graphics.color = <fun>
# let gray1= set_gray 100 and gray2= set_gray 170 and gray3= set_gray 240;;
val gray1 : Graphics.color = 6579300
val gray2 : Graphics.color = 11184810
val gray3 : Graphics.color = 15790320
```

We define a function for creating a grid of boxes of same size.

```
# let rec create_grid nb_col n sep b  =
    if n < 0 then []
    else
      let px = n mod nb_col and py = n / nb_col in
      let nx = b.x +sep + px*(b.w+sep)
      and ny = b.y +sep + py*(b.h+sep) in
      let b1 = {b with x=nx; y=ny} in
      b1::(create_grid nb_col (n-1) sep b);;
val create_grid : int -> int -> int -> box_config -> box_config list = <fun>
```

And we create the vector of boxes:

```
# let vb =
    let b =  {x=0; y=0; w=20;h=20; bw=2;
              b1_col=gray1; b2_col=gray3; b_col=gray2; r=Top} in
    Array.of_list (create_grid 5 24 2  b);;
val vb : box_config array =
  [|{x = 90; y = 90; w = 20; h = 20; bw = 2; r = Top; b1_col = 6579300;
     b2_col = 15790320; b_col = 11184810};
    {x = 68; y = 90; w = 20; h = 20; bw = 2; r = Top; b1_col = 6579300;
     b2_col = 15790320; b_col = ...};
    ...|]
```

Figure 5.6 corresponds to the following function calls:

```
# Array.iter draw_box vb;
  draw_string_in_box Center "X" vb.(5) Graphics.black;
  draw_string_in_box Center "X" vb.(8) Graphics.black;
  draw_string_in_box Center "O" vb.(12) Graphics.yellow;
  draw_string_in_box Center "O" vb.(11) Graphics.yellow;;
Exception: Graphics.Graphic_failure "graphic screen not opened".
```

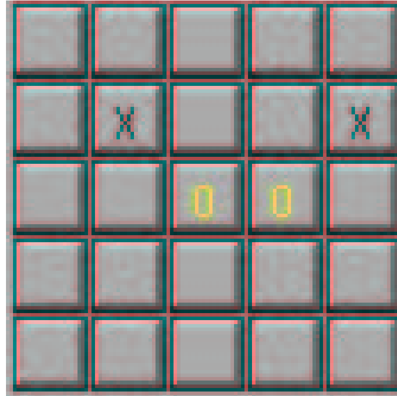


図 5.6: Displaying of boxes with text.

Animation

The animation of graphics on a screen reuses techniques of animated drawings. The major part of a drawing does not change, only the animated part must modify the color of its constituent pixels. One of the immediate problems we meet is the speed of animation. It can vary depending on the computational complexity and on the execution speed of the processor. Therefore, to be portable, an application containing animated graphics must take into account the speed of the processor. To get smooth rendering, it is advisable to display the animated object at the new position, followed by the erasure of the old one and taking special care with the intersection of the old and new regions.

Moving an object We simplify the problem of moving an object by choosing objects of a simple shape, namely rectangles. The remaining difficulty is knowing how to redisplay the background of the screen once the object has been moved.

We try to make a rectangle move around in a closed space. The object moves at a certain speed in directions X and Y. When it encounters a border of the graphical window, it bounces back depending on the angle of impact. We assume a situation without overlapping of the new and old positions of the object. The function `calc_pv` computes the new position and the new velocity from an old position `(x,y)`, the size of the object `(sx,sy)` and from the old speed `(dx,dy)`, taking into account the borders of the window.

```
# let calc_pv (x,y) (sx,sy) (dx,dy) =
    let nx1 = x+dx     and  ny1 = y + dy
    and nx2 = x+sx+dx  and  ny2 = y+sy+dy
    and ndx = ref dx   and  ndy = ref dy
    in
      ( if (nx1 < 0) || (nx2 >= Graphics.size_x()) then ndx := -dx );
```

```
      ( if (ny1 < 0) || (ny2 >= Graphics.size_y()) then ndy := -dy );
      ((x+ !ndx, y+ !ndy), (!ndx, !ndy));;
val calc_pv :
  int * int -> int * int -> int * int -> (int * int) * (int * int) = <fun>
```

The function `move_rect` moves the rectangle given by `pos` and `size` `n` times, the trajectory being indicated by its `speed` and by taking into account the borders of the space. The trace of movement which one can see in figure 5.7 is obtained by inversion of the corresponding bitmap of the displaced rectangle.

```
# let move_rect pos size speed n =
     let (x, y) = pos and (sx,sy) = size in
     let mem = ref (Graphics.get_image x y sx sy) in
     let rec move_aux x y speed n =
      if n = 0 then Graphics.moveto x y
      else
       let ((nx,ny),n_speed) = calc_pv (x,y) (sx,sy) speed
       and old_mem = !mem in
        mem := Graphics.get_image nx ny sx sy;
        Graphics.set_color Graphics.blue;
        Graphics.fill_rect nx ny sx sy;
        Graphics.draw_image (inv_image old_mem) x y;
        move_aux nx ny n_speed (n-1)
     in move_aux x y speed n;;
val move_rect : int * int -> int * int -> int * int -> int -> unit = <fun>
```

The following code corresponds to the drawings in figure 5.7. The first is obtained on a uniformly red background, the second by moving the rectangle across the image of Jussieu.

```
# let anim_rect () =
     Graphics.moveto 105 120;
     Graphics.set_color Graphics.white;
     Graphics.draw_string "Start";
     move_rect (140,120) (8,8) (8,4) 150;
     let (x,y) = Graphics.current_point() in
      Graphics.moveto (x+13) y;
     Graphics.set_color Graphics.white;
     Graphics.draw_string "End";;
val anim_rect : unit -> unit = <fun>
# anim_rect();;
Exception: Graphics.Graphic_failure "graphic screen not opened".
```

The problem was simplified, because there was no intersection between two successive positions of the moved object. If this is not the case, it is necessary to write a function that computes this intersection, which can be more or less complicated depending on

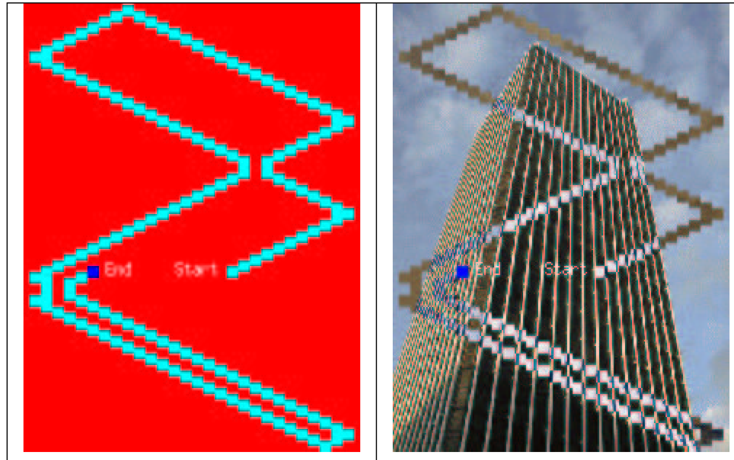


図 5.7: Moving an object.

the form of the object. In the case of a square, the intersection of two squares yields a rectangle. This intersection has to be removed.

Events

The handling of events produced in the graphical window allows interaction between the user and the program. `Graphics` supports the treating of events like keystrokes, mouse clicks and movements of the mouse.

The programming style therefore changes the organization of the program. It becomes an infinite loop waiting for events. After handling each newly triggered event, the program returns to the infinite loop except for events that indicate program termination.

Types and functions for events

The main function for waiting for events is `wait_next_event` of type *event list -> status*.

The different events are given by the sum type *event*.

```
type event = Button_down | Button_up | Key_pressed | Mouse_motion | Poll;;
```

The four main values correspond to pressing and to releasing a mouse button, to movement of the mouse and to keystrokes. Waiting for an event is a blocking operation except if the constructor `Poll` is passed in the event list. This function returns a value of type *status*:

```
type status =
  { mouse_x : int;
    mouse_y : int;
```

```
    button : bool;
    keypressed : bool;
    key : char};;
```

This is a record containing the position of the mouse, a Boolean which indicates whether a mouse button is being pressed, another Boolean for the keyboard and a character which corresponds to the pressed key. The following functions exploit the data contained in the event record:

- `mouse_pos`: *unit -> int * int*: returns the position of the mouse with respect to the window. If the mouse is placed elsewhere, the coordinates are outside the borders of the window.
- `button_down`: *unit -> bool*: indicates pressing of a mouse button.
- `read_key`: *unit -> char*: fetches a character typed on the keyboard; this operation blocks.
- `key_pressed`: *unit -> bool*: indicates whether a key is being pressed on the keyboard; this operation does not block.

The handling of events supported by `Graphics` is indeed minimal for developing interactive interfaces. Nevertheless, the code is portable across various graphical systems like Windows, MacOS or X-Windows. This is the reason why this library does not take into account different mouse buttons. In fact, the Mac does not even possess more than one. Other events, such as exposing a window or changing its size are not accessible and are left to the control of the library.

Program skeleton

All programs implementing a graphical user interface make use of a potentially infinite loop waiting for user interaction. As soon as an action arrives, the program executes the job associated with this action. The following function possesses five parameters of functionals. The first two serve for starting and closing the application. The next two arguments handle keyboard and mouse events. The last one permits handling of exceptions that escape out of the different functions of the application. We assume that the events associated with terminating the application raise the exception `End`.

```
# exception End;;
exception End
# let skel f_init f_end f_key f_mouse f_except =
   f_init ();
   try
       while true do
         try
           let s = Graphics.wait_next_event
                     [Graphics.Button_down; Graphics.Key_pressed]
           in if s.Graphics.keypressed then f_key s.Graphics.key
              else if s.Graphics.button
                   then f_mouse s.Graphics.mouse_x s.Graphics.mouse_y
```

```
         with
              End → raise End
            |  e  → f_except e
       done
   with
       End  → f_end ();;
val skel :
  (unit -> 'a) ->
  (unit -> unit) ->
  (char -> unit) -> (int -> int -> unit) -> (exn -> unit) -> unit = <fun>
```

Here, we use the skeleton to implement a mini-editor. Touching a key displays the typed character. A mouse click changes the current point. The character '&' exits the program. The only difficulty in this program is line breaking. We assume as simplification that the height of characters does not exceed twelve pixels.

```
# let next_line () =
    let (x,y) = Graphics.current_point()
    in if y>12 then Graphics.moveto 0 (y-12)
       else Graphics.moveto 0 y;;
val next_line : unit -> unit = <fun>
# let handle_char c = match c with
      '&'  → raise End
    | '\n' → next_line ()
    | '\r' → next_line ()
    |   _  → Graphics.draw_char c;;
val handle_char : char -> unit = <fun>
# let go () = skel
    (fun () → Graphics.clear_graph ();
              Graphics.moveto 0 (Graphics.size_y() -12) )
    (fun () → Graphics.clear_graph())
    handle_char
    (fun x y → Graphics.moveto x y)
    (fun e → ());;
val go : unit -> unit = <fun>
```

This program does not handle deletion of characters by pressing the key DEL.

Example: telecran

Telecran is a little drawing game for training coordination of movements. A point appears on a slate. This point can be moved in directions X and Y by using two control buttons for these axes without ever releasing the pencil. We try to simulate this behavior to illustrate the interaction between a program and a user. To do this we reuse the previously described skeleton. We will use certain keys of the keyboard to indicate movement along the axes.

We first define the type *state*, which is a record describing the size of the slate in terms of the number of positions in X and Y, the current position of the point and the scaling factor for visualization, the color of the trace, the background color and the color of the current point.

```
# type state = {maxx:int; maxy:int; mutable x : int; mutable y :int;
                scale:int;
                bc : Graphics.color;
                fc: Graphics.color; pc : Graphics.color};;
```

The function draw_point displays a point given its coordinates, the scaling factor and its color.

```
# let draw_point x y s c =
     Graphics.set_color c;
     Graphics.fill_rect (s*x) (s*y) s s;;
val draw_point : int -> int -> int -> Graphics.color -> unit = <fun>
```

All these functions for initialization, handling of user interaction and exiting the program receive a parameter corresponding to the state. The first four functions are defined as follows:

```
# let t_init s () =
     Graphics.open_graph (" " ^ (string_of_int (s.scale*s.maxx)) ^
       "x" ^ (string_of_int (s.scale*s.maxy)));
     Graphics.set_color s.bc;
     Graphics.fill_rect 0 0 (s.scale*s.maxx+1) (s.scale*s.maxy+1);
     draw_point s.x s.y s.scale s.pc;;
val t_init : state -> unit -> unit = <fun>
# let t_end s () =
     Graphics.close_graph();
     print_string "Good bye..."; print_newline();;
val t_end : 'a -> unit -> unit = <fun>
# let t_mouse s x y = ();;
val t_mouse : 'a -> 'b -> 'c -> unit = <fun>
# let t_except s ex = ();;
val t_except : 'a -> 'b -> unit = <fun>
```

The function t_init opens the graphical window and displays the current point, t_end closes this window and displays a message, t_mouse and t_except do not do anything. The program handles neither mouse events nor exceptions which may accidentally arise during program execution. The important function is the one for handling the keyboard t_key:

```
# let t_key s c =
    draw_point s.x s.y s.scale s.fc;
     (match c with
     '8' → if s.y < s.maxy then  s.y <- s.y + 1;
   | '2' → if s.y > 0 then s.y <- s.y - 1
```

```
  | '4' → if s.x > 0 then s.x <- s.x - 1
  | '6' → if s.x < s.maxx then s.x <- s.x + 1
  | 'c' →   Graphics.set_color s.bc;
            Graphics.fill_rect 0 0 (s.scale*s.maxx+1) (s.scale*s.maxy+1);
            Graphics.clear_graph()
  | 'e' → raise End
  | _ → ());
   draw_point s.x s.y s.scale s.pc;;
val t_key : state -> char -> unit = <fun>
```

It displays the current point in the color of the trace. Depending on the character passed, it modifies, if possible, the coordinates of the current point (characters: '2', '4', '6', '8'), clears the screen (character: 'c') or raises the exception `End` (character: 'e'), then it displays the new current point. Other characters are ignored. The choice of characters for moving the cursor comes from the layout of the numeric keyboard: the chosen keys correspond to the indicated digits and to the direction arrows. It is therefore useful to activate the numeric keyboard for the ergonomics of the program.

We finally define a state and apply the skeleton function in the following way:

```
# let stel = {maxx=120; maxy=120; x=60; y=60;
              scale=4; bc=Graphics.rgb 130 130 130;
              fc=Graphics.black; pc=Graphics.red};;
val stel : state =
  {maxx = 120; maxy = 120; x = 60; y = 60; scale = 4; bc = 8553090; fc = 0;
   pc = 16711680}
# let slate () =
     skel (t_init stel)  (t_end stel) (t_key stel)
      (t_mouse stel) (t_except stel);;
val slate : unit -> unit = <fun>
```

Calling function `slate` displays the graphical window, then it waits for user interaction on the keyboard. Figure 5.8 shows a drawing created with this program.

A Graphical Calculator

Let's consider the calculator example as described in the preceding chapter on imperative programming (see page 84). We will give it a graphical interface to make it more usable as a desktop calculator.

The graphical interface materializes the set of keys (digits and functions) and an area for displaying results. Keys can be activated using the graphical interface (and the mouse) or by typing on the keyboard. Figure 5.9 shows the interface we are about to construct.

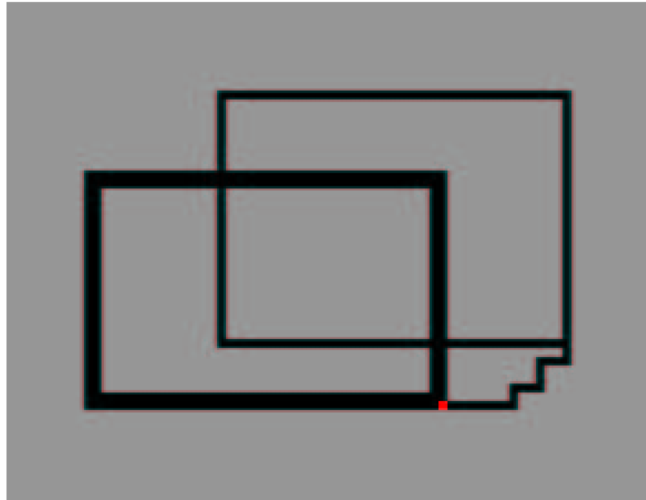


図 5.8: Telecran.

図 5.9: Graphical calculator.

We reuse the functions for drawing boxes as described on page 126. We define the following type:

```
# type calc_state =
    { s : state; k : (box_config * key * string ) list; v : box_config } ;;
```

It contains the state of the calculator, the list of boxes corresponding to the keys and the visualization box. We plan to construct a calculator that is easily modifiable. Therefore, we parameterize the construction of the interface with an association list:

```
# let descr_calc =
```

```
[  (Digit 0,"0");  (Digit 1,"1");  (Digit 2,"2");  (Equals, "=");
   (Digit 3,"3");  (Digit 4,"4");  (Digit 5,"5");  (Plus, "+");
   (Digit 6,"6");  (Digit 7,"7");  (Digit 8,"8");  (Minus, "-");
   (Digit 9,"9");  (Recall,"RCL"); (Div, "/");     (Times, "*");
   (Off,"AC");     (Store, "STO"); (Clear,"CE/C")
] ;;
```

Generation of key boxes At the beginning of this description we construct a list of key boxes. The function gen_boxes takes as parameters the description (descr), the number of the column (n), the separation between boxes (wsep), the separation between the text and the borders of the box (wsepint) and the size of the board (wbord). This function returns the list of key boxes as well as the visualization box. To calculate these placements, we define the auxiliary functions max_xy for calculating the maximal size of a list of complete pairs and max_lbox for calculating the maximal positions of a list of boxes.

```
# let gen_xy vals comp o =
    List.fold_left (fun  a (x,y) → comp (fst a) x,comp (snd a) y) o vals ;;
val gen_xy : ('a * 'a) list -> ('b -> 'a -> 'b) -> 'b * 'b -> 'b * 'b = <fun>
# let max_xy vals = gen_xy vals max (min_int,min_int);;
val max_xy : (int * int) list -> int * int = <fun>
# let max_boxl l =
    let bmax (mx,my) b = max mx b.x, max my b.y
    in List.fold_left bmax  (min_int,min_int) l ;;
val max_boxl : box_config list -> int * int = <fun>
```

Here is the principal function gen_boxes for creating the interface.

```
# let gen_boxes descr n wsep wsepint wbord  =
    let l_l = List.length descr in
    let nb_lig = if l_l mod n = 0 then l_l / n else l_l / n + 1 in
    let ls = List.map (fun (x,y) → Graphics.text_size y) descr in
    let sx,sy = max_xy  ls in
    let sx,sy= sx+wsepint ,sy+wsepint  in
      let r = ref [] in
        for i=0 to l_l-1 do
          let px = i mod n and py = i / n in
            let b = { x = wsep * (px+1) + (sx+2*wbord) * px  ;
                      y = wsep * (py+1) + (sy+2*wbord) * py ;
                      w = sx; h = sy ; bw = wbord;
                      r=Top;
                      b1_col = gray1;  b2_col = gray3; b_col =gray2}
            in r:= b::!r
        done;
        let mpx,mpy = max_boxl !r in
        let upx,upy = mpx+sx+wbord+wsep,mpy+sy+wbord+wsep in
        let (wa,ha) = Graphics.text_size "        0" in
```

```
          let v = { x=(upx-(wa+wsepint +wbord))/2 ; y= upy+ wsep;
                    w=wa+wsepint; h = ha +wsepint; bw = wbord *2; r=Flat ;
                    b1_col = gray1;  b2_col = gray3; b_col =Graphics.black}
          in
            upx,(upy+wsep+ha+wsepint+wsep+2*wbord),v,
            List.map2 (fun b (x,y) → b,x,y ) (List.rev !r) descr;;
val gen_boxes :
  ('a * string) list ->
  int ->
  int ->
  int -> int -> int * int * box_config * (box_config * 'a * string) list =
  <fun>
```

Interaction Since we would also like to reuse the skeleton proposed on page 133 for interaction, we define the functions for keyboard and mouse control, which are integrated in this skeleton. The function for controlling the keyboard is very simple. It passes the translation of a character value of type *key* to the function `transition` of the calculator and then displays the text associated with the calculator state.

```
# let f_key cs c =
     transition cs.s (translation c);
     erase_box cs.v;
     draw_string_in_box Right (string_of_int cs.s.vpr) cs.v Graphics.white ;;
val f_key : calc_state -> char -> unit = <fun>
```

The control of the mouse is a bit more complex. It requires verification that the position of the mouse click is actually in one of the key boxes. For this we first define the auxiliary function `mem`, which verifies membership of a position within a rectangle.

```
# let mem (x,y) (x0,y0,w,h) =
      (x >= x0) && (x< x0+w) && (y>=y0) && ( y<y0+h);;
val mem : int * int -> int * int * int * int -> bool = <fun>
# let f_mouse cs x y =
     try
       let b,t,s =
         List.find (fun (b,_,_) →
                      mem (x,y) (b.x+b.bw,b.y+b.bw,b.w,b.h)) cs.k
       in
         transition cs.s t;
         erase_box cs.v;
         draw_string_in_box Right (string_of_int cs.s.vpr ) cs.v Graphics.white
     with Not_found → ();;
val f_mouse : calc_state -> int -> int -> unit = <fun>
```

The function `f_mouse` looks whether the position of the mouse during the click is reallydwell within one of the boxes corresponding to a key. If it is, it passes the corresponding key to the transition function and displays the result, otherwise it will not do anything.

The function `f_exc` handles the exceptions which can arise during program execution.

```
# let f_exc cs ex =
    match ex with
     Division_by_zero ->
       transition cs.s Clear;
       erase_box cs.v;
       draw_string_in_box Right "Div 0" cs.v (Graphics.red)
   | Invalid_key -> ()
   | Key_off -> raise End
   | _  -> raise ex;;
val f_exc : calc_state -> exn -> unit = <fun>
```

In the case of a division by zero, it restarts in the initial state of the calculator and displays an error message on its screen. Invalid keys are simply ignored. Finally, the exception `Key_off` raises the exception `End` to terminate the loop of the skeleton.

Initialization and termination The initialization of the calculator requires calculation of the window size. The following function creates the graphical information of the boxes from a key/text association and returns the size of the principal window.

```
# let create_e k =
       Graphics.close_graph ();
       Graphics.open_graph " 10x10";
       let mx,my,v,lb = gen_boxes k 4 4 5 2 in
       let s = {lcd=0; lka = false; loa = Equals; vpr = 0; mem = 0}   in
         mx,my,{s=s; k=lb;v=v};;
val create_e : (key * string) list -> int * int * calc_state = <fun>
```

The initialization function makes use of the result of the preceding function.

```
# let f_init mx my cs () =
         Graphics.close_graph();
         Graphics.open_graph (" "^(string_of_int mx)^"x"^(string_of_int my));
         Graphics.set_color gray2;
         Graphics.fill_rect 0 0 (mx+1) (my+1);
         List.iter (fun (b,_,_) -> draw_box b) cs.k;
         List.iter
           (fun (b,_,s) -> draw_string_in_box Center s b Graphics.black) cs.k ;
         draw_box cs.v;
         erase_box cs.v;
         draw_string_in_box Right "hello" cs.v (Graphics.white);;
val f_init : int -> int -> calc_state -> unit -> unit = <fun>
```

Finally the termination function closes the graphical window.

```
# let f_end e () = Graphics.close_graph();;
val f_end : 'a -> unit -> unit = <fun>
```

The function go is parameterized by a description and starts the interactive loop.

```
# let go descr =
    let mx,my,e = create_e descr in
      skel (f_init mx my e) (f_end e) (f_key e) (f_mouse e) (f_exc e);;
val go : (key * string) list -> unit = <fun>
```

The call to `go descr_calc` corresponds to the figure 5.9.

Exercises

Polar coordinates

Coordinates as used in the library `Graphics` are Cartesian. There a line segment is represented by its starting point `(x0,y0)` and its end point `(x1,y1)`. It can be useful to use polar coordinates instead. Here a line segment is described by its point of origin `(x0,y0)`, a length (radius) `(r)` and an angle `(a)`. The relation between Cartesian and Polar coordinates is defined by the following equations:

$$\begin{cases} x_1 &= x_0 + r * \cos(a) \\ y_1 &= y_0 + r * \sin(a) \end{cases}$$

The following type defines the polar coordinates of a line segment:

```
# type seg_pol = {x:float; y:float; r:float; a:float};;
type seg_pol = { x : float; y : float; r : float; a : float; }
```

1. Write the function `to_cart` that converts polar coordinates to Cartesian ones.
2. Write the function `draw_seg` which displays a line segment defined by polar coordinates in the reference point of `Graphics`.
3. One of the motivations behind polar coordinates is to be able to easily apply transformations to line segments. A translation only modifies the point of origin, a rotation only affects the angle field and modifying the scale only changes the length field. Generally, one can represent a transformation as a triple of floats: the first represents the translation (we do not consider the case of translating the second point of the line segment here), the second the rotation and the third the scaling factor. Define the function `app_trans` which takes a line segment in polar coordinates and a triple of transformations and returns the new segment.
4. One can construct recursive drawings by iterating transformations. Write the function `draw_r` which takes as arguments a line segment `s`, a number of itera-

tions n, a list of transformations and displays all the segments resulting from the transformations on s iterated up to n.

5. Verify that the following program does produce the images in figure 5.10.

```
let pi = 3.1415927 ;;
let s = {x=100.; y= 0.; a= pi /. 2.; r = 100.} ;;
draw_r s 6 [ (-.pi/.2.),0.6,1.; (pi/.2.), 0.6,1.0] ;;
Graphics.clear_graph();;
draw_r s 6 [(-.pi /. 6.), 0.6, 0.766;
             (-.pi /. 4.), 0.55, 0.333;
             (pi /. 3.), 0.4, 0.5 ] ;;
```

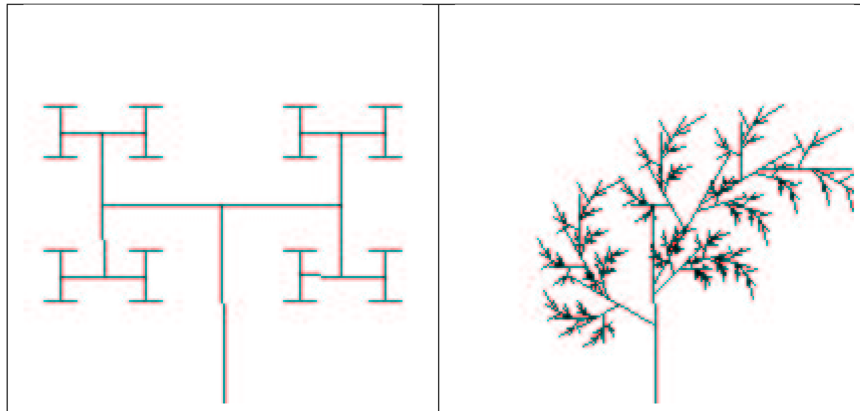


図 5.10: Recursive drawings.

Bitmap editor

We will attempt to write a small bitmap editor (similar to the command `bitmap` in X-window). For this we represent a bitmap by its dimensions (width and height), the pixel size and a two-dimensional table of booleans.

1. Define a type *bitmap_state* describing the information necessary for containing the values of the pixels, the size of the bitmap and the colors of displayed and erased points.
2. Write a function for creating bitmaps (`create_bitmap`) and for displaying bitmaps (`draw_bitmap`) .
3. Write the functions `read_bitmap` and `write_bitmap` which respectively read and write in a file passed as parameter following the ASCII format of X-window. If the file does not exist, the function for reading creates a new bitmap using the function `create_bitmap`. A displayed pixel is represented by the character `#`, the absence of a pixel by the character `-`. Each line of characters represents a line of the bitmap. One can test the program using the functions `atobm` and `bmtoa` of

X-window, which convert between this ASCII format and the format of bitmaps created by the command `bitmap`. Here is an example.

```
###################-------------#######---------######
###################---------------###-------------##--
###-----###-----###---------------###-------------#---
##------###------##----------------###-----------##---
#-------###-------#-----------------###---------##----
#-------###-------#-----------------###--------##-----
--------###--------------------------###-------#------
--------###-------###############-----###----##-------
--------###-------###---------###------###--##--------
--------###-------###----------##-------###-#---------
--------###-------###-----------#-------#####---------
--------###-------###-----------#--------###----------
--------###-------###--------------------####---------
--------###-------###--------------------####---------
--------###-------###------#-----------##---###-------
--------###-------###------#----------##----###-------
--------###-------##########----------#------###------
--------###-------##########---------##-------###-----
--------###-------###------#--------##--------###-----
--------###-------###------#-------##----------###----
--------###-------###--------------#------------###---
------#######-----###-----------#######--------#######
------------------###---------------------------------
------------------###-----------#---------------------
------------------###-----------#---------------------
------------------###----------##---------------------
------------------###---------###---------------------
------------------###############---------------------
```

4. We reuse the skeleton for interactive loops on page 133 to construct the graphical interface of the editor. The human-computer interface is very simple. The bitmap is permanently displayed in the graphical window. A mouse click in one of the slots of the bitmap inverts its color. This change is reflected on the screen. Pressing the key 'S' saves the bitmap in a file. The key 'Q' terminates the program.
   - Write a function `start` of type *`bitmap_state -> unit -> unit`* which opens a graphical window and displays the bitmap passed as parameter.
   - Write a function `stop` that closes the graphical window and exits the program.
   - Write a function `mouse` of type *`bitmap_state -> int -> int -> unit`* which modifies the pixel state corresponding to the mouse click and displays the change.
   - Write a function `key` of type *`string -> bitmap_state -> char -> unit`* which takes as arguments the name of a file, a bitmap and the char of the pressed key and executes the associated actions: saving to a file for the key 'S' and raising of the exception `End` for the key 'Q'.
5. Write a function `go` which takes the name of a file as parameter, loads the bitmap, displays it and starts the interactive loop.

Earth worm

The earth worm is a small, longish organism of a certain size which grows over time while eating objects in a world. The earth worm moves constantly in one direction. The only actions allowing a player to control it are changes in direction. The earth worm vanishes if it touches a border of the world or if it passes over a part of its body. It is most often represented by a vector of coordinates with two principal indices: its head and its tail. A move will therefore be computed from the new coordinates of its head, will display it and erase the tail. A growth step only modifies its head without affecting the tail of the earth worm.

1. Write the Objective Caml type or types for representing an earth worm and the world where it evolves. One can represent an earth worm by a queue of its coordinates.
2. Write a function for initialization and displaying an earth worm in a world.
3. Modify the function `skel` of the skeleton of the program which causes an action at each execution of the interactive loop, parameterized by a function. The treatment of keyboard events must not block.
4. Write a function `run` which advances the earth worm in the game. This function raises the exception `Victory` (if the worm reaches a certain size) and `Loss` if it hits a full slot or a border of the world.
5. Write a function for keyboard interaction which modifies the direction of the earth worm.
6. Write the other utility functions for handling interaction and pass them to the new skeleton of the program.
7. Write the initiating function which starts the application.

Summary

This chapter has presented the basic notions of graphics programming and event-driven programming using the `Graphics` library in the distribution of Objective Caml. After having explained the basic graphical elements (colors, drawing, filling, text and bitmaps) we have approached the problem of animating them. The mechanism of handling events in `Graphics` was then described in a way that allowed the introduction of a general method of handling user interaction. This was accomplished by taking a game as model for event-driven programming. To improve user interactions and to provide interactive graphical components to the programmer, we have developed a new library called `Awi`, which facilitates the construction of graphical interfaces. This library was used for writing the interface to the imperative calculator.

To learn more

Although graphics programming is naturally event-driven, the associated style of programming being imperative, it is not only possible but also often useful to introduce more functional operators to manipulate graphical objects. A good example comes from the use of the `MLgraph` library,

リンク: http://www.pps.jussieu.fr/~cousinea/MLgraph/mlgraph.html

which implements the graphical model of PostScript and proposes functional operators to manipulate images. It is described in [CC92, CS94] and used later in [CM98] for the optimized placement of trees to construct drawings in the style of Escher.

One interesting characteristic of the `Graphics` library is that it is portable to the graphical interfaces of Windows, MacOS and Unix. The notion of virtual bitmaps can

be found in several languages like Le_Lisp and more recently in Java. Unfortunately, the `Graphics` library in Objective Caml does not possess interactive components for the construction of interfaces. One of the applications described in part II of this book contains the first bricks of the `Awi` library. It is inspired by the *Abstract Windowing Toolkit* of the first versions of Java. One can perceive that it is relatively easy to extend the functionality of this library thanks to the existence of functional values in the language. Therefore chapter 16 compares the adaptation of object oriented programming and functional and modular programming for the construction of graphical interfaces. The example of `Awi` is functional and imperative, but it is also possible to only use the functional style. This is typically the case for purely functional languages. We cite the systems `Fran` and `Fudget` developed in Haskell and derivatives. The system `Fran` permits construction of interactive animations in 2D and 3D, which means with events between animated objects and the user.

リンク: http://www.research.microsoft.com/˜conal/fran/

The `Fudget` library is intended for the construction of graphical interfaces.

リンク: http://www.cs.chalmers.se/ComputingScience/Research/Functional/Fudgets/

One of the difficulties when one wants to program a graphical interface for ones application is to know which of the numerous existing libraries to choose. It is not sufficient to determine the language and the system to fix the choice of the tool. For Objective Caml there exist several more or less complete ones:

- the encapsulation of `libX`, for X-Windows;
- the `librt` library, also for X-Windows;
- `ocamltk`, an adaptation of `Tcl/Tk`, portable;
- `mlgtk`, an adaptation of `Gtk`, portable.

We find the links to these developments in the “Caml Hump”:

リンク: http://caml.inria.fr/humps/index.html

Finally, we have only discussed programming in 2D. The tendency is to add one dimension. Functional languages must also respond to this necessity, perhaps in the model of VRML or the Java 3D-extension. In purely functional languages the system `Fran` offers interesting possibilities of interaction between *sprites*. More closely to Objective Caml one can use the `VRcaML` library or the development environment SCOL.

The `VRcaML` library was developed in the manner of `MLgraph` and integrates a part of the graphical model of VRML in Objective Caml.

リンク: http://www.pps.jussieu.fr/˜emmanuel/Public/enseignement/VRcaML

One can therefore construct animated scenes in 3D. The result is a VRML-file that can be directly visualized.

Still in the line of Caml, the language SCOL is a functional communication language with important libraries for 2D and 3D manipulations, which is intended as environment for people with little knowledge in computer science.

リンク: http://www.cryo-networks.com

The interest in the language SCOL and its development environment is to be able to create distributed applications, e.g. client-server, thus facilitating the creation of Internet sites. We present distributed programming in Objective Caml in chapter 20.

6

Applications

The reason to prefer one programming language over another lies in the ease of developing and maintaining robust applications. Therefore, we conclude the first part of this book, which dealt with a general presentation of the Objective Caml language, by demonstrating its use in a number of applications.

The first application implements a few functions which are used to write database queries. We emphasize the use of list manipulations and the functional programming style. The user has access to a set of functions with which it is easy to write and run queries using the Objective Caml language directly. This application shows the programmer how he can easily provide the user with most of the query tools that the user should need.

The second application is an interpreter for a tiny BASIC[1]. This kind of imperative language fueled the success of the first microcomputers. Twenty years later, they seem to be very easy to design. Although BASIC is an imperative language, the implementation of the interpreter uses the functional features of Objective Caml, especially for the evaluation of commands. Nevertheless, the lexer and parser for the language use a mutable structure.

The third application is a one-player game, Minesweeper, which is fairly well-known since it is bundled with the standard installation of Windows systems. The goal of the game is to uncover a bunch of hidden mines by repeatedly uncovering a square, which then indicates the number of mines around itself. The implementation uses the imperative features of the language, since the data structure used is a two-dimensional array which is modified after each turn of the game. This application uses the `Graphics` module to draw the game board and to interact with the player. However, the automatic uncovering of some squares will be written in a more functional style.
This latter application uses functions from the `Graphics` module described in chapter

1. which means “Beginner’s All purpose Symbolic Instruction Code”.

5 (see page 117) as well as some functions from the `Random` and `Sys` modules (see chapter 8, pages 216 and 234).

Database queries

The implementation of a database, its interface, and its query language is a project far too ambitious for the scope of this book and for the Objective Caml knowledge of the reader at this point. However, restricting the problem and using the functional programming style at its best allows us to create an interesting tool for query processing. For instance, we show how to use iterators as well as partial application to formulate and execute queries. We also show the use of a data type encapsulating functional values.

For this application, we use as an example a database on the members of an association. It is presumed to be stored in the file `association.dat`.

Data format

Most database programs use a "proprietary" format to store the data they manipulate. However, it is usually possible to store the data as some text that has the following structure:

- the database is a list of *cards* separated by carriage-returns;
- each card is a list of *fields* separated by some given character, `':'` in our case;
- a field is a string which contains no carriage-return nor the character `':'`;
- the first card is the list of the names associated with the fields, separated by the character `'|'`.

The association data file starts with:

```
Num|Lastname|Firstname|Address|Tel|Email|Pref|Date|Amount
0:Chailloux:Emmanuel:Université P6:0144274427:ec@lip6.fr:email:25.12.1998:100.00
1:Manoury:Pascal:Laboratoire PPS::pm@lip6.fr:mail:03.03.1997:150.00
2:Pagano:Bruno:Cristal:0139633963::mail:25.12.1998:150.00
3:Baro:Sylvain::0144274427:baro@pps.fr:email:01.03.1999:50.00
```

The meaning of the fields is the following:

- `Num` is the member number;
- `Lastname`, `Firstname`, `Address`, `Tel`, and `Email` are obvious;
- `Pref` indicates the means by which the member wishes to be contacted: by mail (`mail`), by email (`email`), or by phone (`tel`);
- `Date` and `Amount` are the date and the amount of the last membership fee received, respectively.

We need to decide what represention the program should use internally for a database. We could use either a list of cards or an array of cards. On the one hand, a list has the nice property of being easily modified: adding and removing a card are simple operations. On the other hand, an array allows constant access time to any card. Since our goal is to work on all the cards and not on some of them, each query accesses all the cards. Thus a list is a good choice. The same issue arises concerning the cards themselves: should they be lists or arrays of strings? This time an array is a good choice, since the format of a card is fixed for the whole database. It not possible to add a new field. Since a query might access only a few fields, it is important for this access to be fast.

The most natural solution for a card would be to use an array indexed by the names of the fields. Since such a type is not available in Objective Caml, we can use an array (indexed by integers) and a function associating a field name with the array index corresponding to the field.

```
# type data_card = string array ;;
# type data_base = { card_index : string → int ; data : data_card list } ;;
```

Access to the field named `n` of a card `dc` of the database `db` is implemented by the function:

```
# let field db n (dc : data_card) = dc.(db.card_index n) ;;
val field : data_base -> string -> data_card -> string = <fun>
```

The type of `dc` has been set to *data_card* to constrain the function `field` to only accept string arrays and not arrays of other types.

Here is a small example:

```
# let base_ex =
    { data = [ [|"Chailloux"; "Emmanuel"|] ; [|"Manoury"; "Pascal"|] ]  ;
      card_index = function "Lastname"→0 | "Firstname"→1
                            | _->raise Not_found  } ;;
val base_ex : data_base =
  {card_index = <fun>;
   data = [[|"Chailloux"; "Emmanuel"|]; [|"Manoury"; "Pascal"|]]}
# List.map (field base_ex "Lastname") base_ex.data ;;
- : string list = ["Chailloux"; "Manoury"]
```

The expression *field base_ex* `"Lastname"` evaluates to a function which takes a card and returns the value of its `"Lastname"` field. The library function `List.map` applies the function to each card of the database `base_ex`, and returns the list of the results: a list of the `"Lastname"` fields of the database.

This example shows how we wish to use the functional style in our program. Here, the partial application of `field` allows us to define an access function for a given field, which we can use on any number of cards. This also shows us that the implementation of the `field` function is not very efficient, since although we are always accessing the same field, its index is computed for each access. The following implementation is better:

```
# let field base name =
    let i = base.card_index name in fun (card : data_card) → card.(i) ;;
val field : data_base -> string -> data_card -> string = <fun>
```

Here, after applying the function to two arguments, the index of the field is computed and is used for any subsequent application.

Reading a database from a file

As seen from Objective Caml, a file containing a database is just a list of lines. The first work that needs to be done is to read each line as a string, split it into smaller parts according to the separating character, and then extract the corresponding data as well as the field indexing function.

Tools for processing a line

We need a function `split` that splits a string at every occurrence of some separating character. This function uses the function `suffix` which returns the suffix of a string `s` after some position `i`. To do this, we use three predefined functions:

- `String.length` returns the length of a string;
- `String.sub` returns the substring of `s` starting at position `i` and of length `l`;
- `String.index_from` computes the position of the first occurrence of character `c` in the string `s`, starting at position `n`.

```
# let suffix s i = try String.sub s i ((String.length s)-i)
                   with Invalid_argument("String.sub") → "" ;;
val suffix : string -> int -> string = <fun>
# let split c s =
    let rec split_from n =
      try  let p = String.index_from s n c
           in (String.sub s n (p-n)) :: (split_from (p+1))
      with Not_found → [ suffix s n ]
    in if s="" then [] else split_from 0 ;;
val split : char -> string -> string list = <fun>
```

The only remarkable characteristic in this implementation is the use of exceptions, specifically the exception `Not_found`.

Computing the *data_base* structure There is no difficulty in creating an array of strings from a list of strings, since this is what the `of_list` function in the `Array` module does. It might seem more complicated to compute the index function from a list of field names, but the `List` module provides all the needed tools.

Starting from a list of strings, we need to code a function that associates each string with an index corresponding to its position in the list.

```
# let mk_index list_names =
    let rec make_enum a b = if a > b then [] else a::(make_enum (a+1) b) in
    let list_index =  (make_enum 0 ((List.length list_names) - 1)) in
    let assoc_index_name = List.combine list_names list_index  in
      function name → List.assoc name assoc_index_name  ;;
val mk_index : 'a list -> 'a -> int = <fun>
```

To create the association function between field names and indexes, we combine the list of indexes and the list of names to obtain a list of associations of the type *string * int list*. To look up the index associated with a name, we use the function `assoc` from the `List` library. The function `mk_index` returns a function that takes a name and calls `assoc` on this name and the previously built association list.

It is now possible to create a function that reads a file of the given format.

```
# let read_base filename =
    let channel = open_in filename in
    let split_line = split ':' in
    let list_names = split '|' (input_line channel) in
    let rec read_file () =
      try
        let data = Array.of_list (split_line (input_line channel )) in
          data :: (read_file ())
      with End_of_file →  close_in channel ; []
    in
      { card_index = mk_index list_names ; data = read_file () } ;;
val read_base : string -> data_base = <fun>
```

The auxiliary function `read_file` reads records from the file, and works recursively on the input channel. The base case of the recursion corresponds to the end of the file, signaled by the `End_of_file` exception. In this case, the empty list is returned after closing the channel.

The association's file can now be loaded:

```
# let base_ex = read_base "association.dat" ;;
val base_ex : data_base =
  {card_index = <fun>;
   data =
    [[|"0"; "Chailloux"; "Emmanuel"; "Universit\233 P6"; "0144274427";
       "ec@lip6.fr"; "email"; "25.12.1998"; "100.00"|];
     [|"1"; "Manoury"; "Pascal"; "Laboratoire PPS"; ...|]; ...]}
```

General principles for database processing

The effectiveness and difficulty of processing the data in a database is proportional to the power and complexity of the query language. Since we want to use Objective Caml as query language, there is no limit *a priori* on the requests we can express! However,

we also want to provide some simple tools to manipulate cards and their data. This desire for simplicity requires us to limit the power of the Objective Caml language, through the use of general goals and principles for database processing.

The goal of database processing is to obtain a **state** of the database. Building such a state may be decomposed into three steps:

1. selecting, according to some given criterion, a set of cards;
2. processing each of the selected cards;
3. processing all the data collected on the cards.

Figure 6.1 illustrates this decomposition.

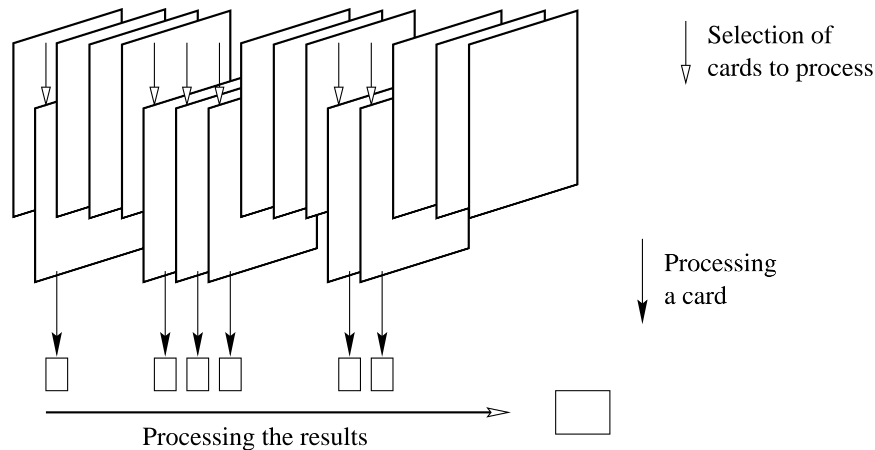


図 6.1: Processing a request.

According to this decomposition, we need three functions of the following types:

1. *(data_card -> bool) -> data_card list -> data_card list*
2. *(data_card -> 'a) -> data_card list -> 'a list*
3. *('a -> 'b -> 'b) -> 'a list -> 'b -> 'b*

Objective Caml provides us with three higher-order function, also known as iterators, introduced page 219, that satisfy our specification:

```
# List.find_all ;;
- : ('a -> bool) -> 'a list -> 'a list = <fun>
# List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# List.fold_right ;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

We will be able to use them to implement the three steps of building a state by choosing the functions they take as an argument.

For some special requests, we will also use:

```
# List.iter ;;
- : ('a -> unit) -> 'a list -> unit = <fun>
```

Indeed, if the required processing consists only of displaying some data, there is nothing to compute.

In the next paragraphs, we are going to see how to define functions expressing simple selection criteria, as well as simple queries. We conclude this section with a short example using these functions according to the principles stated above.

Selection criteria

Concretely, the boolean function corresponding to the selection criterion of a card is a boolean combination of properties of some or all of the fields of the card. Each field of a card, even though it is a string, can contain some information of another type: a float, a date, etc.

Selection criteria on a field

Selecting on some field is usually done using a function of the type *data_base -> 'a -> string -> data_card -> bool*. The *'a* type parameter corresponds to the type of the information contained in the field. The *string* argument corresponds to the name of the field.

String fields We define two simple tests on strings: equality with another string, and non-emptiness.

```
# let eq_sfield db s n dc =  (s = (field db n dc)) ;;
val eq_sfield : data_base -> string -> string -> data_card -> bool = <fun>
# let nonempty_sfield db n dc = ("" <> (field db n dc)) ;;
val nonempty_sfield : data_base -> string -> data_card -> bool = <fun>
```

Float fields To implement tests on data of type float, it is enough to translate the *string* representation of a decimal number into its *float* value. Here are some examples obtained from a generic function `tst_ffield`:

```
# let tst_ffield r db v n dc = r v (float_of_string (field db n dc)) ;;
val tst_ffield :
  ('a -> float -> 'b) -> data_base -> 'a -> string -> data_card -> 'b = <fun>
# let eq_ffield = tst_ffield (=) ;;
# let lt_ffield = tst_ffield (<) ;;
# let le_ffield = tst_ffield (<=) ;;
(* etc. *)
```

These three functions have type:

data_base -> float -> string -> data_card -> bool.

Dates This kind of information is a little more complex to deal with, as it depends on the representation format of dates, and requires that we define date comparison.

We decide to represent dates in a card as a string with format `dd.mm.yyyy`. In order to be able to define additional comparisons, we also allow the replacement of the day, month or year part with the underscore character (`'_'`). Dates are compared according to the lexicographic order of lists of integers of the form *[year; month; day]*. To express queries such as: "is before July 1998", we use the *date pattern*: `"_.07.1998"`. Comparing a date with a pattern is accomplished with the function `tst_dfield` which analyses the pattern to create the *ad hoc* comparison function. To define this generic test function on dates, we need a few auxiliary functions.

We first code two conversion functions from dates (`ints_of_string`) and date patterns (`ints_of_dpat`) to lists of ints. The character `'_'` of a pattern will be replaced by the integer 0:

```
# let split_date = split '.' ;;
val split_date : string -> string list = <fun>
# let ints_of_string d =
    try  match split_date d with
           [d;m;y] → [int_of_string y; int_of_string m; int_of_string d]
          |    _   → failwith "Bad date format"
    with Failure("int_of_string") → failwith "Bad date format" ;;
val ints_of_string : string -> int list = <fun>

# let ints_of_dpat d =
    let int_of_stringpat = function "_" → 0 | s → int_of_string s
    in try  match split_date d with
               [d;m;y] → [ int_of_stringpat y; int_of_stringpat m;
                           int_of_stringpat d ]
              | _ → failwith "Bad date format"
       with Failure("int_of_string") → failwith "Bad date pattern" ;;
val ints_of_dpat : string -> int list = <fun>
```

Given a relation `r` on integers, we now code the test function. It simply consists of implementing the lexicographic order, taking into account the particular case of 0:

```
# let rec app_dtst r d1 d2 = match d1, d2 with
     [] , [] → false
   | (0 :: d1) , (_::d2) → app_dtst r d1 d2
   | (n1 :: d1) , (n2 :: d2) → (r n1 n2) || ((n1 = n2) && (app_dtst r d1 d2))
   | _, _ → failwith "Bad date pattern or format" ;;
val app_dtst : (int -> int -> bool) -> int list -> int list -> bool = <fun>
```

We finally define the generic function `tst_dfield` which takes as arguments a relation `r`, a database `db`, a pattern `dp`, a field name `nm`, and a card `dc`. This function checks that the pattern and the field from the card satisfy the relation.

```
# let tst_dfield r db dp nm dc =
```

```
   r (ints_of_dpat dp) (ints_of_string (field db nm dc)) ;;
val tst_dfield :
  (int list -> int list -> 'a) ->
  data_base -> string -> string -> data_card -> 'a = <fun>
```

We now apply it to three relations.

```
# let eq_dfield = tst_dfield (=) ;;
# let le_dfield = tst_dfield (<=) ;;
# let ge_dfield = tst_dfield (>=) ;;
```

These three functions have type:
`data_base -> string -> string -> data_card -> bool.`

Composing criteria

The tests we have defined above all take as first arguments a database, a value, and the name of a field. When we write a query, the value of these three arguments are known. For instance, when we work on the database `base_ex`, the test "is before July 1998" is written

```
# ge_dfield base_ex "_.07.1998" "Date" ;;
- : data_card -> bool = <fun>
```

Thus, we can consider a test as a function of type `data_card -> bool`. We want to obtain boolean combinations of the results of such functions applied to a given card. To this end, we implement the iterator:

```
# let fold_funs b c fs dc =
   List.fold_right (fun f → fun r → c (f dc) r) fs b ;;
val fold_funs : 'a -> ('b -> 'a -> 'a) -> ('c -> 'b) list -> 'c -> 'a = <fun>
```

Where `b` is the base value, the function `c` is the boolean operator, `fs` is the list of test functions on a field, and `dc` is a card.

We can obtain the conjunction and the disjunction of a list of tests with:

```
# let and_fold fs = fold_funs true (&) fs ;;
val and_fold : ('a -> bool) list -> 'a -> bool = <fun>
# let or_fold fs = fold_funs false (or) fs ;;
val or_fold : ('a -> bool) list -> 'a -> bool = <fun>
```

We easily define the negation of a test:

```
# let not_fun f dc = not (f dc) ;;
val not_fun : ('a -> bool) -> 'a -> bool = <fun>
```

For instance, we can use these combinators to define a selection function for cards whose date field is included in a given range:

```
# let date_interval db d1 d2 =
   and_fold [(le_dfield db d1 "Date"); (ge_dfield db d2 "Date")] ;;
val date_interval : data_base -> string -> string -> data_card -> bool =
```

```
<fun>
```

Processing and computation

It is difficult to guess how a card might be processed, or the data that would result from that processing. Nevertheless, we can consider two common cases: numerical computation and data formatting for printing. Let's take an example for each of these two cases.

Data formatting

In order to print, we wish to create a string containing the name of a member of the association, followed by some information.

We start with a function that reverses the splitting of a line using a given separating character:

```
# let format_list c =
   let s = String.make 1 c in
   List.fold_left (fun x y → if x="" then y else x^s^y) "" ;;
val format_list : char -> string list -> string = <fun>
```

In order to build the list of fields we are interested in, we code the function `extract` that returns the fields associated with a given list of names in a given card:

```
# let extract db ns dc =
    List.map (fun n → field db n dc) ns  ;;
val extract : data_base -> string list -> data_card -> string list = <fun>
```

We can now write the line formatting function:

```
# let format_line db ns dc =
   (String.uppercase (field db "Lastname" dc))
   ^" "^(field db "Firstname" dc)
   ^"\t"^(format_list '\t' (extract db ns dc))
   ^"\n" ;;
val format_line : data_base -> string list -> data_card -> string = <fun>
```

The argument `ns` is the list of requested fields. In the resulting string, fields are separated by a tab (`'\t'`) and the string is terminated with a newline character.

We display the list of last and first names of all members with:

```
# List.iter print_string (List.map (format_line base_ex [] ) base_ex.data) ;;
CHAILLOUX Emmanuel
MANOURY Pascal
PAGANO Bruno
BARO Sylvain
- : unit = ()
```

Numerical computation

We want to compute the total amount of received fees for a given set of cards. This is easily done by composing the extraction and conversion of the correct field with the addition. To get nicer code, we define an infix composition operator:

```
# let (++) f g x = g (f x) ;;
val ( ++ ) : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c = <fun>
```

We use this operator in the following definition:

```
# let total db dcs =
    List.fold_right ((field db "Amount") ++ float_of_string ++ (+.)) dcs 0.0 ;;
val total : data_base -> data_card list -> float = <fun>
```

We can now apply it to the whole database:

```
# total base_ex base_ex.data ;;
- : float = 450
```

An example

To conclude, here is a small example of an application that uses the principles described in the paragraphs above.

We expect two kinds of queries on our database:

- a query returning two lists, the elements of the first containing the name of a member followed by his mail address, the elements of the other containing the name of the member followed by his email address, according to his preferences.
- another query returning the state of received fees for a given period of time. This state is composed of the list of last and first names, dates and amounts of the fees as well as the total amount of the received fees.

List of addresses

To create these lists, we first select the relevant cards according to the field `"Pref"`, then we use the formatting function `format_line`:

```
# let mail_addresses db =
   let dcs = List.find_all (eq_sfield db "mail" "Pref") db.data in
    List.map (format_line db ["Mail"]) dcs ;;
val mail_addresses : data_base -> string list = <fun>

# let email_addresses db =
   let dcs = List.find_all (eq_sfield db "email" "Pref") db.data in
    List.map (format_line db ["Email"]) dcs ;;
val email_addresses : data_base -> string list = <fun>
```

State of received fees

Computing the state of the received fees uses the same technique: selection then processing. In this case however the processing part is twofold: line formatting followed by the computation of the total amount.

```
# let fees_state db d1 d2 =
    let dcs = List.find_all (date_interval db d1 d2) db.data in
    let ls = List.map (format_line db ["Date";"Amount"]) dcs in
    let t = total db dcs in
     ls, t ;;
val fees_state : data_base -> string -> string -> string list * float = <fun>
```

The result of this query is a tuple containing a list of strings with member information, and the total amount of received fees.

Main program

The main program is essentially an interactive loop that displays the result of queries asked by the user through a menu. We use here an imperative style, except for the display of the results which uses an iterator.

```
# let main() =
    let db = read_base "association.dat" in
    let finished = ref false in
     while not !finished do
      print_string" 1: List of mail addresses\n";
      print_string" 2: List of email addresses\n";
      print_string" 3: Received fees\n";
      print_string" 0: Exit\n";
      print_string"Your choice: ";
      match read_int() with
        0 → finished := true
      | 1 → (List.iter print_string (mail_addresses db))
      | 2 → (List.iter print_string (email_addresses db))
      | 3
       → (let d1 = print_string"Start date: "; read_line() in
           let d2 = print_string"End date: "; read_line() in
           let ls, t = fees_state db d1 d2 in
            List.iter print_string ls;
            print_string"Total: "; print_float t; print_newline())
      | _ → ()
     done;
     print_string"bye\n" ;;
val main : unit -> unit = <fun>
```

This example will be extended in chapter 21 with an interface using a web browser.

Further work

A natural extension of this example would consist of adding type information to every field of the database. This information would be used to define generic comparison operators with type `data_base -> 'a -> string -> data_card -> bool` where the name of the field (the third argument) would trigger the correct conversion and test functions.

BASIC interpreter

The application described in this section is a program interpreter for Basic. Thus, it is a program that can run other programs written in Basic. Of course, we will only deal with a restricted language, which contains the following commands:

- **PRINT** *expression*
  Prints the result of the evaluation of the expression.
- **INPUT** *variable*
  Prints a *prompt* (`?`), reads an integer typed in by the user, and assigns its value to the variable.
- **LET** *variable* = *expression*
  Assigns the result of the evaluation of *expression* to the variable.
- **GOTO** *line number*
  Continues execution at the given line.
- **IF** *condition* **THEN** *line number*
  Continues execution at the given line if the *condition* is true.
- **REM** *any string*
  One-line comment.

Every line of a Basic program is labelled with a line number, and contains only one command. For instance, a program that computes and then prints the factorial of an integer given by the user is written:

```
 5  REM inputting the argument
10  PRINT " factorial of:"
20  INPUT A
30  LET B = 1
35  REM beginning of the loop
40  IF A <= 1 THEN 80
50  LET B = B * A
60  LET A = A - 1
70  GOTO 40
75  REM prints the result
80  PRINT B
```

We also wish to write a small text editor, working as a toplevel interactive loop. It should be able to add new lines, display a program, execute it, and display the result.

Execution of the program is started with the RUN command. Here is an example of the evaluation of this program:

```
> RUN
 factorial of: ? 5
120
```

The interpreter is implemented in several distinct parts:

Description of the abstract syntax : describes the definition of data types to represent Basic programs, as well as their components (lines, commands, expressions, etc.).

Program pretty printing : consists of transforming the internal representation of Basic programs to strings, in order to display them.

Lexing and parsing : accomplish the inverse transformation, that is, transform a string into the internal representation of a Basic program (the abstract syntax).

Evaluation : is the heart of the interpreter. It controls and runs the program. As we will see, functional languages, such as Objective Caml, are particularly well adapted for this kind of problem.

Toplevel interactive loop : glues together all the previous parts.

Abstract syntax

Figure 6.2 introduces the concrete syntax, as a BNF grammar, of the Basic we will implement. This kind of description for language syntaxes is described in chapter 11, page 297.

We can see that the way expressions are defined does not ensure that a **well formed** expression can be evaluated. For instance, 1+"hello" is an expression, and yet it is not possible to evaluate it. This deliberate choice lets us simplify both the abstract syntax and the parsing of the Basic language. The price to pay for this choice is that a syntactically correct Basic program may generate a runtime error because of a type mismatch.

Defining Objective Caml data types for this abstract syntax is easy, we simply translate the concrete syntax into a sum type:

```
# type unr_op = UMINUS | NOT ;;
# type bin_op = PLUS | MINUS | MULT | DIV | MOD
              | EQUAL | LESS | LESSEQ | GREAT | GREATEQ | DIFF
              | AND | OR ;;
# type expression =
      ExpInt of int
    | ExpVar of string
    | ExpStr of string
```

```
UNARY_OP    ::=  −  |  !

BINARY_OP   ::=  +  |  −  |  ∗  |  /  |  %
             |   =  |  <  |  >  |  <=  |  >=  |  <>
             |   &  |  '|'

EXPRESSION  ::=  integer
             |   variable
             |   "string"
             |   UNARY_OP  EXPRESSION
             |   EXPRESSION  BINARY_OP  EXPRESSION
             |   ( EXPRESSION )

COMMAND     ::=  REM string
             |   GOTO integer
             |   LET variable = EXPRESSION
             |   PRINT EXPRESSION
             |   INPUT variable
             |   IF EXPRESSION THEN integer

LINE        ::=  integer COMMAND

PROGRAM     ::=  LINE
             |   LINE PROGRAM

PHRASE      ::=  LINE | RUN | LIST | END
```

図 6.2: BASIC Grammar.

```
    | ExpUnr of unr_op * expression
    | ExpBin of expression * bin_op * expression  ;;
# type command =
      Rem of string
    | Goto of int
    | Print of expression
    | Input of string
    | If of expression * int
    | Let of string * expression  ;;
# type line = { num : int ; cmd : command }  ;;
# type program = line list  ;;
```

We also define the abstract syntax for the commands for the small program editor:

```
# type phrase =  Line of line | List | Run | PEnd  ;;
```

It is convenient to allow the programmer to skip some parentheses in arithmetic expressions. For instance, the expression $1 + 3 * 4$ is usually interpreted as $1 + (3 * 4)$. To this end, we associate an integer with each operator of the language:

```
# let priority_uop = function NOT → 1 | UMINUS → 7
  let priority_binop = function
      MULT | DIV  → 6
    | PLUS | MINUS → 5
    | MOD → 4
    | EQUAL | LESS | LESSEQ | GREAT | GREATEQ | DIFF → 3
    | AND | OR → 2 ;;
val priority_uop : unr_op -> int = <fun>
val priority_binop : bin_op -> int = <fun>
```

These integers indicate the **priority** of the operators. They will be used to print and parse programs.

Program pretty printing

To print a program, one needs to be able to convert abstract syntax program lines into strings.

Converting operators is easy:

```
# let pp_binop = function
      PLUS → "+" | MULT → "*" | MOD   → "%" | MINUS → "-"
    | DIV → "/"  | EQUAL → " = " | LESS → " < "
    | LESSEQ → " <= "   | GREAT   → " > "
    | GREATEQ → " >= "   | DIFF  → " <> " | AND → " & " | OR → " | "
  let pp_unrop = function UMINUS → "-"  | NOT → "!"  ;;
val pp_binop : bin_op -> string = <fun>
val pp_unrop : unr_op -> string = <fun>
```

Expression printing needs to take into account operator priority to print as few parentheses as possible. For instance, parentheses are put around a subexpression at the right of an operator only if the subexpression's main operator has a lower priority that the main operator of the whole expression. Also, arithmetic operators are left-associative, thus the expression $1 - 2 - 3$ is interpreted as $(1 - 2) - 3$.

To deal with this, we use two auxiliary functions `ppl` and `ppr` to print left and right subtrees, respectively. These functions take two arguments: the tree to print and the priority of the enclosing operator, which is used to decide if parentheses are necessary. Left and right subtrees are distinguished to deal with associativity. If the current operator priority is the same than the enclosing operator priority, left trees do not need parentheses whereas right ones may require them, as in $1 - (2 - 3)$ or $1 - (2 + 3)$.

The initial tree is taken as a left subtree with minimal priority (0). The expression pretty printing function `pp_expression` is:

```
# let parenthesis x = "(" ^ x ^ ")";;
val parenthesis : string -> string = <fun>
# let pp_expression =
   let rec ppl pr = function
        ExpInt n → (string_of_int n)
      | ExpVar v → v
      | ExpStr s → "\"" ^ s ^ "\""
      | ExpUnr (op,e) →
          let res = (pp_unrop op)^(ppl (priority_uop op) e)
          in if pr=0 then res else parenthesis res
      | ExpBin (e1,op,e2) →
          let pr2 = priority_binop op
          in let res = (ppl pr2 e1)^(pp_binop op)^(ppr pr2 e2)
          (* parenthesis if priority is not greater *)
          in if pr2 >= pr then res else parenthesis res
    and ppr pr exp = match exp with
      (*  right subtrees only differ for binary operators *)
        ExpBin (e1,op,e2) →
          let pr2 = priority_binop op
          in let res = (ppl pr2 e1)^(pp_binop op)^(ppr pr2 e2)
          in if pr2 > pr then res else parenthesis res
      | _ → ppl pr exp
    in ppl 0 ;;
val pp_expression : expression -> string = <fun>
```

Command pretty printing uses the expression pretty printing function. Printing a line consists of printing the line number before the command.

```
# let pp_command = function
      Rem s      →  "REM " ^ s
    | Goto n     →  "GOTO " ^ (string_of_int n)
    | Print e    →  "PRINT " ^ (pp_expression e)
    | Input v    →  "INPUT " ^ v
    | If (e,n)   →  "IF "^(pp_expression e)^" THEN "^(string_of_int n)
    | Let (v,e)  →  "LET " ^ v ^ " = " ^ (pp_expression e)  ;;
val pp_command : command -> string = <fun>
# let pp_line l = (string_of_int l.num) ^ "  " ^ (pp_command l.cmd)  ;;
val pp_line : line -> string = <fun>
```

Lexing

Lexing and parsing do the inverse transformation of printing, going from a string to a syntax tree. Lexing splits the text of a command line into independent lexical units called **lexemes**, with Objective Caml type:

```
# type lexeme = Lint of int
              | Lident of string
```

```
                | Lsymbol of string
                | Lstring of string
                | Lend ;;
```

A particular lexeme denotes the end of an expression: `Lend`. It is not present in the text of the expression, but is created by the lexing function (see the `lexer` function, page 165).

The string being lexed is kept in a record that contains a mutable field indicating the position after which lexing has not been done yet. Since the size of the string is used several times and does not change, it is also stored in the record:

```
# type string_lexer = {string:string; mutable current:int; size:int } ;;
```

This representation lets us define the lexing of a string as the application of a function to a value of type *string_lexer* returning a value of type *lexeme*. Modifying the current position in the string is done as a side effect.

```
# let init_lex s = { string=s; current=0 ; size=String.length s } ;;
val init_lex : string -> string_lexer = <fun>
# let forward cl = cl.current <- cl.current+1  ;;
val forward : string_lexer -> unit = <fun>
# let forward_n cl n = cl.current <- cl.current+n ;;
val forward_n : string_lexer -> int -> unit = <fun>
# let extract pred cl =
    let st = cl.string and pos = cl.current in
    let rec ext n = if n<cl.size && (pred st.[n]) then ext (n+1) else n in
    let res = ext pos
    in cl.current <- res ; String.sub cl.string pos (res-pos)  ;;
val extract : (char -> bool) -> string_lexer -> string = <fun>
```

The following functions extract a lexeme from the string and modify the current position. The two functions `extract_int` and `extract_ident` extract an integer and an identifier, respectively.

```
# let extract_int =
     let is_int = function '0'..'9' -> true | _ -> false
     in function cl -> int_of_string (extract is_int cl)
  let extract_ident =
     let is_alpha_num = function
       'a'..'z' | 'A'..'Z' | '0' .. '9' | '_' -> true
     | _ -> false
     in extract is_alpha_num ;;
val extract_int : string_lexer -> int = <fun>
val extract_ident : string_lexer -> string = <fun>
```

The `lexer` function uses the two previous functions to extract a lexeme.

```
# exception LexerError ;;
exception LexerError
```

```
# let  rec lexer cl =
    let lexer_char c = match c with
        ' '
      | '\t'     → forward cl ; lexer cl
      | 'a'..'z'
      | 'A'..'Z' → Lident (extract_ident cl)
      | '0'..'9' → Lint (extract_int cl)
      | '"'      → forward cl ;
                   let res = Lstring (extract ((<>) '"') cl)
                   in forward cl ; res
      | '+' | '-' | '*' | '/' | '%' | '&' | '|' | '!' | '=' | '(' | ')'  →
                   forward cl; Lsymbol (String.make 1 c)
      | '<'
      | '>'      → forward cl;
                   if cl.current >= cl.size then Lsymbol (String.make 1 c)
                   else  let cs = cl.string.[cl.current]
                         in ( match (c,cs) with
                                ('<','=') → forward cl; Lsymbol "<="
                              | ('>','=') → forward cl; Lsymbol ">="
                              | ('<','>') → forward cl; Lsymbol "<>"
                              |     _     → Lsymbol (String.make 1 c) )
      | _ → raise LexerError
    in
       if cl.current >= cl.size then Lend
       else lexer_char cl.string.[cl.current] ;;
val lexer : string_lexer -> lexeme = <fun>
```

The `lexer` function is very simple: it matches the current character of a string and, based on its value, extracts the corresponding lexeme and modifies the current position to the start of the next lexeme. The code is simple because, for all characters except two, the current character defines which lexeme to extract. In the more complicated cases of `'<'`, we need to look at the next character, which might be a `'='` or a `'>'`, producing two different lexemes. The same problem arises with `'>'`.

Parsing

The only difficulty in parsing our language comes from expressions. Indeed, knowing the beginning of an expression is not enough to know its structure. For instance, having parsed the beginning of an expression as being $1 + 2 + 3$, the resulting syntax tree for this part depends on the rest of the expression: its structure is different when it is followed by $+4$ or $*4$ (see figure 6.3). However, since the tree structure for $1 + 2$ is the same in both cases, it can be built. As the position of $+3$ in the structure is not fully known, it is temporarily stored.

To build the abstract syntax tree, we use a **pushdown automaton** similar to the one built by *yacc* (see page 305). Lexemes are read one by one and put on a stack until

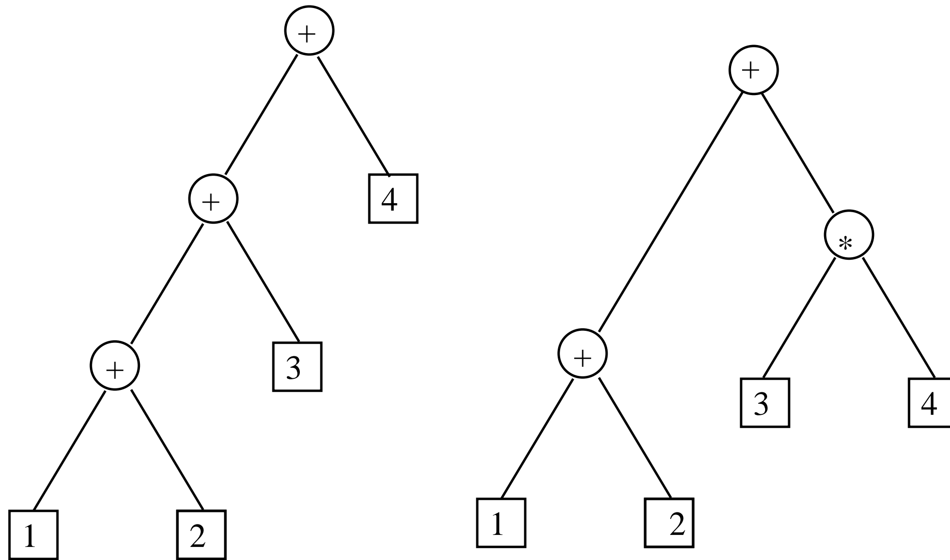


図 6.3: Basic: abstract syntax tree examples.

there is enough information to build the expression. They are then removed from the stack and replaced by the expression. This latter operation is called **reduction**.

The stack elements have type:

```
# type exp_elem =
      Texp of expression   (* expression       *)
    | Tbin of bin_op       (* binary operator  *)
    | Tunr of unr_op       (* unary operator   *)
    | Tlp                  (* left parenthesis *) ;;
```

Right parentheses are not stored on the stack as only left parentheses matter for reduction.

Figure 6.4 illustrates the way the stack is used to parse the expression $(1 + 2 * 3) + 4$. The character above the arrow is the current character of the string.

We define an exception for syntax errors.

```
# exception ParseError ;;
```

The first step consists of transforming symbols into operators:

```
# let unr_symb = function
    "!" → NOT | "-" → UMINUS | _ → raise ParseError
  let bin_symb = function
      "+" → PLUS | "-" → MINUS | "*" → MULT | "/" → DIV | "%" → MOD
    | "=" → EQUAL | "<" → LESS | "<=" → LESSEQ | ">" → GREAT
    | ">=" → GREATEQ | "<>" → DIFF | "&" → AND | "|" → OR
    | _ → raise ParseError
  let tsymb s = try Tbin (bin_symb s) with ParseError → Tunr (unr_symb s) ;;
```

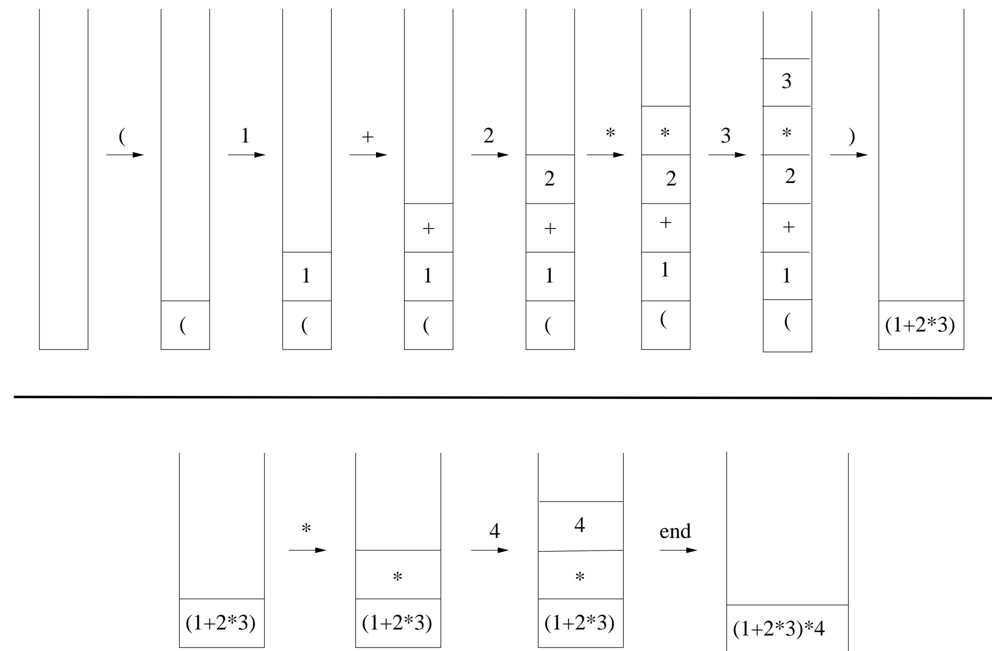


図 6.4: Basic: abstract syntax tree construction example.

```
val unr_symb : string -> unr_op = <fun>
val bin_symb : string -> bin_op = <fun>
val tsymb : string -> exp_elem = <fun>
```

The `reduce` function implements stack reduction. There are two cases to consider, whether the stack starts with:

- an expression followed by a unary operator,
- an expression followed by a binary operator and an expression.

Moreover, `reduce` takes an argument indicating the minimal priority that an operator should have to trigger reduction. To avoid this reduction condition, it suffices to give the minimal value, zero, as the priority.

```
# let reduce pr = function
      (Texp e) :: (Tunr op) :: st  when  (priority_uop op) >= pr
         → (Texp (ExpUnr (op,e))) :: st
    | (Texp e1) :: (Tbin op) :: (Texp e2) :: st  when  (priority_binop op) >= pr
         → (Texp (ExpBin (e2,op,e1))) :: st
    | _ → raise ParseError ;;
val reduce : int -> exp_elem list -> exp_elem list = <fun>
```

Notice that expression elements are stacked as they are read. Thus it is necessary to swap them when they are arguments of a binary operator.

The main function of our parser is `stack_or_reduce` that, according to the lexeme given in argument, puts it on the stack or triggers a reduction.

```
# let rec stack_or_reduce lex stack = match lex , stack with
      Lint n ,  _      →  (Texp (ExpInt n)) :: stack
    | Lident v ,  _    →  (Texp (ExpVar v)) :: stack
    | Lstring s , _    →  (Texp (ExpStr s)) :: stack
    | Lsymbol "(" , _  →  Tlp :: stack
    | Lsymbol ")" , (Texp e) :: Tlp :: st  →  (Texp e) :: st
    | Lsymbol ")" , _ → stack_or_reduce lex (reduce 0 stack)
    | Lsymbol s , _
         → let symbol =
              if s<>"-" then tsymb s
              (* remove the ambiguity of the ‘‘-’’ symbol          *)
              (* according to the last exp element put on the stack *)
              else match stack
                   with (Texp _)::_  →  Tbin MINUS
                                  | _ →  Tunr UMINUS
            in ( match symbol with
                   Tunr op  →  (Tunr op) :: stack
                 | Tbin op  →
                     ( try stack_or_reduce lex (reduce (priority_binop op)
                                                  stack )
                       with ParseError → (Tbin op) :: stack )
                 | _ → raise ParseError )
    | _ , _ → raise ParseError ;;
val stack_or_reduce : lexeme -> exp_elem list -> exp_elem list = <fun>
```

Once all lexemes are defined and stacked, the function `reduce_all` builds the abstract syntax tree with the elements remaining in the stack. If the expression being parsed is well formed, only one element should remain in the stack, containing the tree for this expression.

```
# let rec reduce_all = function
    | [] → raise ParseError
    | [Texp x] → x
    | st → reduce_all (reduce 0 st) ;;
val reduce_all : exp_elem list -> expression = <fun>
```

The `parse_exp` function is the main expression parsing function. It reads a string, extracts its lexemes and passes them to the `stack_or_reduce` function. Parsing stops when the current lexeme satisfies a predicate that is given as an argument.

```
# let parse_exp stop cl =
    let p = ref 0 in
    let rec parse_one stack  =
```

```
      let l = ( p:=cl.current ; lexer cl)
      in if not (stop l) then parse_one (stack_or_reduce l stack)
         else ( cl.current <- !p ; reduce_all stack )
    in parse_one []  ;;
val parse_exp : (lexeme -> bool) -> string_lexer -> expression = <fun>
```

Notice that the lexeme that made the parsing stop is not used to build the expression. It is thus necessary to modify the current position to its beginning (variable p) to parse it later.

We can now parse a command line:

```
# let parse_cmd cl = match lexer cl with
      Lident s → ( match s with
          "REM" → Rem (extract (fun _ → true) cl)
        | "GOTO" → Goto (match lexer cl with
                           Lint p → p
                         | _ → raise ParseError)
        | "INPUT" → Input (match lexer cl with
                             Lident v → v
                           | _ → raise ParseError)
        | "PRINT" → Print (parse_exp ((=) Lend) cl)
        | "LET" →
            let l2 = lexer cl and l3 = lexer cl
            in ( match l2 ,l3 with
                      (Lident v,Lsymbol "=") → Let (v,parse_exp ((=) Lend) cl)
                    | _ → raise ParseError )
        | "IF" →
            let test = parse_exp ((=) (Lident "THEN")) cl
            in ( match ignore (lexer cl) ; lexer cl with
                     Lint n  →  If (test,n)
                   | _ → raise ParseError )
        | _ → raise ParseError )
    | _ → raise ParseError  ;;
val parse_cmd : string_lexer -> command = <fun>
```

Finally, we implement the function to parse commands typed by the user:

```
# let parse str =
    let cl = init_lex str
    in match lexer cl with
        Lint n → Line { num=n ; cmd=parse_cmd cl }
      | Lident "LIST" → List
      | Lident "RUN" → Run
      | Lident "END" → PEnd
      | _ → raise ParseError  ;;
val parse : string -> phrase = <fun>
```

Evaluation

A Basic program is a list of lines. Execution starts at the first line. Interpreting a program line consists of executing the task corresponding to its command. There are three different kinds of commands: input-output (**PRINT** and **INPUT**), variable declaration or modification (**LET**), and flow control (**GOTO** and **IF... THEN**). Input-output commands interact with the user and use the corresponding Objective Caml functions.

Variable declaration and modification commands need to know how to compute the value of an arithmetic expression and the memory location to store the result. Expression evaluation returns an integer, a boolean, or a string. Their type is `value`.

```
# type value = Vint of int | Vstr of string | Vbool of bool ;;
```

Variable declaration should allocate some memory to store the associated value. Similarly, variable modification requires the modification of the associated value. Thus, evaluation of a Basic program uses an *environment* that stores the association between a variable name and its value. It is represented by an association list of tuples (name,value):

```
# type environment = (string * value) list ;;
```

The variable name is used to access its value. Variable modification modifies the association.

Flow control commands, conditional or unconditional, specify the number of the next line to execute. By default, it is the next line. To do this, it is necessary to remember the number of the current line.

The list of commands representing the program being edited under the toplevel is not an efficient data structure for running the program. Indeed, it is then necessary to look at the whole list of lines to find the line indicated by a flow control command (`If` and `goto`). Replacing the list of lines with an array of commands allows direct access to the command following a flow control command, using the array index instead of the line number in the flow control command. This solution requires some preprocessing called **assembly** before executing a `RUN` command. For reasons that will be detailed shortly, a program after assembly is not represented as an array of commands but as an array of lines:

```
# type code = line array ;;
```

As in the calculator example of previous chapters, the interpreter uses a state that is modified for each command evaluation. At each step, we need to remember the whole program, the next line to interpret and the values of the variables. The program being interpreted is not exactly the one that was entered in the toplevel: instead of a list of commands, it is an array of commands. Thus the state of a program during execution is:

```
# type state_exec = { line:int ; xprog:code ; xenv:environment } ;;
```

Two different reasons may lead to an error during the evaluation of a line: an error while computing an expression, or branching to an absent line. They must be dealt with so that the interpreter exits nicely, printing an error message. We define an exception as well as a function to raise it, indicating the line where the error occurred.

```
# exception RunError of int
  let runerr n = raise (RunError n) ;;
exception RunError of int
val runerr : int -> 'a = <fun>
```

Assembly Assembling a program that is a list of numbered lines (type *program*) consists of transforming this list into an array and modifying the flow control commands. This last modification only needs an association table between line numbers and array indexes. This is easily provided by storing lines (with their line numbers), instead of commands, in the array: to find the association between a line number and the index in the array, we look the line number up in the array and return the corresponding index. If no line is found with this number, the index returned is `-1`.

```
# exception Result_lookup_index of int ;;
exception Result_lookup_index of int
# let lookup_index tprog num_line =
    try
      for i=0 to (Array.length tprog)-1 do
        let num_i = tprog.(i).num
        in if num_i=num_line then raise (Result_lookup_index i)
           else if num_i>num_line then raise (Result_lookup_index (-1))
      done ;
      (-1 )
    with Result_lookup_index i → i ;;
val lookup_index : line array -> int -> int = <fun>

# let assemble prog =
    let tprog = Array.of_list prog in
      for i=0 to (Array.length tprog)-1 do
        match tprog.(i).cmd with
            Goto n → let index = lookup_index tprog n
                     in tprog.(i) <- { tprog.(i) with cmd = Goto index }
          | If(c,n) → let index = lookup_index tprog n
                      in tprog.(i) <- { tprog.(i) with cmd = If (c,index) }
          | _ → ()
      done ;
    tprog ;;
val assemble : line list -> line array = <fun>
```

Expression evaluation The evaluation function does a depth-first traversal on the abstract syntax tree, and executes the operations indicated at each node.

The RunError exception is raised in case of type inconsistency, division by zero, or an undeclared variable.

```
# let rec eval_exp n envt expr = match expr with
      ExpInt p  →  Vint p
    | ExpVar v  → ( try List.assoc v envt with Not_found → runerr n )
    | ExpUnr (UMINUS,e) →
        ( match eval_exp n envt e with
            Vint p → Vint (-p)
          | _ → runerr n )
    | ExpUnr (NOT,e) →
        ( match eval_exp n envt e with
            Vbool p → Vbool (not p)
          | _ → runerr n )
    | ExpStr s → Vstr s
    | ExpBin (e1,op,e2)
        → match eval_exp n envt e1 , op , eval_exp n envt e2 with
               Vint v1 , PLUS   , Vint v2  →  Vint (v1 + v2)
             | Vint v1 , MINUS  , Vint v2  →  Vint (v1 - v2)
             | Vint v1 , MULT   , Vint v2  →  Vint (v1 * v2)
             | Vint v1 ,  DIV   , Vint v2  when v2<>0 →  Vint (v1 / v2)
             | Vint v1 ,  MOD   , Vint v2  when v2<>0 →  Vint (v1 mod v2)

             | Vint v1 , EQUAL   , Vint v2  →  Vbool (v1 = v2)
             | Vint v1 , DIFF    , Vint v2  →  Vbool (v1 <> v2)
             | Vint v1 ,  LESS   , Vint v2  →  Vbool (v1 < v2)
             | Vint v1 ,  GREAT  , Vint v2  →  Vbool (v1 > v2)
             | Vint v1 , LESSEQ  , Vint v2  →  Vbool (v1 <= v2)
             | Vint v1 , GREATEQ , Vint v2  →  Vbool (v1 >= v2)

             | Vbool v1 , AND , Vbool v2  →  Vbool (v1 && v2)
             | Vbool v1 , OR , Vbool v2  →  Vbool (v1 || v2)

             | Vstr v1 , PLUS , Vstr v2 → Vstr (v1 ^ v2)
             | _ , _ , _  → runerr n  ;;
val eval_exp : int -> (string * value) list -> expression -> value = <fun>
```

Command evaluation To evaluate a command, we need a few additional functions.

We add an association to an environment by removing a previous association for the same variable name if there is one:

```
# let rec add v e env = match env with
      []  → [v,e]
    | (w,f) :: l → if w=v then (v,e) :: l else (w,f) :: (add v e l) ;;
val add : 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list = <fun>
```

A function that prints the value of an integer or string is useful for evaluation of the PRINT command.

```
# let print_value v = match v with
      Vint n → print_int n
    | Vbool true → print_string "true"
    | Vbool false → print_string "false"
    | Vstr s → print_string s ;;
val print_value : value -> unit = <fun>
```

The execution of a command corresponds to a **transition** from one state to another. More precisely, the environment is modified if the command is an assignment. Furthermore, the next line to execute is always modified. As a convention, if the next line to execute does not exist, we set its value to -1

```
# let next_line state =
   let n = state.line+1 in
    if n < Array.length state.xprog then n else -1 ;;
val next_line : state_exec -> int = <fun>
# let eval_cmd state =
    match state.xprog.(state.line).cmd with
      Rem _    →  { state with line = next_line state }
    | Print e  →  print_value (eval_exp state.line state.xenv e) ;
                  print_newline () ;
                  { state with line = next_line state }
    | Let(v,e) →  let ev = eval_exp state.line state.xenv e
                    in { state with line = next_line state ;
                                   xenv = add v ev state.xenv }
    | Goto n   →  { state with line = n }
    | Input v  →  let x = try read_int ()
                          with Failure "int_of_string" → 0
                  in { state with line = next_line state;
                                 xenv = add v (Vint x) state.xenv }
    | If (t,n) →  match eval_exp state.line state.xenv t with
                    Vbool true  → { state with line = n }
                  | Vbool false → { state with line = next_line state }
                  | _ → runerr state.line  ;;
val eval_cmd : state_exec -> state_exec = <fun>
```

On each call of the transition function eval_cmd, we look up the current line, run it, then set the number of the next line to run as the current line. If the last line of the program is reached, the current line is given the value -1. This will tell us when to stop.

Program evaluation We recursively apply the transition function until we reach a state where the current line number is -1.

```
# let rec run state =
```

```
    if state.line = -1 then state else run (eval_cmd state) ;;
val run : state_exec -> state_exec = <fun>
```

Finishing touches

The only thing left to do is to write a small editor and to plug together all the functions we wrote in the previous sections.

The `insert` function adds a new line in the program at the requested place.

```
# let rec insert line p = match p with
       [] → [line]
     | l :: prog →
         if l.num < line.num then l :: (insert line prog)
         else if l.num=line.num then line :: prog
         else line :: l :: prog ;;
val insert : line -> line list -> line list = <fun>
```

The `print_prog` function prints the source code of a program.

```
# let print_prog prog =
     let print_line x = print_string (pp_line x) ; print_newline () in
      print_newline () ;
      List.iter print_line prog ;
      print_newline () ;;
val print_prog : line list -> unit = <fun>
```

The `one_command` function processes the insertion of a line or the execution of a command. It modifies the state of the toplevel loop, which consists of a program and an environment. This state, represented by the *loop_state* type, is different from the evaluation state.

```
# type loop_state = { prog:program; env:environment } ;;
# exception End ;;

# let one_command state =
     print_string "> " ; flush stdout ;
     try
       match parse (input_line stdin) with
           Line l → { state with prog = insert l state.prog }
         | List  → (print_prog state.prog ; state )
         | Run
          → let tprog = assemble state.prog in
             let xstate = run { line = 0; xprog = tprog; xenv = state.env } in
              {state with env = xstate.xenv }
         | PEnd → raise End
     with
```

```
        LexerError → print_string "Illegal character\n"; state
      | ParseError → print_string "syntax error\n"; state
      | RunError n →
          print_string "runtime error at line ";
          print_int n ;
          print_string "\n";
          state ;;
val one_command : loop_state -> loop_state = <fun>
```

The main function is the go function, which starts the toplevel loop of our Basic.

```
# let go () =
  try
    print_string "Mini-BASIC version 0.1\n\n";
    let rec loop state = loop (one_command state)  in
      loop { prog = [] ; env = []  }
  with End → print_string "See you later...\n";;
val go : unit -> unit = <fun>
```

The loop is implemented by the local function `loop`. It stops when the `End` exception is raised by the `one_command` function.

Example: C+/C-

We return to the example of the C+/C- game described in chapter 3, page 76. Here is the Basic program corresponding to that Objective Caml program:

```
10 PRINT "Give the hidden number: "
20 INPUT N
30 PRINT "Give a number: "
40 INPUT R
50 IF R = N THEN 110
60 IF R < N THEN 90
70 PRINT "C-"
80 GOTO 30
90 PRINT "C+"
100 GOTO 30
110 PRINT "CONGRATULATIONS"
```

And here is a sample run of this program.

```
> RUN
Give the hidden number:
64
Give a number:
88
C-
```

```
Give a number:
44
C+
Give a number:
64
CONGRATULATIONS
```

Further work

The Basic we implemented is minimalist. If you want to go further, the following exercises hint at some possible extensions.

1. *Floating-point numbers*: as is, our language only deals with integers, strings and booleans. Add floats, as well as the corresponding arithmetic operations in the language grammar. We need to modify not only parsing, but also evaluation, taking into account the implicit conversions between integers and floats.
2. *Arrays*: Add to the syntax the command `DIM var[x]` that declares an array `var` of size `x`, and the expression `var[i]` that references the `i`th element of the array `var`.
3. *Toplevel directives*: Add the toplevel directives `SAVE "file_name"` and `LOAD "file_name"` that save a Basic program to the hard disk, and load a Basic program from the hard disk, respectively.
4. *Sub-program*: Add sub-programs. The `GOSUB line number` command calls a sub-program by branching to the given line number while storing the line from where the call is made. The `RETURN` command resumes execution at the line following the last `GOSUB` call executed, if there is one, or exits the program otherwise. Adding sub-programs requires evaluation to manage not only the environement but also a stack containing the return addresses of the current `GOSUB` calls. The `GOSUB` command adds the possibility of defining recursive sub-programs.

Minesweeper

Let us briefly recall the object of this game: to explore a mine field without stepping on one. A mine field is a two dimensional array (a matrix) where some cells contain hidden mines while others are empty. At the beginning of the game, all the cells are closed and the player must open them one after another. The player wins when he opens all the cells that are empty.

Every turn, the player may open a cell or flag it as containing a mine. If he opens a cell that contains a mine, it blows up and the player loses. If the cell is empty, its appearance is modified and the number of mines in the 8 neighbor cells is displayed (thus at most 8). If the player decides to flag a cell, he cannot open it until he removes the flag.

We split the implementation of the game into three parts.

図 6.5: Screenshot.

1. The abstract game, including the internal representation of the mine field as well as the functions manipulating this representation.
2. The graphical part of the game, including the function for displaying cells.
3. The interaction between the program and the player.

The abstract mine field

This part deals with the mine field as an abstraction only, and does not address its display.

Configuration A mine field is defined by its dimensions and the number of mines it contains. We group these three pieces of data in a record and define a default configuration: 10×10 cells and 15 mines.

```
# type config = {
    nbcols  : int ;
    nbrows : int ;
    nbmines : int };;
```

```
# let default_config = { nbcols=10; nbrows=10; nbmines=15 } ;;
```

The mine field It is natural to represent the mine field as a two dimensional array. However, it is still necessary to specify what the cells are, and what information their encoding should provide. The state of a cell should answer the following questions:

- is there a mine in this cell?
- is this cell opened (has it been seen)?
- is this cell flagged?
- how many mines are there in neighbor cells?

The last item is not mandatory, as it is possible to compute it when it is needed. However, it is simpler to do this computation once at the beginning of the game.

We represent a cell with a record that contains these four pieces of data.

```
# type cell = {
    mutable mined : bool ;
    mutable seen : bool ;
    mutable flag : bool ;
    mutable nbm : int
  } ;;
```

The two dimensional array is an array of arrays of cells:

```
# type board = cell array array  ;;
```

An iterator In the rest of the program, we often need to iterate a function over all the cells of the mine field. To do it generically, we define the operator `iter_cells` that applies the function `f`, given as an argument, to each cell of the board defined by the configuration `cf`.

```
# let iter_cells cf f =
    for i=0 to cf.nbcols-1 do for j=0 to cf.nbrows-1 do f (i,j) done done ;;
val iter_cells : config -> (int * int -> 'a) -> unit = <fun>
```

This is a good example of a mix between functional and imperative programming styles, as we use a higher order function (a function taking another function as an argument) to iterate a function that operates through side effects (as it returns no value).

Initialization We randomly choose which cells are mines. If c and r are respectively the number of columns and rows of the mine field, and m the number of mines, we need to generate m different numbers between 1 and $c \times r$. We suppose that $m \leq c \times r$ to define the algorithm, but the program using it will need to check this condition.

The straightforward algorithm consists of starting with an empty list, picking a random number and putting it in the list if it is not there already, and repeating this until the

list contains m numbers. We use the following functions from the `Random` and `Sys` modules:

- `Random.int`: *int -> int*, picks a number between 0 and $n-1$ (n is the argument) according to a random number generator;
- `Random.init`: *int -> unit*, initializes the random number generator;
- `Sys.time`: *unit -> float*, returns the number of milliseconds of processor time the program used since it started. This function will be used to initialize the random number generator with a different seed for each game.

The modules containing these functions are described in more details in chapter 8, pages 216 and 234.

The random mine placement function receives the number of cells (`cr`) and the number of mines to place (`m`), and returns a list of linear positions for the `m` mines.

```
# let random_list_mines cr m =
    let cell_list = ref []
    in while (List.length !cell_list) < m do
         let n = Random.int cr in
          if not (List.mem n !cell_list) then cell_list := n :: !cell_list
       done ;
      !cell_list ;;
val random_list_mines : int -> int -> int list = <fun>
```

With such an implementation, there is no upper bound on the number of steps the function takes to terminate. If the random number generator is reliable, we can only insure that the probability it does not terminate is zero. However, all experimental uses of this function have never failed to terminate. Thus, even though it is not guaranteed that it will terminate, we will use it to generate the list of mined cells.

We need to initialize the random number generator so that each run of the game does not use the same mine field. We use the processor time since the beginning of the program execution to initialize the random number generator.

```
# let generate_seed () =
    let t = Sys.time () in
    let n = int_of_float (t*.1000.0)
    in Random.init(n mod 100000) ;;
val generate_seed : unit -> unit = <fun>
```

In practice, a given program very often takes the same execution time, which results in a similar result for `generate_seed` for each run. We ought to use the `Unix.time` function (see chapter 18).

We very often need to know the neighbors of a given cell, during the initialization of the mine field as well as during the game. Thus we write a `neighbors` function. This function must take into account the side and corner cells that have fewer neighbors than the middle ones (function `valid`).

```
# let valid cf (i,j) = i>=0 && i<cf.nbcols && j>=0 && j<cf.nbrows ;;
val valid : config -> int * int -> bool = <fun>
```

```
# let neighbors cf (x,y) =
    let ngb = [x-1,y-1; x-1,y; x-1,y+1; x,y-1; x,y+1; x+1,y-1; x+1,y; x+1,y+1]
    in List.filter (valid cf) ngb ;;
val neighbors : config -> int * int -> (int * int) list = <fun>
```

The initialize_board function creates the initial mine field. It proceeds in four steps:

1. generation of the list of mined cells;
2. creation of a two dimensional array containing different cells;
3. setting of mined cells in the board;
4. computation of the number of mines in neighbor cells for each cell that is not mined.

The function initialize_board uses a few local functions that we briefly describe.

cell_init : creates an initial cell value;

copy_cell_init : puts a copy of the initial cell value in a cell of the board;

set_mined : puts a mine in a cell;

count_mined_adj : computes the number of mines in the neighbors of a given cell;

set_count : updates the number of mines in the neighbors of a cell if it is not mined.

```
# let initialize_board cf =
    let cell_init () = { mined=false; seen=false; flag=false; nbm=0 } in
    let copy_cell_init b (i,j) = b.(i).(j) <- cell_init() in
    let set_mined b n = b.(n / cf.nbrows).(n mod cf.nbrows).mined <- true
    in
    let count_mined_adj b (i,j) =
      let x = ref 0 in
      let inc_if_mined (i,j) = if b.(i).(j).mined then incr x
      in List.iter inc_if_mined (neighbors cf (i,j)) ;
         !x
    in
    let set_count b (i,j) =
      if not b.(i).(j).mined
      then b.(i).(j).nbm <- count_mined_adj b (i,j)
    in
    let list_mined = random_list_mines (cf.nbcols*cf.nbrows) cf.nbmines in
    let board = Array.make_matrix cf.nbcols cf.nbrows (cell_init ())
    in iter_cells cf (copy_cell_init board) ;
       List.iter (set_mined board) list_mined ;
       iter_cells cf (set_count board) ;
       board ;;
val initialize_board : config -> cell array array = <fun>
```

Opening a cell During a game, when the player opens a cell whose neighbors are empty (none contains a mine), he knows that he can open the neighboring cells without risk, and he can keep opening cells as long as he opens cells without any mined neighbor. In order to relieve the player of this boring process (as it is not challenging at all), our Minesweeper opens all these cells itself. To this end, we write the function `cells_to_see` that returns a list of all the cells to open when a given cell is opened.

The algorithm needed is simple to state: if the opened cell has some neighbors that contain a mine, then the list of cells to see consists only of the opened cell; otherwise, the list of cells to see consists of the neighbors of the opened cell, as well as the lists of cells to see of these neighbors. The difficulty is in writing a program that does not loop, as every cell is a neighbor of any of its neighbors. We thus need to avoid processing the same cell twice.
To remember which cells were processed, we use the array of booleans `visited`. Its size is the same as the mine field. The value **`true`** for a cell of this array denotes that it was already visited. We recurse only on cells that were not visited.

We use the auxiliary function `relevant` that computes two sublists from the list of neighbors of a cell. Each one of these lists only contains cells that do not contain a mine, that are not opened, that are not flagged by the player, and that were not visited. The first sublist is the list of neighboring cells who have at least one neighbor containing a mine; the second sublist is the list of neighboring cells whose neighbors are all empty. As these lists are computed, all these cells are marked as visited. Notice that flagged cells are not processed, as a flag is meant to prevent opening a cell.

The local function `cells_to_see_rec` implements the recursive search loop. It takes as an argument the list of cells to visit, updates it, and returns the list of cells to open. This function is called with the list consisting only of the cell being opened, after it is marked as visited.

```
# let cells_to_see bd cf (i,j) =
    let visited = Array.make_matrix cf.nbcols cf.nbrows false in
    let rec relevant = function
        [] → ([],[])
      | ((x,y) as c) :: t →
          let cell=bd.(x).(y)
          in if cell.mined || cell.flag || cell.seen || visited.(x).(y)
             then relevant t
             else let (l1,l2) = relevant t
                  in visited.(x).(y) <- true ;
                     if cell.nbm=0 then (l1,c :: l2) else (c :: l1,l2)
    in
    let rec cells_to_see_rec = function
        [] → []
      | ((x,y) as c) :: t →
           if bd.(x).(y).nbm<>0 then c :: (cells_to_see_rec t)
           else let (l1,l2) = relevant (neighbors cf c)
                in (c :: l1) @ (cells_to_see_rec (l2 @ t))
    in visited.(i).(j) <- true ;
```

```
    cells_to_see_rec [(i,j)] ;;
val cells_to_see :
  cell array array -> config -> int * int -> (int * int) list = <fun>
```

At first sight, the argument of `cells_to_see_rec` may grow between two consecutive calls, although the recursion is based on this argument. It is legitimate to wonder if this function always terminates.
The way the `visited` array is used guarantees that a visited cell cannot be in the result of the `relevant` function. Also, all the cells to visit come from the result of the `relevant` function. As the `relevant` function marks as visited all the cells it returns, it returns each cell at most once, thus a cell may be added to the list of cells to visit at most once. The number of cells being finite, we deduce that the function terminates.

Except for graphics, we are done with our Minesweeper. Let us take a look at the programming style we have used. Mutable structures (arrays and mutable record fields) make us use an imperative style of loops and assignments. However, to deal with auxiliary issues, we use lists that are processed by functions written in a functional style. Actually, the programming style is a consequence of the data structure that it manipulates. The function `cells_to_see` is a good example: it processes lists, and it is natural to write it in a functional style. Nevertheless, we use an array to remember the cells that were already processed, and we update this array imperatively. We could use a purely functional style by using a list of visited cells instead of an array, and check if a cell is in the list to see if it was visited. However, the cost of such a choice is important (looking up an element in a list is linear in the size of the list, whereas accessing an array element takes constant time) and it does not make the program simpler.

Displaying the Minesweeper game

This part depends on the data structures representing the state of the game (see page 177). It consists of displaying the different components of the Minesweeper window, as shown in figure 6.6. To this end, we use the box drawing functions seen on page 126.

The following parameters characterize the components of the graphical window.

```
# let b0 = 3 ;;
# let w1 = 15 ;;
# let w2 = w1 ;;
# let w4 = 20 + 2*b0 ;;
# let w3 = w4*default_config.nbcols + 2*b0 ;;
# let w5 = 40 + 2*b0 ;;
```

```
# let h1 = w1 ;;
# let h2 = 30 ;;
# let h3 = w5+20 + 2*b0 ;;
# let h4 = h2 ;;
# let h5 = 20 + 2*b0 ;;
# let h6 = w5 + 2*b0 ;;
```

We use them to extend the basic configuration of our Minesweeper board (value of type *config*). Below, we define a record type *window_config*. The `cf` field contains the basic configuration. We associate a box with every component of the display: main window

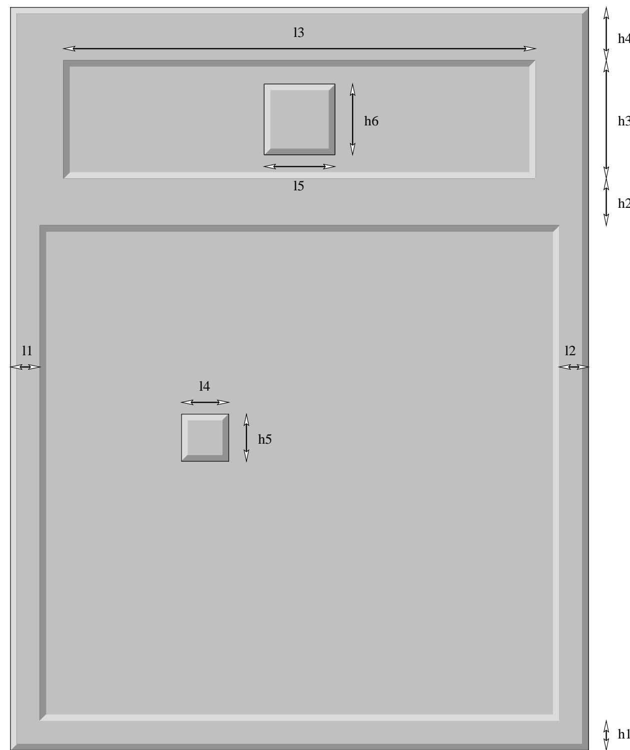


図 6.6: The main window of Minesweeper.

(field `main_box`), mine field (field `field_box`), dialog window (field `dialog_box`) with two sub-boxes (fields `d1_box` and `d2_box`), flagging button (field `flag_box`) and current cell (field `current_box`).

```
# type window_config = {
    cf : config ;
    main_box : box_config ;
    field_box : box_config ;
    dialog_box : box_config ;
    d1_box : box_config ;
    d2_box : box_config ;
    flag_box : box_config ;
    mutable current_box : box_config ;
    cell : int*int → (int*int) ;
    coor : int*int → (int*int)
  } ;;
```

Moreover, a record of type *window_config* contains two functions:

- `cell`: takes the coordinates of a cell and returns the coordinates of the corresponding box;
- `coor`: takes the coordinates of a pixel of the window and returns the coordinates of the corresponding cell.

Configuration We now define a function that builds a graphical configuration (of type *window_config*) according to a basic configuration (of type *config*) and the parameters above. The values of the parameters of some components depend on the value of the parameters of other components. For instance, the global box width depends on the mine field width, which, in turn, depends on the number of columns. To avoid computing the same value several times, we incrementally create the components. This initialization phase of a graphical configuration is always a little tedious when there is no adequate primitive or tool available.

```
# let make_box x y w h bw r =
    { x=x; y=y; w=w; h=h; bw=bw; r=r; b1_col=gray1; b2_col=gray3; b_col=gray2 } ;;
val make_box : int -> int -> int -> int -> int -> relief -> box_config =
  <fun>
# let make_wcf cf =
    let wcols =  b0 + cf.nbcols*w4 + b0
    and hrows =  b0 + cf.nbrows*h5 + b0  in
    let main_box =  let gw = (b0 + w1 + wcols + w2 + b0)
                    and gh = (b0 + h1 + hrows + h2 + h3 + h4 + b0)
                    in make_box 0 0 gw gh b0 Top
    and field_box = make_box w1 h1 wcols hrows b0 Bot  in
    let dialog_box = make_box  ((main_box.w - w3) / 2)
                               (b0+h1+hrows+h2)
                               w3 h3 b0 Bot
    in
    let d1_box = make_box (dialog_box.x + b0) (b0 + h1 + hrows + h2)
                          ((w3-w5)/2-(2*b0)) (h3-(2*b0)) 5 Flat  in
    let flag_box = make_box (d1_box.x + d1_box.w)
                         (d1_box.y + (h3-h6) / 2) w5 h6 b0 Top  in
    let d2_box = make_box (flag_box.x + flag_box.w)
                          d1_box.y d1_box.w d1_box.h 5 Flat  in
    let current_box = make_box 0 0 w4 h5 b0 Top
    in { cf = cf;
         main_box = main_box; field_box=field_box; dialog_box=dialog_box;
         d1_box=d1_box;
         flag_box=flag_box; d2_box=d2_box; current_box = current_box;
         cell = (fun (i,j) → ( w1+b0+w4*i , h1+b0+h5*j)) ;
         coor = (fun (x,y) → ( (x-w1)/w4 , (y-h1)/h5 )) } ;;
val make_wcf : config -> window_config = <fun>
```

Cell display We now need to write the functions to display the cells in their different states. A cell may be open or closed and may contain some information. We always display (the box corresponding with) the current cell in the game configuration (field cc_bcf).

We thus write two functions modifying the configuration of the current cell; one closing it, the other opening it.

```
# let close_ccell wcf i j =
    let x,y = wcf.cell (i,j)
    in wcf.current_box <- {wcf.current_box with x=x; y=y; r=Top} ;;
val close_ccell : window_config -> int -> int -> unit = <fun>
# let open_ccell wcf i j =
    let x,y = wcf.cell (i,j)
    in wcf.current_box <- {wcf.current_box with x=x; y=y; r=Flat} ;;
val open_ccell : window_config -> int -> int -> unit = <fun>
```

Depending on the game phase, we may need to display some information on the cells. We write, for each case, a specialized function.

- Display of a closed cell:

```
# let draw_closed_cc wcf i j =
   close_ccell wcf i j;
   draw_box wcf.current_box ;;
val draw_closed_cc : window_config -> int -> int -> unit = <fun>
```

- Display of an opened cell with its number of neighbor mines:

```
# let draw_num_cc wcf i j n =
    open_ccell wcf i j ;
    draw_box wcf.current_box ;
    if n<>0 then draw_string_in_box Center (string_of_int n)
                                   wcf.current_box Graphics.white ;;
val draw_num_cc : window_config -> int -> int -> int -> unit = <fun>
```

- Display of a cell containing a mine:

```
# let draw_mine_cc wcf i j =
    open_ccell wcf i j ;
    let cc = wcf.current_box
    in draw_box wcf.current_box ;
       Graphics.set_color Graphics.black ;
       Graphics.fill_circle (cc.x+cc.w/2) (cc.y+cc.h/2) (cc.h/3) ;;
val draw_mine_cc : window_config -> int -> int -> unit = <fun>
```

- Display of a flagged cell containing a mine:

```
# let draw_flag_cc wcf i j =
    close_ccell wcf i j ;
    draw_box wcf.current_box ;
```

```
    draw_string_in_box Center "!" wcf.current_box Graphics.blue ;;
val draw_flag_cc : window_config -> int -> int -> unit = <fun>
```

- Display of a wrongly flagged cell:

```
# let draw_cross_cc wcf i j =
   let x,y = wcf.cell (i,j)
   and w,h = wcf.current_box.w, wcf.current_box.h in
   let a=x+w/4 and b=x+3*w/4
   and c=y+h/4 and d=y+3*h/4
   in Graphics.set_color Graphics.red ;
      Graphics.set_line_width 3 ;
      Graphics.moveto a d ; Graphics.lineto b c ;
      Graphics.moveto a c ; Graphics.lineto b d ;
      Graphics.set_line_width 1 ;;
val draw_cross_cc : window_config -> int -> int -> unit = <fun>
```

During the game, the choice of the display function to use is done by:

```
# let draw_cell wcf bd i j =
    let cell = bd.(i).(j)
    in match (cell.flag, cell.seen , cell.mined ) with
           (true,_,_)  → draw_flag_cc wcf i j
         | (_,false,_) → draw_closed_cc wcf i j
         | (_,_,true)  → draw_mine_cc wcf i j
         | _           → draw_num_cc wcf i j cell.nbm ;;
val draw_cell : window_config -> cell array array -> int -> int -> unit =
  <fun>
```

A specialized function displays all the cells at the end of the game. It is slightly different from the previous one as all the cells are taken as opened. Moreover, a red cross indicates the empty cells where the player wrongly put a flag.

```
# let draw_cell_end wcf bd i j =
    let cell = bd.(i).(j)
    in match (cell.flag, cell.mined ) with
           (true,true) → draw_flag_cc wcf i j
         | (true,false) → draw_num_cc wcf i j cell.nbm; draw_cross_cc wcf i j
         | (false,true) → draw_mine_cc wcf i j
         | (false,false) → draw_num_cc wcf i j cell.nbm ;;
val draw_cell_end : window_config -> cell array array -> int -> int -> unit =
  <fun>
```

Display of the other components The state of the flagging mode is indicated by a box that is either at the bottom or on top and that contain either the word `ON` or `OFF`:

```
# let draw_flag_switch wcf on =
```

```
     if on  then wcf.flag_box.r <- Bot  else wcf.flag_box.r <- Top ;
     draw_box wcf.flag_box ;
     if on  then draw_string_in_box Center "ON" wcf.flag_box Graphics.red
     else draw_string_in_box Center "OFF" wcf.flag_box Graphics.blue ;;
val draw_flag_switch : window_config -> bool -> unit = <fun>
```

We display the purpose of the flagging button above it:

```
# let draw_flag_title wcf =
     let m = "Flagging" in
     let w,h = Graphics.text_size m in
     let x = (wcf.main_box.w-w)/2
     and y0 = wcf.dialog_box.y+wcf.dialog_box.h in
     let y = y0+(wcf.main_box.h-(y0+h))/2
     in Graphics.moveto x y ;
        Graphics.draw_string m ;;
val draw_flag_title : window_config -> unit = <fun>
```

During the game, the number of empty cells left to be opened and the number of cells to flag are displayed in the dialog box, to the left and right of the flagging mode button.

```
# let print_score wcf nbcto nbfc =
     erase_box wcf.d1_box ;
     draw_string_in_box Center (string_of_int nbcto) wcf.d1_box Graphics.blue ;
     erase_box wcf.d2_box ;
     draw_string_in_box Center (string_of_int (wcf.cf.nbmines-nbfc)) wcf.d2_box
      ( if nbfc>wcf.cf.nbmines then Graphics.red else Graphics.blue ) ;;
val print_score : window_config -> int -> int -> unit = <fun>
```

To draw the initial mine field, we need to draw (number of rows) × (number of columns) times the same closed cell. It is always the same drawing, but it may take a long time, as it is necessary to draw a rectangle as well as four trapezoids. To speed up this initialization, we draw only one cell, take the bitmap corresponding to this drawing, and paste this bitmap into every cell.

```
# let draw_field_initial wcf =
     draw_closed_cc wcf 0 0 ;
     let cc = wcf.current_box  in
     let bitmap = draw_box cc ; Graphics.get_image cc.x cc.y cc.w cc.h  in
     let draw_bitmap (i,j) = let x,y=wcf.cell (i,j)
                             in Graphics.draw_image bitmap x y
     in iter_cells wcf.cf draw_bitmap  ;;
val draw_field_initial : window_config -> unit = <fun>
```

At the end of the game, we open the whole mine field while putting a red cross on cells wrongly flagged:

```
# let draw_field_end wcf bd =
    iter_cells wcf.cf (fun (i,j) → draw_cell_end wcf bd i j) ;;
val draw_field_end : window_config -> cell array array -> unit = <fun>
```

Finally, the main display function called at the beginning of the game opens the graphical context and displays the initial state of all the components.

```
# let open_wcf wcf =
    Graphics.open_graph ( " " ^ (string_of_int wcf.main_box.w) ^ "x" ^
                          (string_of_int wcf.main_box.h)              ) ;
    draw_box wcf.main_box ;
    draw_box wcf.dialog_box ;
    draw_flag_switch wcf false ;
    draw_box wcf.field_box ;
    draw_field_initial wcf ;
    draw_flag_title wcf ;
    print_score wcf ((wcf.cf.nbrows*wcf.cf.nbcols)-wcf.cf.nbmines) 0 ;;
val open_wcf : window_config -> unit = <fun>
```

Notice that all the display primitives are parameterized by a graphical configuration of type *window_config*. This makes them independent of the layout of the components of our Minesweeper. If we wish to modify the layout, the code still works without any modification, only the configuration needs to be updated.

Interaction with the player

We now list what the player may do:

- he may click on the mode box to change mode (opening or flagging),
- he may click on a cell to open it or flag it,
- he may hit the 'q' key to quit the game.

Recall that a Graphic event (*Graphics.event*) must be associated with a record (*Graphics.status*) that contains the current information on the mouse and keyboard when the event occurs. An interaction with the mouse may happen on the mode button, or on a cell of the mine field. Every other mouse event must be ignored. In order to differentiate these mouse events, we create the type:

```
# type clickon = Out | Cell of (int*int) | SelectBox ;;
```

Also, pressing the mouse button and releasing it are two different events. For a click to be valid, we require that both events occur on the same component (the flagging mode button or a cell of the mine field).

```
# let locate_click wcf st1 st2 =
    let clickon_of st =
      let x = st.Graphics.mouse_x and y = st.Graphics.mouse_y
      in if x>=wcf.flag_box.x && x<=wcf.flag_box.x+wcf.flag_box.w &&
```

```
            y>=wcf.flag_box.y && y<=wcf.flag_box.y+wcf.flag_box.h
          then SelectBox
          else let (x2,y2) = wcf.coor (x,y)
               in if x2>=0 && x2<wcf.cf.nbcols && y2>=0 && y2<wcf.cf.nbrows
                  then Cell (x2,y2) else Out
     in
     let r1=clickon_of st1 and r2=clickon_of st2
     in if r1=r2 then r1 else Out ;;
val locate_click :
  window_config -> Graphics.status -> Graphics.status -> clickon = <fun>
```

The heart of the program is the event waiting and processing loop defined in the function `loop`. It is similar to the function `skel` described page 133, but specifies the mouse events more precisely. The loop ends when:

- the player presses the `q` or `Q` key, meaning that he wants to end the game;
- the player opens a cell containing a mine, then he loses;
- the player has opened all the cell that are empty, then he wins the game.

We gather in a record of type *minesw_cf* the information useful for the interface:

```
# type minesw_cf =
    { wcf : window_config; bd : cell array array;
      mutable nb_flagged_cells : int;
      mutable nb_hidden_cells : int;
      mutable flag_switch_on : bool } ;;
```

The meaning of the fields is:

- `wcf`: the graphical configuration;
- `bd`: the board;
- `flag_switch_on`: a boolean indicating whether flagging mode or opening mode is on;
- `nb_flagged_cells`: the number of flagged cells;
- `nb_hidden_cells`: the number of empty cells left to open;

The main loop is implemented this way:

```
# let loop d f_init f_key f_mouse f_end =
    f_init ();
    try
      while true do
        let st = Graphics.wait_next_event
                   [Graphics.Button_down;Graphics.Key_pressed]
        in if st.Graphics.keypressed  then f_key st.Graphics.key
           else let st2 = Graphics.wait_next_event [Graphics.Button_up]
                in f_mouse (locate_click d.wcf st st2)
      done
    with End → f_end ();;
```

```
val loop :
  minesw_cf ->
  (unit -> 'a) -> (char -> 'b) -> (clickon -> 'b) -> (unit -> unit) -> unit =
  <fun>
```

The initialization function, cleanup function and keyboard event processing function are very simple.

```
# let d_init d () = open_wcf d.wcf
  let d_end  () = Graphics.close_graph()
  let d_key c = if c='q' || c='Q' then raise End;;
val d_init : minesw_cf -> unit -> unit = <fun>
val d_end : unit -> unit = <fun>
val d_key : char -> unit = <fun>
```

However, the mouse event processing function requires the use of some auxiliary functions:

- `flag_cell`: when clicking on a cell with flagging mode on.
- `ending`: when ending the game. The whole mine field is revealed, we display a message indicating whether the game was won or lost, and we wait for a mouse or keyboard event to quit the application.
- `reveal`: when clicking on a cell with opening mode on (*i.e.* flagging mode off).

```
# let flag_cell d i j =
    if d.bd.(i).(j).flag
    then ( d.nb_flagged_cells <- d.nb_flagged_cells -1;
           d.bd.(i).(j).flag <- false )
    else ( d.nb_flagged_cells <- d.nb_flagged_cells +1;
           d.bd.(i).(j).flag <- true );
    draw_cell d.wcf d.bd i j;
    print_score d.wcf d.nb_hidden_cells d.nb_flagged_cells;;
val flag_cell : minesw_cf -> int -> int -> unit = <fun>

# let ending d str =
    draw_field_end d.wcf d.bd;
    erase_box  d.wcf.flag_box;
    draw_string_in_box Center str d.wcf.flag_box Graphics.black;
    ignore(Graphics.wait_next_event
             [Graphics.Button_down;Graphics.Key_pressed]);
    raise End;;
val ending : minesw_cf -> string -> 'a = <fun>

# let reveal d i j =
    let reveal_cell (i,j) =
      d.bd.(i).(j).seen <- true;
```

```
      draw_cell d.wcf d.bd i j;
      d.nb_hidden_cells <- d.nb_hidden_cells -1
    in
      List.iter reveal_cell (cells_to_see d.bd d.wcf.cf (i,j));
      print_score d.wcf d.nb_hidden_cells d.nb_flagged_cells;
      if d.nb_hidden_cells = 0 then ending d "WON";;
val reveal : minesw_cf -> int -> int -> unit = <fun>
```

The mouse event processing function matches a value of type *clickon*.

```
# let d_mouse d click = match click with
      Cell (i,j) →
        if d.bd.(i).(j).seen then ()
        else if d.flag_switch_on then flag_cell d i j
        else if d.bd.(i).(j).flag then ()
        else if d.bd.(i).(j).mined then ending d "LOST"
        else reveal d i j
    | SelectBox →
        d.flag_switch_on <- not d.flag_switch_on;
        draw_flag_switch d.wcf d.flag_switch_on
    | Out → () ;;
val d_mouse : minesw_cf -> clickon -> unit = <fun>
```

To create a game configuration, three parameters are needed: the number of columns, the number of rows, and the number of mines.

```
# let create_minesw nb_c nb_r nb_m =
    let nbc = max default_config.nbcols nb_c
    and nbr = max default_config.nbrows nb_r in
    let nbm = min (nbc*nbr) (max 1 nb_m) in
    let cf = { nbcols=nbc ; nbrows=nbr ; nbmines=nbm } in
    generate_seed () ;
    let wcf = make_wcf cf in
    {  wcf = wcf ;
       bd = initialize_board wcf.cf;
       nb_flagged_cells = 0;
       nb_hidden_cells = cf.nbrows*cf.nbcols-cf.nbmines;
       flag_switch_on = false } ;;
val create_minesw : int -> int -> int -> minesw_cf = <fun>
```

The launch function creates a configuration according to the numbers of columns, rows, and mines, before calling the main event processing loop.

```
# let go nbc nbr nbm =
    let d = create_minesw nbc nbr nbm in
    loop d (d_init d) d_key (d_mouse d) (d_end);;
val go : int -> int -> int -> unit = <fun>
```

The function call `go 10 10 10` builds and starts a game of the same size as the one depicted in figure 6.5.

Exercises

This program can be built as a standalone executable program. Chapter 7 explains how to do this. Once it is done, it is useful to be able to specify the size of the game on the command line. Chapter 8 describes how to get command line arguments in an Objective Caml program, and applies it to our minesweeper (see page 236).

Another possible extension is to have the machine play to discover the mines. To do this, one needs to be able to find the safe moves and play them first, then compute the probabilities of presence of a mine and open the cell with the smallest probability.

Part II

Development Tools

We describe the set of elements of the environment included in the language distribution. There one finds different compilers, numerous libraries, program analysis tools, lexical and syntactic analysis tools, and an interface with the C language.

Objective Caml is a compiled language offering two types of code generation:

1. *bytecode* to be executed by a *virtual machine*;
2. *native code* to be executed directly by a microprocessor.

The Objective Caml toplevel uses bytecode to execute the phrases submitted to it. It constitutes the primary development aid, offering the possibility of rapid typing, compilation and testing of function definitions. Moreover, it offers a trace mechanism visualizing parameter values and return values of functions.

The other usual development tools are supplied by the distribution as well: file dependency computation, debugging and profiling. The debugger allows one to execute programs step-by-step, use breakpoints and inspect values. The profiling tool gives measurements of the number of calls or the amount of time spent in a particular function or a particular part of the code. These two tools are only available for Unix platforms.

The richness of a language derives from its core but also from the libraries, sets of reusable programs, which come with it. Objective Caml is no exception to the rule. We have already portrayed to a large extent the graphical library that comes with the distribution. There are many others which we will describe. Libraries bring new functionality to the language, but they are not without drawbacks. In particular, they can present some difficulty vis-a-vis the type discipline.

However rich a language's set of libraries may be, it will always be necessary that it be able to communicate with another language. The Objective Caml distribution includes an interface with the C language allowing Objective Caml to call C functions or be called by them. The difficulty of understanding and implementing this interface lies in the fact that the memory models of Objective Caml and C are different. The essential reason for this difference is that an Objective Caml program includes a garbage collection mechanism.

C as well as Objective Caml allow dynamic memory allocation, and thus fine control over space according to the needs of a program. This only makes sense if unused space can be reclaimed for other use during the course of execution. Garbage collection frees the programmer from responsibility for managing deallocation, a frequent source of execution errors. This feature constitutes one of the safety elements of the Objective Caml language.

However, this mechanism has an impact on the representation of data. Also, knowledge of the guiding principles of memory management is indispensable in order to use communication between the Objective Caml world and the C world correctly.

Chapter 7 presents the basic elements of the Objective Caml system: virtual machine, compilers, and execution library. It describes the language's different compilation modes and compares their portability and efficiency.

Chapter 8 gives a bird's-eye view of the set of predefined types, functions, and exceptions that come with the system distribution. It does not do away with the need to read the reference manual ([LRVD99]) which describes these libraries very well. On the contrary it focuses on the new functionalities supplied by some of them. In particular we may mention output formatting, persistence of values and interfacing with the operating system.

Chapter 9 presents different garbage collection methods in order to then describe the mechanism used by Objective Caml.

Chapter 10 presents debugging tools for Objective Caml programs. Although still somewhat frustrating in some respects, these tools quite often allow one to understand why a program does not work.

Chapter 11 describes the language's different approaches to lexical and syntactic analysis problems: a regular expression library, the `ocamlex` and `ocamlyacc` tools, but also the use of streams.

Chapter 12 describes the interface with the C language. It is no longer possible for a language to be completely isolated from other languages. This interface lets an Objective Caml program call a C function, while passing it values from the Objective Caml world, and vice-versa. The main difficulty with this interface stems from the memory model. For this reason it is recommended that you read the 9 chapter beforehand.

Chapter 13 covers two applications: an improved graphics library based on a hierarchical model of graphical components inspired by the JAVA AWT[2]; and a classic program to find least-cost paths in a graph using our new graphical interface as well as a cache memory mechanism.

2. *Abstract Windowing Toolkit*

7

Compilation and Portability

The transformation from human readable source code to an executable requires a number of steps. Together these steps constitute the process of **compilation**. The compilation process produces an abstract syntax tree (for an example, see page 159) and a sequence of instructions for a cpu or virtual machine. In Objective Caml, the product of compilation is linked with the Objective Caml **runtime library**. The library is provided with the compiler distribution and is adapted to different host environments (operating system and CPU). The runtime library contains primitive functions such as operations over numbers, the interface to the operating system, and memory management.

Objective Caml has two compilers. The first compiler produces **bytecode** for the Objective Caml virtual machine. The second compiler generates instructions for a number of "real" processors, such as the Intel, Motorola, SPARC, HP-PA, Power-PC and Alpha CPUs. The Objective Caml bytecode compiler produces compact portable code, while the native-code compiler generates high performance architecture dependent code. The Objective Caml toplevel system, which appeared in the first part of this book, uses the bytecode compiler; each user input is compiled and executed in the symbolic environment defined by the current interactive session.

Chapter Overview

This chapter presents the different ways to compile an Objective CAML program and compares their portability and efficiency. The first section explains the different steps of Objective Caml compilation. The second section describes the different types of compilation and the syntax for the production of executables. The third section shows how to construct standalone executables - programs which are independent of an installation of the Objective Caml system. Finally the fourth section compares the different types of compilation with respect to portability and efficiency of execution.

Steps of Compilation

An executable file is obtained by translating and linking as described in figure 7.1.

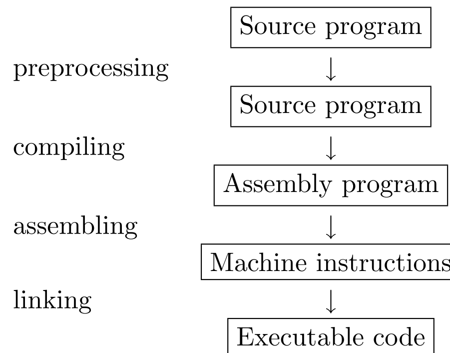


図 7.1: Steps in the production of an executable.

To start off, preprocessing replaces certain pieces of text by other text according to a system of macros. Next, compilation translates the source program into assembly instructions, which are then converted to machine instructions. Finally, the linking process establishes a connection to the operating system for primitives. This includes adding the runtime library, which mainly consists of memory management routines.

The Objective Caml Compilers

The code generation phases of the Objective Caml compiler are detailed in figure 7.2. The internal representation of the code generated by the compiler is called an intermediate language (IL).

The lexical analysis stage transforms a sequence of characters to a sequence of lexical elements. These lexical entities correspond principally to integers, floating point numbers, characters, strings of characters and identifiers. The message `Illegal character` might be generated by this analysis.

The parsing stage constructs a syntax tree and verifies that the sequence of lexical elements is correct with respect to the grammar of the language. The message `Syntax error` indicates that the phrase analyzed does not follow the grammar of the language.

The semantic analysis stage traverses the syntax tree, checking another aspect of program correctness. The analysis consists principally of type inference, which if successful, produces the *most general type* of an expression or declaration. Type error messages may occur during this phase. This stage also detects whether any members of a sequence are not of type *unit*. Other warnings may result, including pattern matching analy-

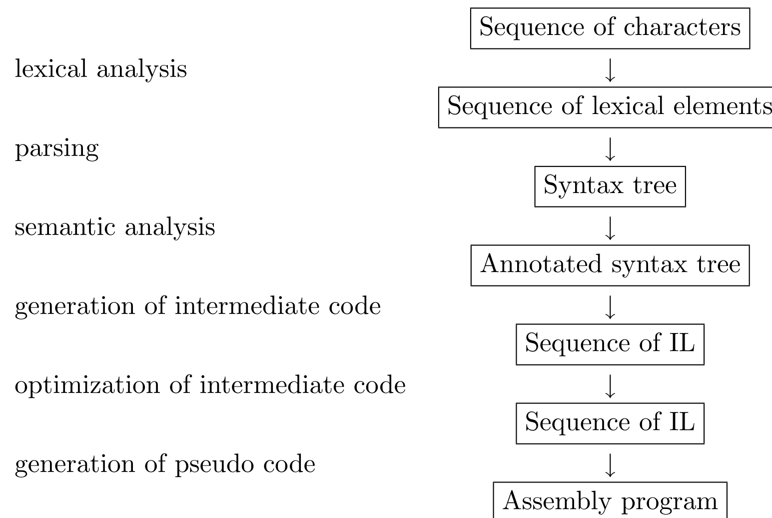


図 7.2: Compilation stages.

sis (e.g `pattern matching is not exhaustive`, `part of pattern matching will not be used`).

Generation and the optimization of intermediate code does not produce errors or warning messages.

The final step in the compilation process is the generation of a program binary. Details differ from compiler to compiler.

Description of the Bytecode Compiler

The Objective Caml virtual machine is called *Zinc* (*"Zinc Is Not Caml"*). Originally created by Xavier Leroy, *Zinc* is described in ([Ler90]). *Zinc*'s name was chosen to indicate its difference from the first implementation of Caml on the virtual machine CAM (Categorical Abstract Machine, see [CCM87]).

Figure 7.3 depicts the bytecode compiler. The first part of this figure shows the Zinc machine interpreter, linked to the runtime library. The second part corresponds to the Objective Caml bytecode compiler which produces instructions for the Zinc machine. The third part contains the set of libraries that come with the compiler. They will be described in Chapter 8. Standard compiler graphical notation is used for describing the components in figure 7.3. A simple box represents a file written in the language indicated in the box. A double box represents the interpretation of a language by a program written in another language. A triple box indicates that a source language is compiled to a machine language by using a compiler written in a third language. Figure 7.4 gives the legend of each box.

The legend of figure 7.3 is as follows:

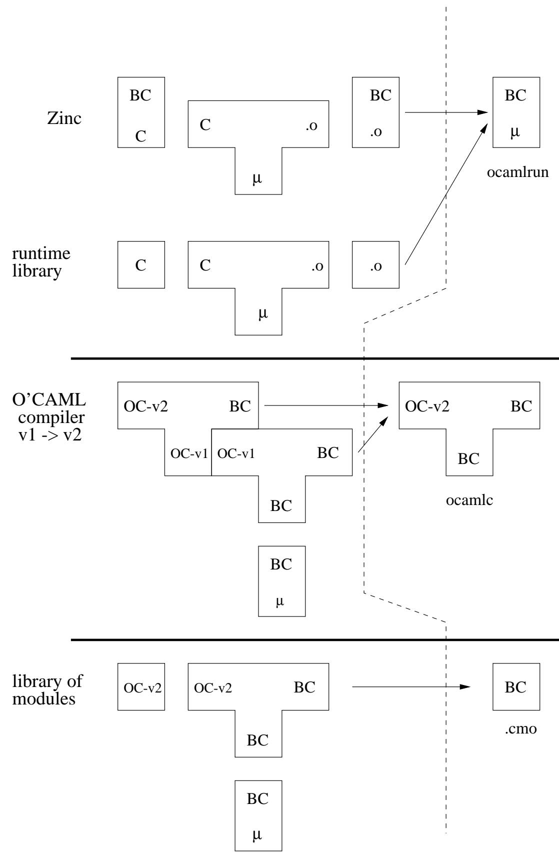


図 7.3: Virtual machine.

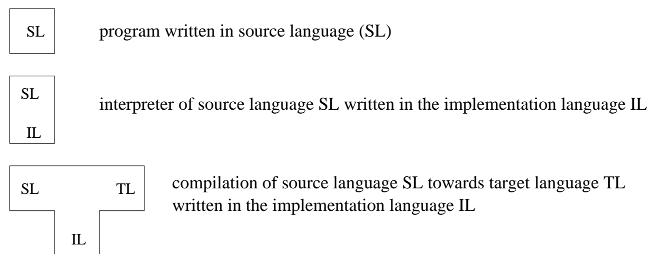


図 7.4: Graphical notation for interpreters and compilers.

- BC : Zinc bytecode;
- C : C code;
- .o : object code
- μ : micro-processor;

- OC (v1 or v2) : Objective Caml code.

注意

> The majority of the Objective Caml compiler is written in Objective Caml. The second part of figure 7.3 shows how to pass from version v1 of a compiler to version v2.

Compilation

The distribution of a language depends on the processor and the operating system. For each architecture, a distribution of Objective Caml contains the toplevel system, the bytecode compiler, and in most cases a native compiler.

Command Names

The figure 7.5 shows the command names of the different compilers in the various Objective Caml distributions. The first four commands are available for all distributions.

`ocaml`	toplevel loop
`ocamlrun`	bytecode interpreter
`ocamlc`	bytecode batch compiler
`ocamlopt`	native code batch compiler
`ocamlc.opt`	optimized bytecode batch compiler
`ocamlopt.opt`	optimized native code batch compiler
`ocamlmktop`	new toplevel constructor

図 7.5: Commands for compiling.

The optimized compilers are themselves compiled with the Objective Caml native compiler. They compile faster but are otherwise identical to their unoptimized counterparts.

Compilation Unit

A compilation unit corresponds to the smallest piece of an Objective Caml program that can be compiled. For the interactive system, the unit of compilation corresponds to a phrase of the language. For the batch compiler, the unit of compilation is two files: the source file, and the interface file. The interface file is optional - if it does not exist, then all global declarations in the source file will be visible to other compilation units. The construction of interface files is described in the chapter on module programming (第 14 章参照). The two file types (source and interface) are differentiated by separate file extensions.

Naming Rules for File Extensions

Figure 7.6 presents the extensions of different files used for Objective CAML and C programs.

extension	meaning
`.ml`	source file
`.mli`	interface file
`.cmo`	object file (bytecode)
`.cma`	library object file (bytecode)
`.cmi`	compiled interface file
`.cmx`	object file (native)
`.cmxa`	library object file (native)
`.c`	C source file
`.o`	C object file (native)
`.a`	C library object file (native)

図 7.6: File extensions.

The files `example.ml` and `example.mli` form a compilation unit. The compiled interface file (`example.cmi`) is used for both the bytecode and native code compiler. The C language related files are used when integrating C code with Objective Caml code. (第 12 章参照).

The Bytecode Compiler

The general form of the batch compiler commands are:

`command` *options file_name*

For example:

```
ocamlc -c example.ml
```

The command-line options for both the native and bytecode compilers follow typical Unix conventions. Each option is prefixed by the character `-`. File extensions are interpreted in the manner described by figure 7.6. In the above example, the file `example.ml` is considered an Objective Caml source file and is compiled. The compiler will produce the files `example.cmo` and `example.cmi`. The option `-c` informs the compiler to generate individual object files, which may be linked at a later time. Without this option, the compiler will produce an executable file named `a.out`.

The table in figure 7.7 describes the principal options of the bytecode compiler. The table in figure 7.8 indicates other possible options.

Principal options	
`-a`	construct a runtime library
`-c`	compile without linking
`-o` *name_of_executable*	specify the name of the executable
`-linkall`	link with all libraries used
`-i`	display all compiled global declarations
`-pp` *command*	uses *command* as preprocessor
`-unsafe`	turn off index checking
`-v`	display the version of the compiler
`-w` *list*	choose among the *list* the level of warning message (see fig. 7.9)
`-impl` *file*	indicate that *file* is a Caml source (.ml)
`-intf` *file*	indicate that *file* is a Caml interface (.mli)
`-I` *directory*	add *directory* in the list of directories

図 7.7: Principal options of the bytecode compiler.

Other options	
light process	`-thread` (第 19 章参照, page 601)
linking	`-g`, `-noassert` (第 10 章参照, page 273)
standalone executable	`-custom`, `-cclib`, `-ccopt`, `-cc` (see page 207)
runtime	`-make-runtime` , `-use-runtime`
C interface	`-output-obj` (第 12 章参照, page 317)

図 7.8: Other options for the bytecode compiler.

To display the list of bytecode compiler options, use the option `-help`.

The different levels of warning message are described in figure 7.9. A message level is a switch (enable/disable) represented by a letter. An upper case letter activates the level and a lower case letter disables it.

Principal levels	
`A/a`	enable/disable all messages
`F/f`	partial application in a sequence
`P/p`	for incomplete pattern matching
`U/u`	for missing cases in pattern matching
`X/x`	enable/disable all other messages
for hidden object	`M/m` and `V/v` (see chapter 15)

図 7.9: Description of compilation warnings.

By default, the highest level (A) is chosen by the compiler.

Example usage of the bytecode compiler is given in figure 7.10.

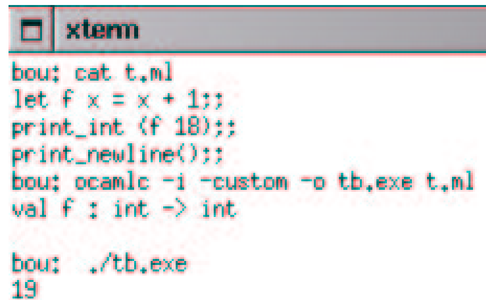


図 7.10: Session with the bytecode compiler.

Native Compiler

The native compiler has behavior similar to the bytecode compiler, but produces different types of files. The compilation options are generally the same as those described in figures 7.7 and 7.8. It is necessary to take out the options related to *runtime* in figure 7.8. Options specific to the native compiler are given in figure 7.11. The different *warning* levels are same.

`-compact`	optimize the produced code for space
`-S`	keeps the assembly code in a file
`-inline` *level*	set the aggressiveness of inlining

図 7.11: Options specific to the native compiler.

Inlining is an elaborated version of macro-expansion in the preprocessing stage. For functions whose arguments are fixed, inlining replaces each function call with the body of the function called. Several different calls produce several copies of the function body. Inlining avoids the overhead that comes with function call setup and return, at the expense of object code size. Principal inlining levels are:

- 0 : The expansion will be done only when it will not increase the size of the object code.
- 1 : This is the default value; it accepts a light increase on code size.
- $n > 1$: Raise the tolerance for growth in the code. Higher values result in more inlining.

Toplevel Loop

The toplevel loop provides only two command line options.

- `-I` *directory*: adds the indicated directory to the list of search paths for compiled source files.
- `-unsafe`: instructs the compiler not to do bounds checking on array and string accesses.

The toplevel loop provides several directives which can be used to interactively modify its behavior. They are described in figure 7.12. All these directives begin with the character `#` and are terminated by `;;`.

`#quit ;;`	quit from the toplevel interaction
`#directory` *directory* `;;`	add the directory to the search path
`#cd` *directory* `;;`	change the working directory
`#load` *object_file* `;;`	load an object file (`.cmo`)
`#use` *source_file* `;;`	compile and load a source file
`#print_depth` *depth* `;;`	modify the depth of printing
`#print_length` *width* `;;`	modify the length of printing
`#install_printer` *function* `;;`	specify a printing function
`#remove_printer` *function* `;;`	remove a printing function
`#trace` *function* `;;`	trace the arguments of the function
`#untrace` *function* `;;`	stop tracing the function
`#untrace_all ;;`	stop all tracing

図 7.12: Toplevel loop directives.

The directives dealing with directories respect the conventions of the operating system used.

The loading directives do not have exactly the same behavior. The directive `#use` reads the source file as if it was typed directly in the toplevel loop. The directive `#load` loads the file with the extension `.cmo`. In the later case, the global declarations of this file are not directly accessible. If the file `example.ml` contains the global declaration `f`, then once the bytecode is loaded (`#load "example.cmo";;`), it is assumed that the value of `f` could be accessed by `Example.f`, where the first letter of the file is capitalized. This notation comes from the module system of Objective Caml (see chapter 14, page 409).

The directives for the depth and width of printing are used to control the display of values. This is useful when it is necessary to display the contents of a value in detail.

The directives for printer redefinition are used to install or remove a user defined printing function for values of a specified type. In order to integrate these printer functions

into the default printing procedure, it is necessary to use the `Format` library(第 8 章参照) for the definition.

The directives for tracing arguments and results of functions are particularly useful for debugging programs. They will be discussed in the chapter on program analysis (第 10 章参照).

Figure 7.13 shows a session in the toplevel loop.

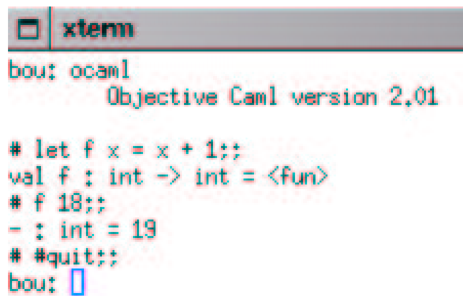


図 7.13: Session with the toplevel loop.

Construction of a New Interactive System

The command `ocamlmktop` can be used to construct a new toplevel executable which has specific library modules loaded by default. For example, `ocamlmktop` is often used for pulling native object code libraries (typically written in C) into a new toplevel.

`ocamlmktop` options are a subset of those used by the bytecode compiler (`ocamlc`):

> `-cclib` *libname*, `-ccopt` *option*, `-custom`, `-I` *directory* `-o` *executable_name*

The chapter on graphics programming (第 5 章参照, page 117) uses this command for constructing a toplevel system containing the `Graphics` library in the following manner:

```
ocamlmktop -custom -o mytoplevel graphics.cma -cclib \
           -I/usr/X11/lib -cclib -lX11
```

This command constructs an executable with the name `mytoplevel`, containing the bytecode library `graphics.cma`. This standalone executable (`-custom`, see the following section) will be linked to the library `X11` (`libX11.a`) which in turn will be looked up in the path `/usr/X11/lib`.

Standalone Executables

A standalone executable is a program that does not depend an Objective Caml installation to run. This facilitates the distribution of binary applications and robustness against runtime library changes across Objective Caml versions.

The Objective Caml native compiler produces standalone executables by default. But without the `-custom` option, the bytecode compiler produces an executable which requires the bytecode interpreter *ocamlrun*. Imagine the file `example.ml` is as follows:

```
let f x = x + 1;;
print_int (f 18);;
print_newline();;
```

Then the following command produces the (approximately 8k) file `example.exe`:

```
ocamlc -o example.exe example.ml
```

This file can be executed by the Objective Caml bytecode interpreter:

```
$ ocamlrun example.exe
19
```

The interpreter executes the Zinc machine instructions contained in the file `example.exe`.

Under Unix, the first line of the file `example.exe` contains the location of the interpreter, for example:

```
#!/usr/local/bin/ocamlrun
```

This means the file can be executed directly (without using `ocamlrun`. Like a shell-script, executing the file in turn runs the program specified on the first line, which is then used to interpret the remainder of the file. If `ocamlrun` can't be found, execution will fail and the error message `Command not found` will be displayed.

The same compilation with the option `-custom` produces a standalone executable with name `exauto.exe`:

```
ocamlc -custom -o exauto.exe example.ml
```

This time the file is about 85K, as it contains the Zinc interpreter as well as the program bytecode. This file can be executed directly or copied to another machine (using the same CPU/Operating System) for execution.

Portability and Efficiency

One reason to compile to an abstract machine is to produce an executable independent of the architecture of the real machine where it runs. A native compiler will produce more efficient code, but the binary can only be executed on the architecture it was compiled for.

Standalone Files and Portability

To produce a standalone executable, the bytecode compiler links the bytecode object file `example.cmo` with the runtime library, the bytecode interpreter and some C code. It is assumed that there is a C compiler on the host system. The inclusion of machine code means that stand-alone bytecode executables are not portable to other systems or other architectures.

This is not the case for the non-standalone version. Since the Zinc machine is not included, the only things generated are the platform independent bytecode instructions. Bytecode programs will run on any platform that has the interpreter. `Ocamlrun` is part of the default Objective Caml distribution for Sparc running Solaris, Intel running Windows, etc. It is always preferable to use the same version of interpreter and compiler.

The portability of bytecode object files makes it possible to directly distribute Objective Caml libraries in bytecode form.

Efficiency of Execution

The bytecode compiler produces a sequence of instructions for the Zinc machine, which at the moment of the execution, will be interpreted by `ocamlrun`. Interpretation has a moderately negative linear effect on speed of execution. It is possible to view Zinc's bytecode interpretation as a big pattern matching machine (matching `match` ... `with`) where each instruction is a trigger and the computation branch modifies the stack and the counter (address of the next instruction).

Without testing all parts of the language, the following small example which computes Fibonacci numbers shows the difference in execution time between the bytecode compiler and the native compiler. Let the program `fib.ml` as follows:

```
let rec fib n =
  if n < 2 then 1
  else (fib (n-1)) + (fib(n-2));;
```

and the following program `main.ml` as follows:

```
for i = 1 to 10 do
```

```
  print_int (Fib.fib 30);
  print_newline()
done;;
```

Their compilation is as follows:

```
$ ocamlc -o fib.exe fib.ml main.ml
$ ocamlopt -o fibopt.exe fib.ml main.ml
```

These commands produce two executables: `fib.exe` and `fibopt.exe`. Using the Unix command time in Pentium 350 under Linux, we get the following data:

`fib.exe` (bytecode)	`fibopt.exe` (native)
7 s	1 s

This corresponds to a factor 7 between the two versions of the same program. This program does not test all characteristics of the language. The difference depends heavily on the type of application, and is typically much smaller.

Exercises

Creation of a Toplevel and Standalone Executable

Consider again the Basic interpreter. Modify it to make a new toplevel.

1. Split the Basic application into 4 files, each with the extension `.ml`. The files will be organized like this: abstract syntax (`syntax.ml`), printing (`pprint.ml`), parsing (`alexsynt.ml`) and evaluation of instructions (`eval.ml`). The head of each file should contain the open statements to load the modules required for compilation.
2. Compile all files separately.
3. Add a file `mainbasic.ml` which contains only the statement for calling the main function.
4. Create a new toplevel with the name `topbasic`, which starts the Basic interpreter.
5. Create a standalone executable which runs the Basic interpreter.

Comparison of Performance

Try to compare the performance of code produced by the bytecode compiler and by the native compiler. For this purpose, write an application for sorting lists and arrays.

1. Write a polymorphic function for sorting lists. The order relation should be passed as an argument to the sort function. The sort algorithm can be selected by the reader. For example: bubble sort, or quick sort. Write this function as `sort.ml`.
2. Create the main function in the file `trilist.ml`, which uses the previous function and applies it to a list of integers by sorting it in increasing order, then in decreasing order.
3. Create two standalone executables - one with the bytecode compiler, and another with the native compiler. Measure the execution time of these two programs. Choose lists of sufficient size to get a good idea of the time differences.
4. Rewrite the sort program for arrays. Continue using an order function as argument. Perform the test on arrays filled in the same manner as for the lists.
5. What can we say about the results of these tests?

Summary

This chapter has shown the different ways to compile an Objective Caml program. The bytecode compiler is favorable for portable code, allowing for the system independent distribution of programs and libraries. This property is lost in the case of standalone bytecode executables. The native compiler trades producing efficient architecture dependent code for a loss of portability.

To Learn More

The techniques to compile for abstract machines were used in the first generation of SmallTalk, then in the functional languages LISP and ML. The argument that the use of abstract machines will hinder performance has put a shadow on this technique for a long time. Now, the JAVA language has shown that the opposite is true. An abstract machine provides several advantages. The first is to facilitate the porting of a compiler to different architectures. The part of the compiler related to portability has been well defined (the abstract machine interpreter and part of runtime library). Another benefit of this technique is portable code. It is possible to compile an application on one architecture and execute it on another. Finally, this technique simplifies compiler construction by adding specific instructions for the type of language to compile. In the case of functional languages, the abstract machines make it easy to create the closures (packing environment and code together) by adding the notion of execution environment to the abstract machine.

To compensate for the loss in efficiency caused by the use of the bytecode interpreter, one can expand the set of abstract machine instructions to include those of a real machine at runtime. This type of expansion has been found in the implementation of Lisp (llm3) and JAVA (JIT). The performance increases, but does not reach the level of a native C compiler.

One difficulty of functional language compilation comes from closures. They contain both the executable code and execution environment (see page 23).
The choice of implementation for the environment and the access of values in the environment has a significant influence on the performance of the code produced. An important function of the environment consists of obtaining access to values in constant time; the variables are viewed as indexes in an array containing their values. This requires the preprocessing of functional expressions. An example can be found in L. Cardelli's book - *Functional Abstract Machine*. Zinc uses this technique. Another crucial optimization is to avoid the construction of useless closures. Although all functions in ML can be viewed as functions with only one argument, it is necessary to not create intermediate closures in the case of application on several arguments. For example, when the function `add` is applied with two integers, it is not useful to create the first closure corresponding to the function of applying `add` to the first argument. It is necessary to note that the creation of a closure would allocate certain memory space for the environment and would require the recovery of that memory space in the future (第 9 章参照). Automatic memory recovery is the second major performance concern, along with environment.

Finally, bootstrapping allows us to write the majority of a compiler with the same language which it is going to compile. For this reason, like the chicken and the egg, it is necessary to define the minimal part of the language which can be expanded later. In fact, this property is hardly appreciable for classifying the languages and their implementations. This property is also used as a measure of the capability of a language to be used in the implementation of a compiler. A compiler is a large program, and bootstrapping is a good test of it's correctness and performance. The following are links to the references:

リンク: http://caml.inria.fr/camlstone.txt

At that time, Caml was compiled over fifty machines, these were antecedent versions of Objective Caml. We can get an idea of how the present Objective Caml has been improved since then.

8

Libraries

Every language comes with collections of programs that are reusable by the programmer, called **libraries**. The quality and diversity of these programs are often some of the criteria one uses to assess the ease of use of a language. You could separate libraries into two categories: those that offer types and functions that are often useful but could be written in the language, and those that offer functionality that cannot be defined in the language. The first group saves the programmer the effort of redefining utilities such as stacks, lists, etc. The second group extends the possible uses of the language by incorporating new functionality into it.

The Objective Caml language distribution comes with many precompiled libraries. For the curious reader, the uncompiled version of these libraries comes packaged with the source code distribution for the language.

In Objective Caml, all the libraries are organized into **modules** that are also compilation units. Each one contains declarations of globals and types, exceptions and values that can be used in programs. In this chapter we are not interested in how to create new modules; we just want to use the existing ones. Chapter 14 will revisit the concepts of the module and the compilation unit while describing the module language of Objective Caml, including parameterized modules. Regarding the creation of libraries that incorporate code that is not written in Objective Caml, chapter 12 will describe how to integrate Objective Caml programs with code written in C.

The Objective Caml distribution contains a preloaded library (the `Pervasives` module), a collection of basic modules called the **standard library**, and many other libraries adding functionality to the language. Some of the libraries are briefly shown in this chapter while others are described in later chapters.

Chapter Outline

This chapter describes the collection of libraries in the Objective Caml distribution. Some have been used in previous chapters, such as the `Graphics` library (see chapter 5), or the `Array` library. The first section shows the organization of the various libraries. The second section finishes describing the preloaded `Pervasives` module. The third section classifies the set of modules found in the standard library. The fourth section examines the high precision math libraries and the libraries for dynamically loading code.

Categorization and Use of the Libraries

The libraries in the Objective Caml distribution fall into three categories. The first contains preloaded global declarations. The second is called the standard library and is subdivided into four parts:

- data structures;
- input/output
- system interface;
- lexical and syntactic analysis.

Finally there are the libraries in the third group that generally extend the language, such as the `Graphics` library (see chapter 5). In this last group you will find libraries dealing with the following areas: regular expressions (`Str`), arbitrary-precision math (`Num`), Unix system calls (`Unix`), lightweight processes (`Threads`) and dynamic loading of bytecode (`Dynlink`).

The I/O and the system interface portions of the standard library are compatible with different operating systems such as Unix, Windows and MacOS. This is not always the case with the libraries in the third group (those that extend the language). There are also many independently written libraries that are not part of the Objective Caml distribution.

Usage and naming To use modules or libraries in a program, one has to use dot notation to specify the module name and the object to access. For example if one wants to use a function `f` in a library called `Name`, one qualifies it as `Name.f`. To avoid having to prefix everything with the name of the library, it is possible to open the library and use `f` directly.

構文 : `open Name`

From then on, all the global declarations of the library `Name` will be considered as if they belonged to the global environment. If two declarations have the same name in two distinct open libraries, then only the last declaration is visible. To be able to call the first, it would be necessary to use the point notation.

Preloaded Library

The `Pervasives` library is always preloaded so that it will be available at the toplevel (interactive) loop or for inline compilation. It is always linked and is the initial environment of the language. It contains the declarations of:

- **type**: basic types (*int*, *char*, *string*, *float*, *bool*, *unit*, *exn*, *'a array*, *'a list*) and the types *'a option* (see page 223) and *('a, 'b, 'c) format* (see page 267).
- **exceptions**: A number of exceptions are raisable by the execution library. Some of the more common ones are the following:
  - *Failure* **of** *string* that is raised by the function `failwith` applied to a string.
  - *Invalid_argument* **of** *string* that indicates that an argument cannot be handled by the function having raised the exception. The function `invalid_arg` applied to a string starts this exception.
  - *Sys_error* **of** *string*, for the input/output, typically in attempting to open a nonexistent file for reading.
  - *End_of_file* for detecting the end of a file.
  - *Division_by_zero* for zero divide errors between integers.

  As well as internal exceptions like:
  - *Out_of_memory* and *Stack_overflow* for going beyond the memory of the heap or the stack. It should be noted that a program cannot recover from the `Out_of_memory` exception. In effect, when it is raised it is too late to allocate new memory space to continue functioning.
    Handling the `Stack_Overflow` exception differs depending on whether the program was compiled in byte code or native code. In the latter case, it is not possible to recover.
- **functions**: there are roughly 140, half of which correspond to the C functions of the execution library. There you may find mathematical and comparison operators, functions on integer and floating-point numbers, functions on character strings, on references and input-output. It should be noted that a certain number of these declarations are in fact synonyms for declarations defined in other modules. They are nevertheless declared here for historical and implementation reasons.

Standard Library

The standard library contains a group of stable modules. These are operating system independent. There are currently 29 modules in the standard library containing 400 functions, 30 types of which half are abstract, 8 exceptions, 10 sub-modules, and 3 parameterized modules. Clearly we will not describe all of the declarations in all of these modules. Indeed, the reference manual [LRVD99] already does that quite well. Only those modules presenting a new concept or a real difficulty in use will be detailed.

The standard library can be divided into four distinct parts:

- **linear data structures** (15 modules), some of which have already appeared in the first part;
- **input-output** (4 modules), for the formatting of output, the persistence and creation of cryptographic keys;
- **parsing and lexical analysis** (4 modules). They are described in chapter 11 (page 289);
- **system interface** that permit communication and examination of parameters passed to a command, directory navigation and file access.

To these four groups we add a fifth containing some utilities for handling or creating structures such as functions for text processing or generating pseudo-random numbers, etc.

Utilities

The modules that we have named "utilities" concern:

- characters: the `Char` module primarily contains conversion functions;
- object cloning: `OO` will be presented in chapter 15 (page 439), on object oriented programming
- lazy evaluation: `Lazy` is first presented on page 105;
- random number generator: `Random` will be described below.

Generation of Random Numbers

The `Random` module is a pseudo-random number generator. It establishes a random number generation function starting with a number or a list of numbers called a **seed**. In order to ensure that the function does not always return the same list of numbers, the programmer must give it a different seed each time the generator is initialized.

From this seed the function generates a succession of seemingly random numbers. Nevertheless, an initialization with the same seed will create the same list. To correctly initialize the generator, you need to find some outside resource, like the date represented in milliseconds, or the length of time since the start of the program.

The functions of the module:

- initialization: `init` of type *`int -> unit`* and `full_init` of type *`int array -> unit`* initialize the generator. The second function takes an array of seeds.
- generate random numbers: `bits` of type *`unit -> int`* returns a positive integer, `int` of type *`int -> int`* returns a positive integer ranging from 0 to a limit given as a parameter, and `float` returns a float between 0. and a limit given as a parameter.

Linear Data Structures

The modules for linear data structures are:

- simple modules: `Array`, `String`, `List`, `Sort`, `Stack`, `Queue`, `Buffer`, `Hashtbl` (that is also parameterized) and `Weak`;
- parameterized modules: `Hashtbl` (of `HashedType` parameters), `Map` and `Set` (of `OrderedType` parameters).

The parameterized modules are built from the other modules, thus making them more generic. The construction of parameterized modules will be presented in chapter 14, page 423.

Simple Linear Data Structures

The name of the module describes the type of data structures manipulated by the module. If the type is abstract, that is to say, if the representation is hidden, the current convention is to name it *t* inside the module. These modules establish the following structures:

- module `Array`: vectors;
- module `List`: lists;
- module `String`: character strings;
- module `Hashtbl`: hash tables (abstract type);
- module `Buffer`: extensible character strings (abstract type);
- module `Stack`: stacks (abstract type);
- module `Queue`: queues or **FIFO** (abstract type);
- module `Weak`: vector of weak pointers (abstract type).

Let us mention one last module that implements linear data structures:

- module `Sort`: sorting on lists and vectors, merging of lists.

Family of common functions Each of these modules (with the exception of `Sort`), has functions for defining structures, creating/accessing elements (such as handler functions), and converting to other types. Only the `List` module is not physically modifiable. We will not give a complete description of all these functions. Instead, we will focus on families of functions that one finds in these modules. Then we will detail the `List` and `Array` modules that are the most commonly used structures in functional and imperative programming.

One finds more or less the following functionality in all these modules:

- a `length` function that takes the value of a type and calculates an integer corresponding to its length;
- a `clear` function that empties the linear structure, if it is modifiable;

- a function to add an element, `add` in general, but sometimes named differently according to common practice, (for example, `push` for stacks);
- a function to access the n-th element, often called `get`;
- a function to remove an element (often the first) `remove` or `take`.

In the same way, in several modules the names of functions for traversal and processing are the same:

- `map`: applies a function on all the elements of the structure and returns a new structure containing the results of these calls;
- `iter`: like `map`, but drops successive results, and returns `()`.

For the structures with indexed elements we have:

- `fill`: replaces (modifies in place) a part of the structure with a value;
- `blit`: copies a part of one structure into another structure of the same type;
- `sub`: copies a part of one structure into a newly created structure.

Modules `List` *and* `Array`

We describe the functions of the two libraries while placing an emphasis on the similarities and the particularities of each one. For the functions common to both modules, *t* designates either the *'a list* or *'a array* type. When a function belongs to one module, we will use the dot notation.

Common or analogous functionality The first of them is the calculation of length.

```
List.length  :  'a t -> int
```

Two functions permitting the concatenation of two structures or all the structures of a list.

```
List.append  :  'a t -> 'a t -> 'a t
List.concat  :  'a t list -> 'a t
```

Both modules have a function to access an element designated by its position in the structure.

```
List.nth   :  'a list -> int -> 'a
Array.get  :  'a array -> int -> 'a
```

The function to access an element at index `i` of a vector `t`, which is frequently used, has a syntactic shorthand: `t.(i)`.

Two functions allow you to apply an operation to all the elements of a structure.

iter	:	('a -> unit) -> 'a t -> unit
map	:	('a -> 'b) -> 'a t -> 'b t

You can use `iter` to print the contents of a list or a vector.

```
# let print_content iter print_item xs =
    iter (fun x → print_string"("; print_item x; print_string")") xs;
    print_newline() ;;
val print_content : (('a -> unit) -> 'b -> 'c) -> ('a -> 'd) -> 'b -> unit =
  <fun>
# print_content List.iter print_int [1;2;3;4;5] ;;
(1)(2)(3)(4)(5)
- : unit = ()
# print_content Array.iter print_int [|1;2;3;4;5|] ;;
(1)(2)(3)(4)(5)
- : unit = ()
```

The `map` function builds a new structure containing the result of the application. For example, with vectors whose contents are modifiable:

```
# let a = [|1;2;3;4|] ;;
val a : int array = [|1; 2; 3; 4|]
# let b = Array.map succ a ;;
val b : int array = [|2; 3; 4; 5|]
# a, b;;
- : int array * int array = ([|1; 2; 3; 4|], [|2; 3; 4; 5|])
```

Two iterators can be used to compose successive applications of a function on all elements of a structure.

fold_left	:	('a -> 'b -> 'a) -> 'a -> 'b t -> 'a
fold_right	:	('a -> 'b -> 'b) -> 'a t -> 'b -> 'b

You have to give these iterators a base case that supplies a default value when the structure is empty.

```
fold_left f r [v1; v2; ...; vn]   =  f ... ( f  (f r v1) v2 ) ... vn
fold_right f [v1; v2; ...; vn] r  =  f v1 ( f v2   ... (f vn r) ... )
```

These functions allow you to easily transform binary operations into n-ary operations. When the operation is commutative and associative, left and right iteration are indistinguishable:

```
# List.fold_left (+) 0 [1;2;3;4] ;;
- : int = 10
# List.fold_right (+) [1;2;3;4] 0 ;;
```

```
- : int = 10
# List.fold_left List.append [0] [[1];[2];[3];[4]] ;;
- : int list = [0; 1; 2; 3; 4]
# List.fold_right List.append [[1];[2];[3];[4]] [0] ;;
- : int list = [1; 2; 3; 4; 0]
```

Notice that, for binary concatenation, an empty list is a neutral element to the left and to the right. We find thus, in this specific case, the equivalence of the two expressions:

```
# List.fold_left List.append [] [[1];[2];[3];[4]] ;;
- : int list = [1; 2; 3; 4]
# List.fold_right List.append [[1];[2];[3];[4]] [] ;;
- : int list = [1; 2; 3; 4]
```

We have, in fact, found the `List.concat` function.

Operations specific to lists. It is useful to have the following list functions that are provided by the `List` module:

`List.hd`	:	*'a list -> 'a* first element of the list
`List.tl`	:	*'a list -> 'a* the list, without its first element
`List.rev`	:	*'a list -> 'a list* reversal of a list
`List.mem`	:	*'a -> 'a list -> bool* membership test
`List.flatten`	:	*'a list list -> 'a list* flattens a list of lists
`List.rev_append`	:	*'a list -> 'a list -> 'a list* is the same as `append (rev l1) l2`

The first two functions are partial. They are not defined on the empty list and raise a `Failure` exception. There is a variant of `mem`: `memq` that uses physical equality.

```
# let c = (1,2) ;;
val c : int * int = (1, 2)
# let l = [c] ;;
val l : (int * int) list = [(1, 2)]
# List.memq (1,2) l ;;
- : bool = false
# List.memq c l ;;
- : bool = true
```

The `List` module provides two iterators that generalize boolean conjunction and disjunction (and / or): `List.for_all` and `List.exists` that are defined by iteration:

```
# let for_all f xs = List.fold_right (fun x → fun b → (f x) & b) xs true ;;
val for_all : ('a -> bool) -> 'a list -> bool = <fun>
# let exists f xs = List.fold_right (fun x → fun b → (f x) or b) xs false ;;
val exists : ('a -> bool) -> 'a list -> bool = <fun>
```

There are variants of the iterators in the `List` module that take two lists as arguments and traverse them in parallel (`iter2`, `map2`, etc.). If they are not the same size, the `Invalid_argument` exception is raised.

The elements of a list can be searched using the criteria provided by the following boolean functions:

`List.find`	:	*('a -> bool) -> 'a list -> 'a*
`List.find_all`	:	*('a -> bool) -> 'a list -> 'a list*

The `find_all` function has an alias: `filter`.

A variant of the general search function is the partitioning of a list:

`List.partition`	:	*('a -> bool) -> 'a list -> 'a list * 'a list*

The `List` module has two often necessary utility functions permitting the division and creation of lists of pairs:

`List.split`	:	*('a * 'b) list -> 'a list * 'b list*
`List.combine`	:	*'a list -> 'b list -> ('a * 'b) list*

Finally, a structure combining lists and pairs is often used: **association lists**. They are useful to store values associated to keys. These are lists of pairs such that the first entry is a key and the second is the information associated to the key. One has these data structures to deal with pairs:

`List.assoc`	:	*'a -> ('a * 'b) list -> 'b* extract the information associated to a key
`List.mem_assoc`	:	*'a -> ('a * 'b) list -> bool* test the existence of a key
`List.remove_assoc`	:	*'a -> ('a * 'b) list -> ('a * 'b) list* deletion of an element corresponding to a key

Each of these functions has a variant using physical equality instead of structural equality: `List.assq`, `List.mem_assq` and `List.remove_assq`.

Handlers specific to Vectors. The vectors that imperative programmers often use are physically modifiable structures. The `Array` module furnishes a function to change the value of an element:

Array.set	:	*'a array -> int -> 'a -> unit*

Like `get`, the `set` function has a syntactic shortcut: `t.(i) <- a`.

There are three vector allocation functions:

Array.create	:	*int -> 'a -> 'a array* creates a vector of a given size whose elements are all initialized with the same value
Array.make	:	*int -> 'a -> 'a array* alias for `create`
Array.init	:	*int -> (int -> 'a) -> 'a array* creates a vector of a given size whose elements are each initialized with the result of the application of a function to the element's index

Since they are frequently used, the `Array` module has two functions for the creation of matrices (vectors of vectors):

Array.create_matrix	:	*int -> int -> 'a -> 'a array array*
Array.make_matrix	:	*int -> int -> 'a -> 'a array array*

The `set` function is generalized as a function modifying the values on an interval described by a starting index and a length:

Array.fill	:	*'a array -> int -> int -> 'a -> unit*

One can copy a whole vector or extract a sub-vector (described by a starting index and a length) to obtain a new structure:

Array.copy	:	*'a array -> 'a array*
Array.sub	:	*'a array -> int -> int -> 'a array*

The copy or extraction can also be done towards another vector:

Array.blit	:	*'a array -> int -> 'a array -> int -> int -> unit*

The first argument is the index into the first vector, the second is the index into the second vector and the third is the number of values copied. The three functions `blit`, `sub` and `fill` raise the `Invalid_argument` exception.

The privileged use of indices in the vector manipulation functions leads to the definition of two specific iterators:

Array.iteri	:	(int -> 'a -> unit) -> 'a array -> unit
Array.mapi	:	(int -> 'a -> 'b) -> 'a array -> 'b array

They apply a function whose first argument is the index of the affected element.

```
#   let f i a = (string_of_int i) ^ ":" ^ (string_of_int a) in
    Array.mapi f  [| 4; 3; 2; 1; 0 |] ;;
- : string array = [|"0:4"; "1:3"; "2:2"; "3:1"; "4:0"|]
```

Although the Array module does not have a function to modify the contents of all the elements in a vector, this effect can be easily obtained using iteri:

```
# let iter_and_set f t =
   Array.iteri (fun i → fun x → t.(i) <- f x) t ;;
val iter_and_set : ('a -> 'a) -> 'a array -> unit = <fun>
# let v = [|0;1;2;3;4|] ;;
val v : int array = [|0; 1; 2; 3; 4|]
# iter_and_set succ v ;;
- : unit = ()
# v ;;
- : int array = [|1; 2; 3; 4; 5|]
```

Finally, the Array module provides two list conversion functions:

Array.of_list	:	'a list -> 'a array
Array.to_list	:	'a array -> 'a list

Input-output

The standard library has four input-output modules:

- module Printf: for the formatting of output;
- Format: pretty-printing facility to format text within "pretty-printing boxes". The pretty-printer breaks lines at specified break hints, and indents lines according to the box structure.
- module Marshal: implements a mechanism for persistent values;
- module Digest: for creating unique keys.

The description of the Marshal module will be given later in the chapter when we begin to discuss persistent data structures (see page 228).

Module Printf

The Printf module formats text using the rules of the printf function in the C language library. The display format is represented as a character string that will be

decoded according to the conventions of `printf` in C, that is to say, by specializing the `%` character. This character followed by a letter indicates the type of the argument at this position. The following format `"(x=%d, y=%d)"` indicates that it should put two integers in place of the `%d` in the output string.

Specification of formats. A format defines the parameters for a printed string. Those, of basic types: *int*, *float*, *char* and *string*, will be converted to strings and will replace their occurrence in the printed string. The values 77 and 43 provided to the format `"(x=%d, y=%d)"` will generate the complete printed string `"(x=77, y=43)"`. The principal letters indicating the type of conversion to carry out are given in figure 8.1.

Type	Letter	Result
integer	`d` or `i`	signed decimal
	`u`	unsigned decimal
	`x`	unsigned hexadecimal, lower case form
	`X`	same, with upper case letters
character	`c`	character
string	`s`	string
float	`f`	decimal
	`e` or `E`	scientific notation
	`g` or `G`	same
boolean	`b`	`true` or `false`
special	`a` or `t`	functional parameter
		of type *(out_channel -> 'a -> unit) -> 'a -> unit*
		or *out_channel -> unit*

図 8.1: Conversion conventions.

The format also allows one to specify the justification of the conversion, which allows for the alignment of the printed values. One can indicate the size in conversion characters. For this one places between the `%` character and the type of conversion an integer number as in `%10d` that indicates a conversion to be padded on the right to ten characters. If the size of the result of the conversion exceeds this limit, the limit will be discarded. A negative number indicates left justification. For conversions of floating point numbers, it is helpful to be able to specify the printed precision. One places a decimal point followed by a number to indicate the number of characters after the decimal point as in `%.5f` that indicates five characters to the right of the decimal point.

There are two specific format letters: `a` and `t` that indicate a functional argument. Typically, a print function defined by the user. This is specific to Objective Caml.

Functions in the module The types of the five functions in this module are given in figure 8.2.

```
fprintf  :  out_channel -> ('a, out_channel, unit) format -> 'a
printf   :   ('a, out_channel, unit) format -> 'a
eprintf  :  ('a, out_channel, unit) format -> 'a
sprintf  :  ('a, unit, string) format -> 'a
bprintf  :  Buffer.t -> ('a, Buffer.t, string) format -> 'a
```

図 8.2: `Printf` formatting functions.

The `fprintf` function takes a channel, a format and arguments of types described in the format. The `printf` and `eprintf` functions are specializations on standard output and standard error. Finally, `sprintf` and `bprintf` do not print the result of the conversion, but instead return the corresponding string.

Here are some simple examples of the utilization of formats.

```
# Printf.printf "(x=%d, y=%d)" 34 78 ;;
(x=34, y=78)- : unit = ()
# Printf.printf "name = %s, age = %d" "Patricia" 18 ;;
name = Patricia, age = 18- : unit = ()
# let s = Printf.sprintf "%10.5f\n%10.5f\n" (-.12.24) (2.30000008) ;;
val s : string = " -12.24000\n   2.30000\n"
# print_string s ;;
-12.24000
   2.30000
- : unit = ()
```

The following example builds a print function from a matrix of floats using a given format.

```
# let print_mat m =
    Printf.printf "\n" ;
    for i=0 to (Array.length m)-1 do
      for j=0 to (Array.length m.(0))-1 do
        Printf.printf "%10.3f" m.(i).(j)
      done ;
      Printf.printf "\n"
    done ;;
val print_mat : float array array -> unit = <fun>
# print_mat (Array.create 4  [| 1.2; -.44.22; 35.2 |]) ;;

     1.200   -44.220    35.200
     1.200   -44.220    35.200
     1.200   -44.220    35.200
     1.200   -44.220    35.200
- : unit = ()
```

Note on the *format* type. The description of a format adopts the syntax of character strings, but it is not a value of type *string*. The decoding of a format, according to the preceding conventions, builds a value of type *format* where the *'a* parameter is instantiated either with *unit* if the format does not mention a parameter, or by a functional type corresponding to a function able to receive as many arguments as are mentioned and returning a value of type *unit*.

One can illustrate this process by partially applying the `printf` function to a format:

```
# let p3 =
    Printf.printf "begin\n%d is val1\n%s is val2\n%f is val3\n" ;;
begin
val p3 : int -> string -> float -> unit = <fun>
```

One obtains thus a function that takes three arguments. Note that the word `begin` had already been printed. Another format would have given another type of function:

```
# let p2 =
    Printf.printf "begin\n%f is val1\n%s is val2\n";;
begin
val p2 : float -> string -> unit = <fun>
```

In providing arguments one by one to `p3`, one progressively obtains the output.

```
# let p31 = p3 45 ;;
45 is val1
val p31 : string -> float -> unit = <fun>
# let p32 = p31 "hello" ;;
hello is val2
val p32 : float -> unit = <fun>
# let p33 = p32 3.14 ;;
3.140000 is val3
val p33 : unit = ()
# p33 ;;
- : unit = ()
```

From the last obtained value, nothing is printed: it is the value `()` of type *unit*.

One cannot build a format using values of type *string*:

```
# let f d =
   Printf.printf (d^d);;
Characters 27-30:
   Printf.printf (d^d);;
                  ^^^
This expression has type string but is here used with type
  ('a, out_channel, unit) format
```

The compiler cannot know the value of the string passed as an argument. It thus cannot know the type that instantiates the *'a* parameter of type `format`.

On the other hand, strings are physically modifiable values, it would thus be possible to replace, for example, the `%d` part with another letter, thus dynamically changing the print format. This conflicts with the static generation of the conversion function.

`Digest` *Module*

A hash function converts a character string of unspecified size into a character string of fixed length, most often smaller. Hashing functions return a **fingerprint** (*digest*) of their entry.

Such functions are used for the construction of hash tables, as in the `Hashtbl` module, permitting one to rapidly test if an element is a member of such a table by directly accessing the fingerprint. For example the function `f_mod_n`, that generates the modulo n sum of the ASCII codes of the characters in a string, is a hashing function. If one creates an n by n table to arrange the strings, from the fingerprint one obtains direct access. Nevertheless two strings can return the same fingerprint. In the case of collisions, one adds to the hash table an extension to store these elements. If there are too many collisions, then access to the hash table is not very effective. If the fingerprint has a length of n bits, then the probability of collision between two different strings is $1/2^n$.

A **non-reversible** hash function has a very weak probability of collision. It is thus difficult, given a fingerprint, to construct a string with this fingerprint. The preceding function `f_mod_n` is not, based on the evidence, such a function. One way hash functions permit the authentification of a string, that it is for some text sent over the Internet, a file, etc.

The `Digest` module uses the **MD5** algorithm, short for **Message Digest 5**. It returns a 128 bit fingerprint. Although the algorithm is public, it is impossible (today) to carry out a reconstruction from a fingerprint. This module defines the *`Digest.t`* type as an abbreviation of the *`string`* type. The figure 8.3 details the main functions of this module.

`string`	:	*`string -> t`* returns the fingerprint of a string
`file`	:	*`string -> t`* returns the fingerprint of a file

図 8.3: Functions of the `Digest` module.

We use the `string` function in the following example on a small string and on a large one built from the first. The fingerprint is always of fixed length.

```
# let s = "The small cat is not dead...";;
val s : string = "The small cat is not dead..."
# Digest.string s;;
- : Digest.t = "\167h{Ic\223)<j\165\250\002H\202\152\201"

# let r = ref s in
```

```
    for i=1 to 100 do r:= s^ !r done;
    String.escaped (Digest.string !r);;
- : string = "-\\012\\218\\210i\\196 \\137\\000M\\004\\015\\001\\251\\1404"
```

The creation of a fingerprint for a program allows one to guarantee the contents and thus avoids the use of a bad version. For example, when code is dynamically loaded (see page 242), a fingerprint is used to select the binary file to load.

```
# Digest.file "basic.ml" ;;
- : Digest.t = "‘\189A]<\209\131)\193\023\166\161\227\017\024-"
```

Persistence

Persistence is the conservation of a value outside the running execution of a program. This is the case when one writes a value in a file. This value is thus accessible to any program that has access to the file. Writing and reading persistent values requires the definition of a format for representing the coding of data. In effect, one must know how to go from a complex structure stored in memory, such as a binary tree, to a **linear** structure, a list of bytes, stored in a file. This is why the coding of persistent values is called **linearization** [1].

Realization and Difficulties of Linearization

The implementation of a mechanism for the linearization of data structures requires choices and presents difficulties that we describe below.

- **read-write of data structures.** Since memory can always be viewed as a vector of words, one value can always correspond to the memory that it occupies, leaving us to preserve the useful part by then compacting the value.
- **share or copy.** Must the linearization of a data structure conserve sharing? Typically a binary tree having two identical children (in the sense of physical equality) can indicate, for the second child, that it has already saved the first. This characteristic influences the size of the saved value and the time taken to do it. On the other hand, in the presence of physically modifiable values, this could change the behavior of this value after a recovery depending on whether or not sharing was conserved.
- **circular structures.** In the case of a circular value, linearization without sharing is likely to loop. It will be necessary to conserve sharing.
- **functional values.** Functional values, or closures, are composed of an environment part and a code part. The code part corresponds to the entry point (address) of the code to execute. What must thus be done with code? It is possible to uniquely store this address, but thus only the same program will find the correct

1. JAVA uses the term **serialization**

meaning of this address. It is also possible to save the list of machine instructions of this function, but that would require having a mechanism to dynamically load code.

- **guaranteeing the type when reloading.** This is the main difficulty of this mechanism. Static typing guarantees that typed values will not generate type errors at execution time. But this is not true except for values belonging to the program during the course of execution. What type can one give to a value outside the program, that was not seen by the type verifier? Just to verify that the re-read value has the monomorphic type generated by the compiler, the type would have to be transmitted at the moment the value was saved, then the type would have to be checked when the value was loaded. Additionally, a mechanism to manage the versions of types would be needed to be safe in case a type is redeclared in a program.

`Marshal` *Module*

The linearization mechanism in the `Marshal` module allows you to choose to keep or discard the sharing of values. It also allows for the use of closures, but in this case, only the pointer to the code is saved.

This module is mainly comprised of functions for linearization via a channel or a string, and functions for recovery via a channel or a string. The linearization functions are parameterizable. The following type declares two possible options:

```
type external_flag =
  No_sharing
| Closures;;
```

The `No_sharing` constant constructor indicates that the sharing of values is not to be preserved, though the default is to keep sharing. The `Closures` constructor allows the use of closures while conserving its pointer to the code. Its absence will raise an exception if one tries to store a functional value.

警告 The `Closures` constructor is inoperative in interactive mode. It can only be used in command line mode.

The reading and writing functions in this module are gathered in figure 8.4.

`to_channel`	:	*out_channel -> 'a -> extern_flag list -> unit*
`to_string`	:	*'a -> extern_flag list -> string*
`to_buffer`	:	*string -> int -> int -> 'a -> extern_flag list -> unit*
`from_channel`	:	*in_channel -> 'a*
`from_string`	:	*string -> int -> 'a*

図 8.4: Functions of the `Marshal` module.

The to_channel function takes an output channel, a value, and a list of options and writes the value to the channel. The to_string function produces a string corresponding to the linearized value, whereas to_buffer accomplishes the same task by modifying part of a string passed as an argument. The from_channel function reads a linearized value from a channel and returns it. The from_string variant takes as input a string and the position of the first character to read in the string. Several linearized values can be stored in the same file or in the same string. For a file, they can be read sequentially. For a string, one must specify the right offset from the beginning of the string to decode the desired value.

```
# let s = Marshal.to_string [1;2;3;4] [] in String.sub s 0 10;;
- : string = "\132\149\166\190\000\000\000\t\000\000"
```

警告 Using this module one loses the safety of static typing (see *infra*, page 234).

Loading a persistent object creates a value of indeterminate type:

```
# let x = Marshal.from_string (Marshal.to_string [1; 2; 3; 4] [] ) 0;;
val x : '_a = <poly>
```

This indetermination is denoted in Objective Caml by the weakly typed variable '_a. You should specify the expected type:

```
# let l =
    let s = (Marshal.to_string [1; 2; 3; 4] [] ) in
      (Marshal.from_string s 0 : int list) ;;
val l : int list = [1; 2; 3; 4]
```

We return to this topic on page 234.

注意

The output_value function of the preloaded library corresponds to calling to_channel with an empty list of options. The input_value function in the Pervasives module directly calls the from_channel function. These functions were kept for compatibility with old programs.

Example: Backup Screens

We want to save the *bitmap*, represented as a matrix of colors, of the whole screen. The save_screen function recovers the *bitmap*, converts it to a table of colors and saves it in a file whose name is passed as a parameter.

```
# let save_screen name =
   let i = Graphics.get_image 0 0 (Graphics.size_x ())
                                  (Graphics.size_y ()) in
    let j = Graphics.dump_image i in
    let oc = open_out name in
      output_value oc j;
      close_out oc;;
val save_screen : string -> unit = <fun>
```

The load_screen function does the reverse operation. It opens the file whose name is passed as a parameter, restores the value stored inside, converts this color matrix into a *bitmap*, then displays the bitmap.

```
# let load_screen name =
    let ic = open_in name in
    let image = ((input_value ic) : Graphics.color array array) in
      close_in ic;
      Graphics.close_graph();
      Graphics.open_graph (" "^(string_of_int(Array.length image.(0)))
                         ^"x"^(string_of_int(Array.length image)));
     let image2 = Graphics.make_image image in
       Graphics.draw_image image2 0 0; image2 ;;
val load_screen : string -> Graphics.image = <fun>
```

警告 Abstract typed values cannot be made persistent.

It is for this reason that the preceding example does not use the abstract *Graphics.image* type, but instead uses the concrete *color array array* type. The abstraction of types is presented in chapter 14.

Sharing

The loss of sharing in a data structure can make the structure completely lose its intended behavior. Let us revisit the example of the symbol generator from page 101. For whatever reason, we want to save the functional values new_s and reset_s, and thereafter use the current value of their common counter. We thus write the following program:

```
# let reset_s,new_s =
    let c = ref 0 in
    ( function () → c := 0 ) ,
    ( function s →  c:=!c+1; s^(string_of_int !c) ) ;;

# let save =
    Marshal.to_string (new_s,reset_s) [Marshal.Closures;Marshal.No_sharing] ;;

# let (new_s1,reset_s1) =
   (Marshal.from_string save 0 : ((string → string ) * (unit → unit))) ;;

# (* 1 *)
  Printf.printf "new_s : \%s\n" (new_s "X");
  Printf.printf "new_s : \%s\n" (new_s "X");
  (* 2 *)
  Printf.printf "new_s1 : \%s\n" (new_s1 "X");
  (* 3 *)
  reset_s1();
```

```
   Printf.printf "new_s1 (after reset_s1) : \%s\n" (new_s1 "X") ;;
Characters 44-46:
Warning: Illegal backslash escape in string
  Printf.printf "new_s : \%s\n" (new_s "X");
                         ^^
  Characters 88-90:
Warning: Illegal backslash escape in string
  Printf.printf "new_s : \%s\n" (new_s "X");
                         ^^
    Characters 140-142:
Warning: Illegal backslash escape in string
  Printf.printf "new_s1 : \%s\n" (new_s1 "X");
                          ^^
      Characters 224-226:
Warning: Illegal backslash escape in string
  Printf.printf "new_s1 (after reset_s1) : \%s\n" (new_s1 "X") ;;
                                           ^^
Characters 148-154:
  Printf.printf "new_s1 : \%s\n" (new_s1 "X");
                                  ^^^^^^
Unbound value new_s1
```

The first two outputs in `(* 1 *)` comply with our intent. The output obtained in `(* 2 *)` after re-reading the closures also appears correct (after `X2` comes `X3`). But, in fact, the sharing of the `c` counter between the re-read functions `new_s1` and `reset_s1` is lost, as the output of `X4` attests that one of them set the counter to zero. Each closure has a copy of the counter and the call to `reset_s1` does not reset the `new_s1` counter to zero. Thus we should not have used the `No_sharing` option during the linearization.

It is generally necessary to conserve sharing. Nevertheless in certain cases where execution speed is important, the absence of sharing speeds up the process of saving. The following example demonstrates a function that copies a matrix. In this case it might be preferable to break the sharing:

```
# let copy_mat_f (m : float array array) =
   let s = Marshal.to_string m [Marshal.No_sharing] in
     (Marshal.from_string s 0 : float array array);;
val copy_mat_f : float array array -> float array array = <fun>
```

One can also use it to create a matrix without sharing:

```
# let create_mat_f n m v =
    let m = Array.create n (Array.create m v) in
      copy_mat_f m;;
val create_mat_f : int -> int -> float -> float array array = <fun>
# let a = create_mat_f 3 4 3.14;;
val a : float array array =
  [|[|3.14; 3.14; 3.14; 3.14|]; [|3.14; 3.14; 3.14; 3.14|];
```

```
    [|3.14; 3.14; 3.14; 3.14|]|]
# a.(1).(2) <- 6.28;;
- : unit = ()
# a;;
- : float array array =
[|[|3.14; 3.14; 3.14; 3.14|]; [|3.14; 3.14; 6.28; 3.14|];
  [|3.14; 3.14; 3.14; 3.14|]|]
```

Which is a more common behavior than that of `Array.create`, and resembles that of `Array.create_matrix`.

Size of Values

It may be useful to know the size of a persistent value. If sharing is conserved, this size also reflects the amount of memory occupied by a value. Although the encoding sometimes optimizes the size of atomic values[2], knowing the size of their respective encodings permits us to compare different implementations of a data structure. In addition, for programs that will never stop themselves, like embedded systems or even network servers; watching the size of data structures can help detect memory leaks. The `Marshal` module has two functions to calculate the size of a constant. They are described in figure 8.5. The total size of a persistent value is the same as the size of its

`header_size`	:	*int*
`data_size`	:	*string -> int -> int*
`total_size`	:	*string -> int -> int*

図 8.5: Size functions of `Marshal`.

data structures plus the size of its header.

Below is a small example of the use of MD5 encoding to compare two representations of binary trees:

```
# let size x = Marshal.data_size (Marshal.to_string x []) 0;;
val size : 'a -> int = <fun>
# type 'a bintree1 = Empty1 | Node1 of 'a * 'a bintree1 * 'a bintree1 ;;
type 'a bintree1 = Empty1 | Node1 of 'a * 'a bintree1 * 'a bintree1
# let s1 =
    Node1(2, Node1(1, Node1(0, Empty1, Empty1), Empty1),
             Node1(3, Empty1, Empty1)) ;;
val s1 : int bintree1 =
  Node1 (2, Node1 (1, Node1 (0, Empty1, Empty1), Empty1),
   Node1 (3, Empty1, Empty1))
# type 'a bintree2 =
```

2. Arrays of characters, for example.

```
    Empty2 | Leaf2 of 'a | Node2 of 'a * 'a bintree2 * 'a bintree2 ;;
type 'a bintree2 =
    Empty2
  | Leaf2 of 'a
  | Node2 of 'a * 'a bintree2 * 'a bintree2
# let s2 =
    Node2(2, Node2(1, Leaf2 0, Empty2), Leaf2 3) ;;
val s2 : int bintree2 = Node2 (2, Node2 (1, Leaf2 0, Empty2), Leaf2 3)
# let s1, s2 = size s1, size s2 ;;
val s1 : int = 13
val s2 : int = 9
```

The values given by the `size` function reflect well the intuition that one might have of the size of `s1` and `s2`.

Typing Problem

The real problem with persistent values is that it is possible to break the type system of Objective Caml. The creation functions return a monomorphic type (*unit* or *string*). On the other hand unmarshalling functions return a polymorphic type *'a*. From the point of view of types, you can do anything with a persistent value. Here is the usage that can be done with it (see chapter 2, page 57): create a function `magic_copy` of type *'a -> 'b*.

```
# let magic_copy a =
    let s = Marshal.to_string a [Marshal.Closures] in
      Marshal.from_string s 0;;
val magic_copy : 'a -> 'b = <fun>
```

The use of such a function causes a brutal halt in the execution of the program.

```
#  (magic_copy 3 : float) +. 3.1;;
Segmentation fault
```

In interactive mode (under Linux), we even leave the toplevel (interactive) loop with a system error signal corresponding to a memory violation.

Interface with the System

The standard library has six system interface modules:

- module `Sys`: for communication between the operating system and the program;
- module `Arg`: to analyze parameters passed to the program from the command line;
- module `Filename`: operations on file names
- module `Printexc`: for the interception and printing of exceptions;

- module `Gc`: to control the mechanism that automatically deallocates memory, described in chapter 9;
- module `Callback`: to call Objective Caml functions from C, described in chapter 12.

The first four modules are described below.

Module `Sys`

This module provides quite useful functions for communication with the operating system, such as handling the signals received by a program. The values in figure 8.6 contain information about the system.

`OS_type`	:	*string* type of system
`interactive`	:	*bool ref* true if executing at the *toplevel*
`word_size`	:	*string* size of a word (32 or 64 bits)
`max_string_length`	:	*int* maximum size of a string
`max_array_length`	:	*int* maximum size of a vector
`time`	:	*unit -> float* gives the time in seconds since the start of the program

図 8.6: Information about the system.

Communication between the program and the system can go through the command line, the value of an environmental variable, or through running another program. These functions are described in figure 8.7.

`argv`	:	*string array* contains the vector of parameters
`getenv`	:	*string -> string* retrieves the value of a variable
`command`	:	*string -> int* executes the command passed as an argument

図 8.7: Communication with the system.

The functions of the figure 8.8 allow us to navigate in the file hierarchy.

`file_exists`	:	*string -> bool* returns `true` if the file exists
`remove`	:	*string -> unit* destroys a file
`rename`	:	*string -> string -> unit* renames a file
`chdir`	:	*string -> unit* change the current directory
`getcwd`	:	*unit -> string* returns the name of the current directory

図 8.8: File manipulation.

Finally, the management of signals will be described in the chapter on system programming (第 18 章参照).

Here is a small program that revisits the example of saving a graphics window as an array of colors. The `main` function verifies that it is not started from the interactive loop, then reads from the command line the names of files to display, then tests if they exist, then displays them (with the `load_screen` function). We wait for a key to be pressed between displaying two images.

```
# let main () =
    if not (!Sys.interactive) then
      for i = 0 to Array.length(Sys.argv) -1 do
        let name = Sys.argv.(i) in
          if Sys.file_exists name then
          begin
            ignore(load_screen name);
            ignore(Graphics.read_key)
          end
      done;;
Characters 235-252:
Warning: this function application is partial,
maybe some arguments are missing.
            ignore(Graphics.read_key)
                   ^^^^^^^^^^^^^^^^^
val main : unit -> unit = <fun>
```

Module `Arg`

The `Arg` module defines a small syntax for command line arguments. With this module, you can parse arguments and associate actions with them. The various elements of the command line are separated by one or more spaces. They are the values stored in the

`Sys.argv` array. In the syntax provided by `Arg`, certain elements are distinguished by starting with the minus character (`-`). These are called command line **keywords** or **switches**. One can associate a specific action with a keyword or take as an argument a value of type *string*, *int* or *float*. The value of these arguments is initialized with the value found on the command line just after the keyword. In this case one can call a function that converts character strings into the expected type. The other elements on the command line are called **anonymous arguments**. One associates an action with them that takes their value as an argument. An undefined option causes the display of some short documentation on the command line. The documentation's contents are defined by the user.

The actions associated with keywords are encapsulated in the type:

```
type spec =
  | Unit of (unit → unit)     (* Call the function with unit argument*)
  | Set of bool ref           (* Set the reference to true*)
  | Clear of bool ref         (* Set the reference to false*)
  | String of (string → unit) (* Call the function with a string
                               argument *)
  | Int of (int → unit)       (* Call the function with an int
                               argument *)
  | Float of (float → unit)   (* Call the function with a float
                               argument *)
  | Rest of (string → unit)   (* Stop interpreting keywords and call the
                               function with each remaining argument*)
```

The command line parsing function is:

```
# Arg.parse ;;
- : (string * Arg.spec * string) list -> (string -> unit) -> string -> unit =
<fun>
```

Its first argument is a list of triples of the form `(key, spec, doc)` such that:

- `key` is a character string corresponding to the keyword. It starts with the reserved character `'-'`.
- `spec` is a value of type *spec* specifying the action associated with `key`.
- `doc` is a character string describing the option `key`. It is displayed upon a syntax error.

The second argument is the function to process the anonymous command line arguments. The last argument is a character string displayed at the beginning of the command line documentation.

The `Arg` module also includes:

- `Bad`: an exception taking as its argument a character string. It can be used by the processing functions.

- `usage`: of type *(string * Arg.spec * string) list -> string -> unit*, this function displays the command line documentation. One preferably provides it with the same arguments as those of `parse`.
- `current`: of type *int ref* that contains a reference to the current value of the index in the `Sys.argv` array. One can therefore modify this value if necessary.

By way of an example, we show a function `read_args` that initializes the configuration of the Minesweeper game seen in chapter 6, page 176. The possible options will be `-col`, `-lin` and `-min`. They will be followed by an integer indicating, respectively: the number of columns, the number of lines and the number of mines desired. These values must not be less than the default values, respectively `10`, `10` and `15`.

The processing functions are:

```
# let set_nbcols cf n = cf := {!cf with nbcols = n} ;;
# let set_nbrows cf n = cf := {!cf with nbrows = n} ;;
# let set_nbmines cf n = cf := {!cf with nbmines = n} ;;
```

All three are of type *config ref -> int -> unit*. The command line parsing function can be written:

```
# let read_args() =
   let cf = ref default_config in
   let speclist =
    [("-col", Arg.Int (set_nbcols cf), "number of columns (>=10)");
     ("-lin", Arg.Int (set_nbrows cf), "number of lines (>=10)");
     ("-min", Arg.Int (set_nbmines cf), "number of mines (>=15)")]
   in
   let usage_msg = "usage : minesweep [-col n] [-lin n] [-min n]" in
    Arg.parse speclist (fun s → ()) usage_msg; !cf ;;
val read_args : unit -> config = <fun>
```

This function calculates a configuration that will be passed as arguments to `open_wcf`, the function that opens the main window when the game is started. Each option is, as its name indicates, optional. If it does not appear on the command line, the corresponding parameter keeps its default value. The order of the options is unimportant.

Module `Filename`

The `Filename` module has operating system independant functions to manipulate the names of files. In practice, the file and directory naming conventions differ greatly between Windows, Unix and MacOS.

Module Printexc

This very short module (three functions described in figure 8.9) provides a general exception handler. This is particularly useful for programs executed in command mode[3] to be sure not to allow an exception to escape that would stop the program.

catch	:	('a -> 'b) -> 'a -> 'b general exception handler
print	:	('a -> 'b) -> 'a -> 'b print and re-raise the exception
to_string	:	exn -> string convert an exception to a string

図 8.9: Handling exceptions.

The catch function applies its first argument to its second. This launches the main function of the program. If an exception arrives at the level of catch, that is to say that if it is not handled inside the program, then catch will print its name and exit the program. The print function has the same behavior as catch but re-raises the exception after printing it. Finally the to_string function converts an exception into a character string. It is used by the two preceding functions. If we look again at the main function for displaying *bitmaps*, we might thus write an encapsulating function go in the following manner:

```
# let go () =
    Printexc.catch main ();;
val go : unit -> unit = <fun>
```

This permits the normal termination of the program by printing the value of the uncaptured exception.

Other Libraries in the Distribution

The other libraries provided with the Objective Caml language distribution relate to the following extensions:

- **graphics**, with the portable Graphics module that was described in chapter 5;
- **exact math**, containing many modules, and allowing the use of exact calculations on integers and rational numbers. Numbers are represented using Objective Caml integers whenever possible;

3. The interactive mode has a general exception handler that prints a message signaling that an exception was not handled.

- **regular expression filtering**, allowing easier string and text manipulations. The Str module will be described in chapter 11;
- **Unix system calls**, with the Unix module allowing one to make unix system calls from Objective Caml. A large part of this library is nevertheless compatible with Windows. This bibliography will be used in chapters 18 and 20;
- **light-weight processes**, comprising many modules that will largely be described and used in chapter 19;
- **access to NDBD databases**, works only in Unix and will not be described;
- **dynamic loading of bytecode**, implemented by the Dynlink module.

We will describe the big integer and dynamic loading libraries by using them.

Exact Math

The big numbers library provides exact math functions using integers and rational numbers. Values of type *int* and *float* have two limitations: calculations on integers are done *modulo* the greatest positive integer, which can cause unperceived overflow errors; the results of floating point calculations are rounded, which by propagation can lead to errors. The library presented here mitigates these defects.

This library is written partly in C. For this reason, you have to build an interactive loop that includes this code using the command:

```
ocamlmktop -custom -o top nums.cma -cclib -lnums
```

The library contains many modules. The two most important ones are Num for all the operations and Arith_status for controlling calculation options. The general type *num* is a variant type gathering three basic types:

```
type num = Int of int
         | Big_int of big_int
         | Ratio of ratio
```

The types *big_int* and *ratio* are abstract.

The operations on values of type *num* are followed by the symbol /. For example the addition of two *num* variables is written +/ and will be of type *num -> num -> num*. It will be the same for comparisons. Here is the first example that calculates the factorial:

```
# let rec fact_num n =
    if  Num.(<=/) n  (Num.Int 0) then (Num.Int 1)
    else  Num.( */ ) n  (fact_num ( Num.(-/) n (Num.Int 1)));;
val fact_num : Num.num -> Num.num = <fun>
# let r = fact_num (Num.Int 100);;
val r : Num.num = Num.Big_int <abstr>
# let n = Num.string_of_num r in (String.sub n 0 50) ^ "..." ;;
- : string = "93326215443944152681699238856266700490715968264381..."
```

Opening the `Num` module makes the code of `fact_num` easier to read:

```
# open Num ;;
# let rec fact_num n =
    if  n <=/  (Int 0) then (Int 1)
    else  n */   (fact_num ( n -/ (Int 1))) ;;
val fact_num : Num.num -> Num.num = <fun>
```

Calculations using rational numbers are also exact. If we want to calculate the number e by following the following definition:

$$e = lim_{m \to \infty} \left(1 + \frac{1}{m}\right)^m$$

We should write a function that calculates this limit up to a certain m.

```
# let  calc_e m =
    let a = Num.(+/) (Num.Int 1) ( Num.(//) (Num.Int 1) m) in
      Num.( **/ ) a  m;;
val calc_e : Num.num -> Num.num = <fun>
# let r = calc_e (Num.Int 100);;
val r : Num.num = Ratio <abstr>
# let n = Num.string_of_num r in (String.sub n 0 50) ^ "..." ;;
- : string = "27048138294215260932671947108075308336779383827810..."
```

The `Arith_status` module allows us to control some calculations such as the normalization of rational numbers, approximation for printing, and processing null denominators. The `arith_status` function prints the state of these indicators.

```
# Arith_status.arith_status();;

Normalization during computation --> OFF
     (returned by get_normalize_ratio ())
     (modifiable with set_normalize_ratio <your choice>)

Normalization when printing --> ON
     (returned by get_normalize_ratio_when_printing ())
     (modifiable with set_normalize_ratio_when_printing <your choice>)

Floating point approximation when printing rational numbers --> OFF
     (returned by get_approx_printing ())
     (modifiable with set_approx_printing <your choice>)

Error when a rational denominator is null --> ON
     (returned by get_error_when_null_denominator ())
     (modifiable with set_error_when_null_denominator <your choice>)
- : unit = ()
```

They can be modified according to the needs of a calculation. For example, if we want to print an approximate value for a rational number, we can obtain, for the preceding calculation:

```
# Arith_status.set_approx_printing true;;
- : unit = ()
# Num.string_of_num (calc_e (Num.Int 100));;
- : string = "0.270481382942e1"
```

Calculations with big numbers take longer than those with integers and the values occupy more memory. Nevertheless, this library tries to use the most economical representations whenever possible. In any event, the ability to avoid the propagation of rounding errors and to do calculations on big numbers justifies the loss of efficiency.

Dynamic Loading of Code

The `Dynlink` module offers the ability to dynamically load programs in the form of bytecode. The dynamic loading of code provides the following advantages:

- reduces the size of a program's code. If certain modules are not used, they are not loaded.
- allows the choice at execution time of which module to load. According to certain conditions at execution time you choose to load one module rather than another.
- allows the modification of the behavior of a module during execution. Here again, under some conditions the program can load a new module and hide the old code.

The interactive loop of Objective Caml already uses such a mechanism. It is convenient to let the programmer have access to it as well.

During the loading of an object file (with the `.cmo` extension), the various expressions are evaluated. The main program, that initiated the dynamic loading of the code does not have access to the names of declarations. Therefore it is up to the dynamically loaded module to update a table of functions used by the main program.

警告 | The dynamic loading of code only works for object files in bytecode.

Description of the Module

For dynamic loading of a bytecode file `f.cmo`, we need to know the access path to the file and the names of the modules that it uses. By default, dynamically loaded bytecode files do not have access to the paths and modules of the libraries in the distribution. Thus we have to add the path and the name of the required modules to the dynamic loading of the module.

Many errors can occur during a request to load a module. Not only must the file exist with the right interface in one of the paths, but the bytecode must also be correct

init	:	*unit -> unit* initialize dynamic loading
add_interfaces	:	*string list -> string list -> unit* add the names of modules and paths for loading
loadfile	:	*string -> unit* load a bytecode file
clear_avalaible_units	:	*unit -> unit* empty the names of loadable modules and paths
add_avalaible_units	:	*(string * Digest.t) list -> unit* add the name of a module and a checksum† for loading without needing the interface file
allow_unsafe_modules	:	*bool -> unit* allow the loading of files containing **external** declarations
loadfile_private	:	*string -> unit* the loaded module is not accessible to modules loaded later

† The checksum of an interface `.cmi` can be obtained from the `extract_crc` command found in the catalog of libraries in the distribution.

図 8.10: Functions of the `Dynlink` module.

and loadable. These errors are gathered in the type *error* used as an argument to the `Error` exception and to the `error` function of type *error -> string* that allows the conversion of an error into a clear description.

Example

To write a small program that allows us to illustrate dynamic loading of bytecode, we provide three modules:

- `F` that contains the definition of a reference to a function `f`;
- `Mod1` and `Mod2` that modify in different ways the function referenced by `F.f`.

The `F` module is defined in the file `f.ml`:

```
let g () =
  print_string "I am the 'f' function by default\n" ; flush stdout  ;;
let f = ref g ;;
```

The `Mod1` module is defined in the file `mod1.ml`:

```
print_string "The 'Mod1' module modifies the value of 'F.f'\n" ; flush stdout ;;
```

```
let g () =
  print_string "I am the 'f' function of module 'Mod1'\n" ;
  flush stdout ;;
F.f := g ;;
```

The Mod2 module is defined in the file mod2.ml:

```
print_string "The 'Mod2' module modifies the value of 'F.f'\n" ; flush stdout ;;
let g () =
  print_string "I am the 'f' function of module 'Mod2'\n" ;
  flush stdout ;;
F.f := g ;;
```

Finally we define in the file main.ml, a main program that calls the original function referenced by F.f, loads the Mod1 module, calls F.f again, then loads the Mod2 module and calls the F.f function one last time:

```
let main () =
  try
    Dynlink.init () ;
    Dynlink.add_interfaces [ "Pervasives"; "F" ; "Mod1" ; "Mod2" ]
                           [ Sys.getcwd() ; "/usr/local/lib/ocaml/" ] ;
    !(F.f) () ;
    Dynlink.loadfile "mod1.cmo" ;  !(F.f) () ;
    Dynlink.loadfile "mod2.cmo" ;  !(F.f) ()
  with
    Dynlink.Error e → print_endline (Dynlink.error_message e) ; exit 1 ;;

main () ;;
```

The main program must, in addition to initializing the dynamic loading, declare by a call to Dynlink.add_interfaces the interface used.

We compile all of these modules:

```
$ ocamlc -c f.ml
$ ocamlc -o main dynlink.cma f.cmo main.ml
$ ocamlc -c f.cmo mod1.ml
$ ocamlc -c f.cmo mod2.ml
```

If we execute program main, we obtain:

```
$ main
I am the 'f' function by default
The 'Mod1' module modifies the value of 'F.f'
I am the 'f' function of module 'Mod1'
The 'Mod2' module modifies the value of 'F.f'
I am the 'f' function of module 'Mod2'
```

Upon the dynamic loading of a module, its code is executed. This is demonstrated in our example, with the outputs beginning with `The 'Mod...`. The possible side effects that it contains are therefore reflected at the level of the program that caused the code to be loaded. This is why the different calls to `F.f` call different functions.

The `Dynlink` library offers the basic mechanism for dynamically loading bytecode. The programmer still has to manage tables such that the loading will really be effective.

Exercises

Resolution of Linear Systems

This exercise revisits the resolution of linear systems presented as an exercise in the chapter on imperative programming (第 3 章参照).

1. By using the `Printf` module, write a function `print_system` that aligns the columns of the system.
2. Test this function on the examples given on page 87.

Search for Prime Numbers

The Sieve of Eratosthenes is an easily programmed algorithm that searches for prime numbers in a range of integers, given that the lower limit is a prime number. The method is:

1. Enumerate, in a list, all the values on the range.
2. Remove from the list all the values that are multiples of the first element.
3. Remove this first element from the list, and keep it as a prime.
4. Restart at step 2 as long as the list is not empty.

Here are the steps to create a program that implements this algorithm:

1. Write a function `range` that builds a range of integers represented in the form of a list.
2. Write a function `eras` that calculates the prime numbers on a range of integers starting with 2, according to the algorithm of the Sieve of Eratosthenes.
   Write a function `era_go` that takes an integer and returns a list of all the prime numbers smaller than this integer.
3. We want to write an executable `primes` that one will launch by typing the command `primes n`, where `n` is an integer. This executable will print the prime numbers smaller than `n`. For this we must use the `Sys` module and check whether a parameter was passed.

Displaying *Bitmaps*

Bitmaps saved as *color array array* are bulky. Since 24 bits of color are rarely used, it is possible to encode a *bitmap* in less space. For this we will analyze the number of colors in a *bitmap*. If the number is small (for example less than 256) we can encode each pixel in 1 byte, representing the number of the color in the table of colors of this *bitmap*.

1. Write a function `analyze_colors` exploring a value of type *color array array* and that returns a list of all the colors found in this image.
2. From this list, construct a palette. We will take a vector of colors. The index in the table will correspond to the order of the color, and the contents are the color itself. Write the function `find_index` that returns the index of a value stored in the array.
3. From this table, write a conversion function, `encode`, that goes from a *color array array* to a *string*. Each pixel is thus represented by a character.
4. Define a type `image_tdc` comprising a table that matches colors to a vector of strings, allowing the encoding of a *bitmap* (or color array) using a smaller method.
5. Write the function `to_image_tdc` to convert a *color array array* to this type.
6. Write the function `save_image_tdc` to save the values to a file.
7. Compare the size of the file obtained with the saved version of an equivalent palette.
8. Write the function `from_image_tdc` to do the reverse conversion.
9. Use it to display an image saved in a file. The file will be in the form of a value of type *bitmap_tdc*.

Summary

This chapter gave an overview of the different Objective Caml libraries presented as a set of simple modules (or compilation units). The modules for output formatting (`Printf`), persistant values (`Marshal`), the system interface (`Sys`) and the handling of exceptions (module `Printexc`) were detailed. The modules concerning parsing, memory management, system and network programming and light-weight processes will be presented in the following chapters.

To Learn More

The overview of the libraries in the distribution of the language showed the richness of the basic environment. For the `Printf` module nothing is worth more than reading a work on the C language, such as [HS94]. In [FW00] a solution is proposed for the typing of intput-output of values (module `Marshal`). The MD5 algorithm of the `Digest` module is described on the web page of its designer:

リンク: http://theory.lcs.mit.edu/˜rivest/homepage.html

In the same way you may find many articles on exact arithmetic used by the `num` library on the web page of Valérie Ménissier-Morain :

リンク: http://www-calfor.lip6.fr/˜vmm/

There are also other libraries than those in the distribution, developed by the community of Objective Caml programmers. Objective Caml. The majority of them are listed on the "Camel's hump" site:

リンク: http://caml.inria.fr/humps/index.html

Some of them will be presented and discussed in the chapter on applications development (第 22 章参照).

To know the exact contents of the various modules, don't hesitate to read the description of the libraries in the reference manual [LRVD99] or consult the online version in HTML format (第 1 章参照). To enter into the details of the implementations of these libraries, nothing is better than reading the source code, available in the distribution of the language (第 1 章参照).
Chapter 14 presents the language of Objective Caml modules. This allows you to build simple modules seen as independent compilation units, which will be similar to the modules presented in this chapter.

9

ガーベージコレクション

マイクロプロセッサ上のプログラムの実行モデルは命令型プログラミング言語の実行モデルと良く対応しています。命令型言語ではプログラムを、マシンのメモリ状態を変更する命令の列として解釈します。メモリには主にプログラムによって設定された値が保存されています。しかし他のリソースと同様にメモリは有限です。プログラムは OS によって提供される以上のメモリを使うことはできません。このため将来の計算で必要としない値が保存されているメモリ領域を再利用する必要が出てきます。このようなメモリ管理はプログラムの実行と効率に大きな影響を持っています。

何らかの目的のためにメモリの一部を予約することを**割り当て** *allocation* と呼びます。プログラムをロードする時に行われる**静的な割り当て**とプログラムの実行中に行われる**動的な割り当て**は違います。静的に割り当てられるメモリはプログラムの実行中に解放されませんが、動的に割り当てられる領域はプログラムの実行中に解放され、再利用されます。

メモリ管理をプログラマに任せることには 2 つの大きな問題があります。

- まだ使っている値が含まれているメモリブロックを間違って解放してしまうとその値をアクセスしようとした時に不具合を生じることがあります。このような解放された領域内の値を指すポインタを**揺れ動くポインタ** *dangling pointer* と呼びます。
- プログラムからメモリブロックのアドレスが失われてしまうとそのメモリブロックをプログラムの実行が終るまで解放できなくなってしまいます。このような状況を**メモリリーク**と呼びます。

プログラマによる明示的なメモリ管理ではこの二つの問題を起こさないように注意を払わなければなりません。この作業はプログラムが複雑なデータ構造を扱う時にはかなり困難です。特にメモリブロックを共有しているデータ構造があるとより難しくなるでしょう。

自動メモリ管理機構は、プログラマからこの困難な作業を解放するために数多くのプログラミング言語に導入されてきました。自動メモリ管理の基本的な考え方は、動的に割り当てられた値の中でプログラムに取って必要な値とはプログラムの実行状態から直接または間接的に参照できる値である、というものです。言い替えるとプログラムから参

照できなくなってしまった値は解放し、再利用しても構わないということです。メモリを実際に解放するタイミングは値に到達できなくなった瞬間でもいいですし、後でメモリが必要になった時点でも構いません。

Objective Caml では自動メモリ管理のために**ガーベージコレクション**(GC) と呼ばれる機構を備えています。メモリはデータ構造が作られる時 (コンストラクタが呼ばれる時) に確保され、暗黙のうちに解放されます。ガーベージコレクションは舞台裏で自動的に機能しているのでほとんどのプログラムでは明示的にガーベージコレクションを扱う必要はありません。しかしメモリを頻繁に利用するプログラムではガーベージコレクションの動作が実行効率に大きな影響を与える場合があります。このような場合にはプログラマが GC のパラメータを制御し、明示的にコレクタを呼び出せるようにした方が有益です。また Objective Caml から他の言語を呼び出す時 (第 12 章参照) にはガーベージコレクタが値の表現に対して課している制限を良く理解しなければなりません。

この章のあらまし

この章では動的なメモリ割り当て戦略とガーベージコレクションのアルゴリズムについて解説します。Objective Caml で採用されているガーベージコレクションはこの章で示されたアルゴリズムを組み合わせたものになっています。最初の節ではメモリの種類と性質について背景知識を説明しています。第二節ではメモリ割り当てについて扱い、明示的な解放と暗黙の解放の比較を行います。第三節では代表的な GC のアルゴリズムを紹介します。第四節では Objective Caml のアルゴリズムについて詳細に説明します。第五節ではヒープ領域を制御する `Gc` モジュールの使い方について紹介します。第六節ではキャッシュを実装するのに使われる**弱いポインタ** *weak pointer* の使い方について紹介します。

プログラムの使用するメモリ

機械語コードのプログラムとはメモリ上の値を操作する命令列です。メモリとは一般的に以下の要素から成り立っています。

- プロセッサレジスタ
- スタック
- データセグメント (静的に割り当てられる領域)
- ヒープ領域 (動的に割り当てられる領域)

スタックと動的に割り当てられる領域のみがプログラムの実行中に大きさを変更できます。メモリの種類によって制御の仕方と機能は変化します。メモリの種類毎に行える制御はプログラム言語と OS に依存して変わります。プログラム命令列は通常静的なメモリに置かれますが、動的リンク (242 ページ参照) では動的なメモリを利用します。

メモリの割り当てと解放

C, Pascal, Lisp, ML, SmallTalk, C++, Java, ADA 等のほとんどのプログラミング言語では動的なメモリの割り当ての機能を持っています。

明示的な割り当て

メモリの割り当てには次の 2 種類の方式があります。

- メモリの内容について考慮せず、一定のサイズの領域を予約する方式。
- メモリの内容を初期化して、一定のサイズの領域を予約する方式。

最初の方式は Pascal の `new` 関数や C 言語の `malloc` 関数によって採用されている方式です。これらの関数は、内容を読みだしたり更新できるメモリ領域を指すポインタ (アドレス) を返します。2 番目の方式は Objective Caml や Lisp 等の言語またはオブジェクト指向言語で値を構築する操作に対応しています。オブジェクト指向言語の `new` 関数は値の初期化に必要なパラメータを引数として受け取り、値が設定されたクラスのインスタンスを返り値として返します。関数型言語では構造的な値 (組、リスト、レコード、配列、関数クロージャ) が定義される地点で値が構築されます。

Objective Caml での値の構築の仕方について具体的に調べてみましょう。図 9.1 にメモリ上での値の表現例を示します。

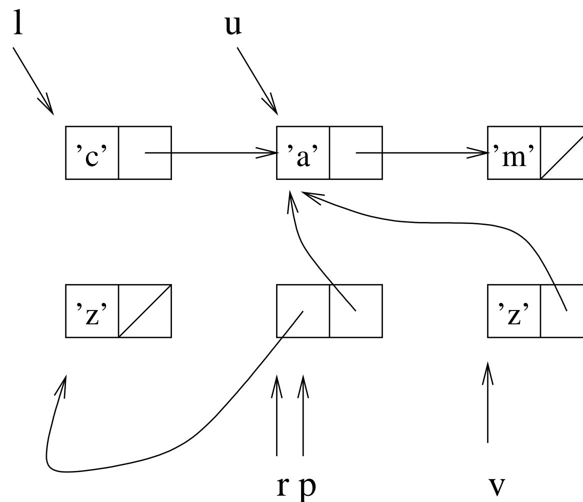


図 9.1: メモリ上の値の表現

```
# let u = let l = ['c'; 'a'; 'm'] in List.tl l ;;
val u : char list = ['a'; 'm']
# let v = let r = ( ['z'] , u )
          in match r with p → (fst p) @ (snd p) ;;
```

```
val v : char list = ['z'; 'a'; 'm']
```

リストの要素は 2 ワードの組によって表現されています。最初の要素は文字であり、2 番目の要素はリストの中の次の要素へのポインタです。実際の実行時の表現は少し違っていますが、その違いは C とのインターフェースについて扱う章 (317 ページ参照) で説明します。

変数 `l` の定義により個々の要素 `['c';'a';'m']` のためのセル (コンストラクタ ::) がメモリ上に確保されます。変数 `u` は変数 `l` の指すセルの末尾のセルに対応しています。すなわち変数 `l` と変数 `u` の指す対象は一部共有されています。これは関数 *`List.tl`* の引数と返り値が持つ性質のためです。

ただし最初の文の評価の後にアクセスできるのは変数 `u` だけです。

二番目の文では要素が 1 つだけのリストが構築され、次にそのリストと変数 `u` が指すリストの組が構築され、変数 `r` に束縛されます。この組はパターンマッチの結果変数 `p` に束縛されます。次に変数 `p` の最初の要素と二番目の要素がリストとして結合され、結果として `['z';'a';'m']` というリストを作り出し、大域変数 `v` に結びつけられます。ここで注意して欲しいのは関数 `snd` の結果 (リスト `['a';'m']`) は引数の変数 `p` と共有されていますが、関数 `fst` の結果 (文字`'z'`) はコピーされているということです。

この例ではメモリ割り当てはすべて明示的でした。言い替えるとプログラマによって構文的に指定されたものでした。

注意

> 図中には明示されていませんが、割り当てられたメモリ領域には後で解放できるようにサイズ情報が書き込まれています。

明示的な解放

明示的にメモリ領域を解放する言語では、メモリ領域のアドレスを受け取ってその領域を解放する演算子 (C 言語の `free` や Pascal の `dispose`) があります。メモリ割り当ての時に書き込まれた情報を利用してプログラムは指定された領域を解放し、後に再利用できるようにします。

動的な割り当ては一般にリストや木などの構造が変化するデータを操作する時に利用されています。そのような複雑なデータ構造を含む領域を一挙に解放することはできません。複雑なデータ構造のすべての要素を辿る関数が必要になります。ここではそのような関数をデストラクタと呼びます。

デストラクタを正しく定義するのはそれほど難しくありませんが、極めて慎重に呼び出す必要があります。データ構造を含むメモリを実際に解放するにはデータ構造を辿りながら言語が持つ解放のための関数を呼び出さなければなりません。メモリを解放する責任をプログラマに任せれば、プログラマはメモリ解放の動作を把握することができますが、メモリの解放の仕方を間違った場合にプログラムの実行に対して深刻な影響が出る危険性を生じます。明示的なメモリ解放の持つ主な危険性とは以下のようなものです。

- 揺れ動くポインタ: あるメモリ領域を指すポインタが存在しているにも関わらず、そのメモリ領域を解放してしまった時にそのポインタのことを揺れ動くポインタ

と呼びます。そのメモリ領域が後に再利用されるとその領域に保存された値の一貫性が失われる危険性があります。

- アクセスできないメモリ領域 (メモリリーク): メモリ領域は割り当てられたままであるが、その領域を指すポインタが失われてしまった状態。このような場合、その領域を解放する手段がなくなってしまいます。明らかにメモリの無駄になります。

明示的なメモリの解放の問題点は、プログラム中で使われる値の生存期間を正確に知ることが難しいという問題に起因しています。

暗黙の解放

暗黙のメモリの解放の機能を持つ言語にはメモリを解放する演算子がありません。つまりプログラマは割り当てられた領域を解放できません。その代わりに言語に自動的にメモリを再利用する機能があり、値が参照されなくなってしまった時や新しいメモリを割り当てることができなくなった時 (ヒープがいっぱいになった時) に自動的に作動します。

自動的にメモリを再利用するアルゴリズムは大域的なデストラクタとも考えることができます。この性質のため特定のデータ構造のためのデストラクタよりも設計と実装が困難です。しかし、一度正確に実装してしまえばメモリ管理の安全性が飛躍的に高まります。とりわけ揺れ動くポインタの危険性は消滅します。

その上、自動メモリ管理機構はヒープに対して良い性質を与えます。

- **稠密性**: 不用なメモリを再利用した後は値は一つの連続したメモリ領域に保存されています。したがってメモリのフラグメンテーションの問題はありません。またヒープに残されている最大の連続領域を割り当てることができます。
- **局所性**: 同じ値を構成する要素はメモリアドレスという意味で近い場所に置かれています。したがって同じメモリページに含まれている可能性が高く、キャッシュの効果を高めます。

ガーベージコレクタを設計する際にはいくつかの基準と制限を考慮に入れなければなりません。

- メモリ利用率: 未使用のメモリ領域を何%にすべきか?
- フラグメンテーション: 連続領域として未使用の領域全体を割り当てることができるか?
- 割り当てと回収の速度
- 値の表現の自由度

実際の言語では安全性の要請が最も重要です。ガーベージコレクタの設計ではこれらの矛盾する要求の中で折衷案を探さなければなりません。

ガーベージコレクション

メモリを自動的に再利用するアルゴリズムには大きく分けて 2 つあります。

- 参照カウント: 個々の割り当て領域は自分を指している参照がいくつあるのか知っています。参照数が 0 になった時、すなわち誰からも参照されなくなった時にその領域を解放します。
- スイープアルゴリズム: **ルート**と呼ばれる参照の集合から辿ることのできるすべての値を、ちょうど有向グラフのすべてのノードを辿るやり方と同じように辿る方式。

スイープアルゴリズムの方がプログラミング言語では良く使われています。参照カウント方式のガーベージコレクタではメモリ領域を回収しない時でも参照カウントを更新するコストがかかります。また相互参照するメモリ領域をうまく回収できないことが知られています。

参照カウント

個々の割り当て領域 (オブジェクト) はカウンタを持っています。このカウンタは自分を指すポインタの数を表しています。オブジェクトを指す参照が増えるごとにカウンタの値に 1 を加えます。オブジェクトを指す参照が消滅するとカウンタの値から 1 を引きます。カウンタの値が 0 になった時にそのオブジェクトは回収されます。

この方式の利点は、オブジェクトが使われなくなった瞬間に解放できることです。カウンタを増減するためのオーバヘッドがかかること以外にもこの方式には欠点があります。相互参照するオブジェクトを扱うのが難しいという点です。以下の例では循環する値 `l` を一時的に作ります。`l` は文字を含むリストですが、最後の要素は最初の文字`'c'` を含むセルを指しています。明らかにこの値は循環しています (図 9.2 参照)。

```
# let rec l = 'c' :: 'a' :: 'm' :: l in List.hd l ;;
- : char = 'c'
```

この文を評価した後、変数 `l` は無効になりますがリストに含まれるすべての要素のカウ

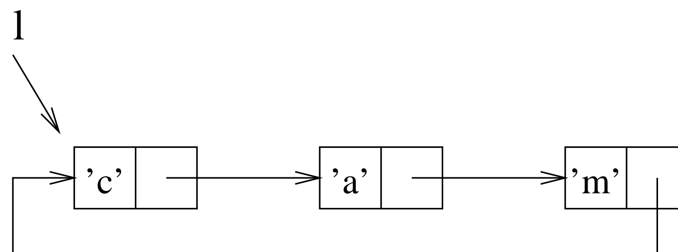


図 9.2: 循環リストのメモリ表現

ンタは 1 に等しくなります (最初の要素は最後の要素から指されているため)。この循環リストへはもはやアクセスできなくなりますが、カウンタが 0 でないので回収もできません。参照カウントを使ったメモリ回収を行い、循環する値を作り出せる言語— 例えば Python — ではスイープアルゴリズムを併用する必要があります。

スイープアルゴリズム

スイープアルゴリズムではヒープ上の値のグラフ構造を探索します。グラフ上の探索はルートと呼ばれる参照の集合から開始されます。ルートとはヒープの外部のデータ構造であり、多くの場合スタックを含んでいます。図 9.1 の例ではトップレベルから参照できる変数 u と変数 v をルートであると仮定します。ルートから参照できるオブジェクトが保存しなければならないオブジェクトです。図 9.3 では太線で表されています。

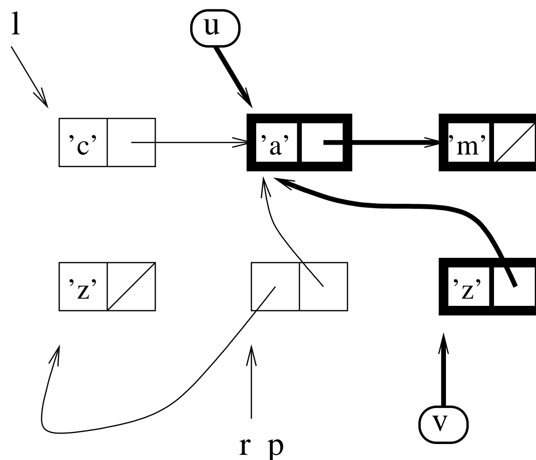


図 9.3: ガーベージコレクション後のメモリの再利用

グラフを辿るためにはヒープ上で整数等の即値とポインタを区別しなければなりません。オブジェクトに含まれている値が整数であれば、その値をアドレスであると思って辿ってはいけません。多くの関数型言語ではこの区別は個々のセルの数ビットを使って表現します。このビットを**タグビット**と呼びます。Objective Caml で整数が 31 ビットなのはこのためです。値の表現法については第 12 章 (327 ページ) でも議論しています。ポインタと即値を区別する別の方法についてこの章の 262 ページで紹介します。

良く使われている代表的なアルゴリズムはマーク・アンド・スイープ とストップ・アンド・コピー です。マーク・アンド・スイープはヒープ領域の未使用のオブジェクトのリスト (フリーリスト) を作ります。ストップ・アンド・コピーは生存しているオブジェクトを新しいメモリ領域にコピーしていきます。

ヒープは値を保存できる箱の列と見なすことができます。図 9.1 の例でのヒープの状態は図 9.4 のように表現できます。

スイープアルゴリズムを比較するのに以下の基準を使います。

- 効率性: 計算量はヒープの大きさに依存するべきか、生存しているオブジェクトの数に依存するべきか?
- 回収率: 未使用のメモリをすべて回収するべきか?
- 稠密性: 未使用のメモリを連続領域にするべきか?

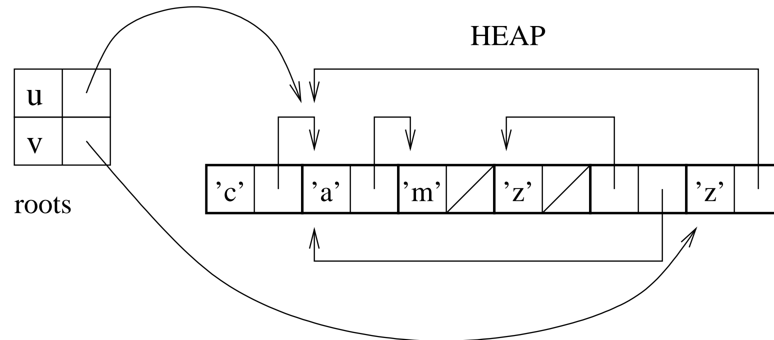


図 9.4: ヒープの状態

- 局所性: 構造を持った値の中のセルはメモリ上で近いアドレスに置くべきか?
- 補助メモリの必要性: 補助的なメモリが必要か?
- 再配置: 値が置かれるアドレスを変えても良いか?

局所性があれば構造を持った値は同じメモリページに置かれます。稠密性があればフラグメンテーションは起こらず、計算に充分な量以上のメモリを使わずに済みます。効率性、回収率、補助メモリの必要性はアルゴリズムの時間空間計算量と密接な関係を持っています。

マーク・アンド・スイープ

マーク・アンド・スイープの基本的なアイデアはヒープ上のすべての未使用のオブジェクトをリストにして (フリーリスト) 管理するというものです。新しいメモリを割り当てる時にはフリーリストから適切なサイズの領域を割り当てます。もしフリーリストが空だったり、あるいは充分なサイズの領域がなかった場合にはマーク・アンド・スイープが行われます。

マーク・アンド・スイープは二段階のアルゴリズムです。

1. マークフェーズ: ルートから到達できるメモリ領域を使用中と見做してマークを付ける。
2. スイープフェーズ: ヒープをアドレス順に調べ、マークの付いていないメモリ領域をフリーリストに加える。

マーク・アンド・スイープによるメモリ管理はヒープ上のセルを 4 色 (白、灰色[1]、黒、斜線) に塗り分けて考えると分かりやすくなります。灰色はマークフェーズでのみ使います。斜線のセルはスイープフェーズで作られます。白のセルは割り当て時に作られます。

The meaning of the gray and black used by marking is as follows:

1. オンラインバージョンでは灰色は青みがかって見えます。

- 灰色: 自分から辿れるセルをまだマークしていない。
- 黒: 自分から辿れるセルはすべてマークしてある。

すべてのセルを辿ったかどうか確かめるためにはすべての灰色のセルを何らかの方法で覚えておかなくてはなりません。マークフェーズが終った段階ですべてのセルは白か黒のどちらかに塗り分けられています。黒いセルはルートから直接的または間接的に参照できるセルです。図 9.5 にマークの途中の状態を示します。ルートの一つである u はすでに辿ってしまっていて、次に v を辿ろうとしている状態です。

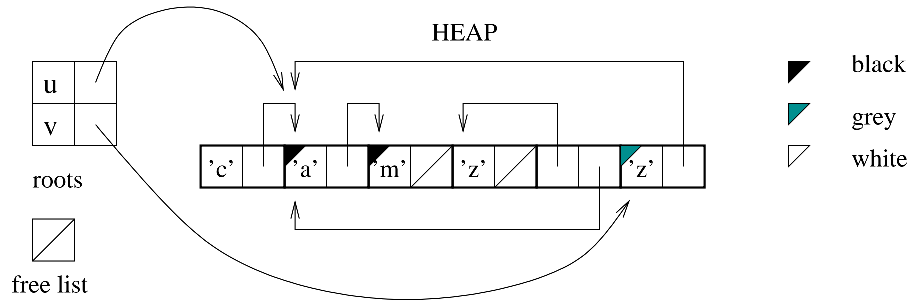


図 9.5: マークフェーズ

フリーリストが形成されるのはスイープフェーズです。スイープフェーズでは次のような規則にしたがって色を書き換えます。

- 生存しているセルである黒を白に書き換えます。
- 白いセルを斜線に書き換え、フリーリストに加えます。

図 9.6 に色を書き換え、フリーリストを作っている途中の状態を示します。

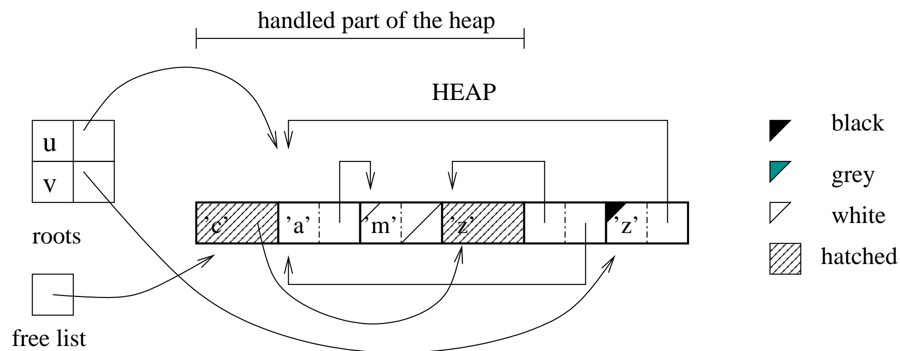


図 9.6: スイープフェーズ

マーク・アンド・スイープには以下のような特徴があります。

- スイープフェーズの計算量はヒープ全体のサイズに依存しています。
- 未使用のメモリはすべて回収されます。
- 未使用のメモリは必ずしも連続領域になりません。
- データの局所性を保証しません。
- 値の置かれるアドレスを変更しません。

マークフェーズは一般に再帰関数を使って実装されます。これは実行スタックのスペースを事実上利用していることになります。実行スタックを無制限に使わないようにマーク・アンド・スイープ のアルゴリズムを記述することは可能ですが、再帰関数を使った場合と比べて効率的に書きにくいことが分かっています。

マーク・アンド・スイープでは値の大きさを知る必要があります。値の大きさは値の中に埋め込むことも可能ですし、ヒープ領域を分割してサイズごとに別のページに保存するようにして、値の大きさをアドレスから計算できるようにすることも可能です。最初期の Lisp に対して設計されたマーク・アンド・スイープのアルゴリズムは今でも広く使われています。Objective Caml のガーベージコレクタの一部でもマーク・アンド・スイープのアルゴリズムを使っています。

ストップ・アンド・コピー

このガーベージコレクタでは値を詰めてコピーするための二次メモリ領域を使用します。ヒープは同じ大きさの二つの領域に分割されます。使用中の領域を起領域 *from-space*、コピーされる領域を至領域 *to-space* と呼びます。

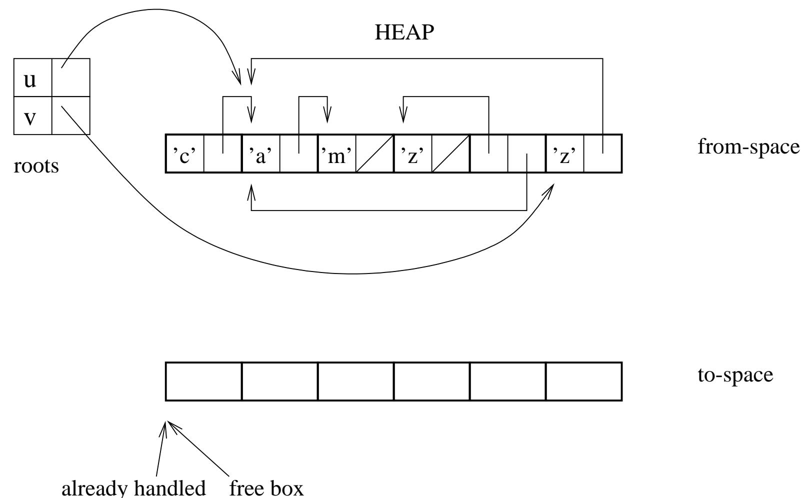


図 9.7: ストップ・アンド・コピーの開始直前の状態

アルゴリズムは以下のように進みます。まず起領域の中で、ルートから辿れる値を至領域にコピーします。値の置かれるアドレスは、その値を指すポインタを正しく書き換えることができるように (多くの場合、古い値の領域に) 保存されます。

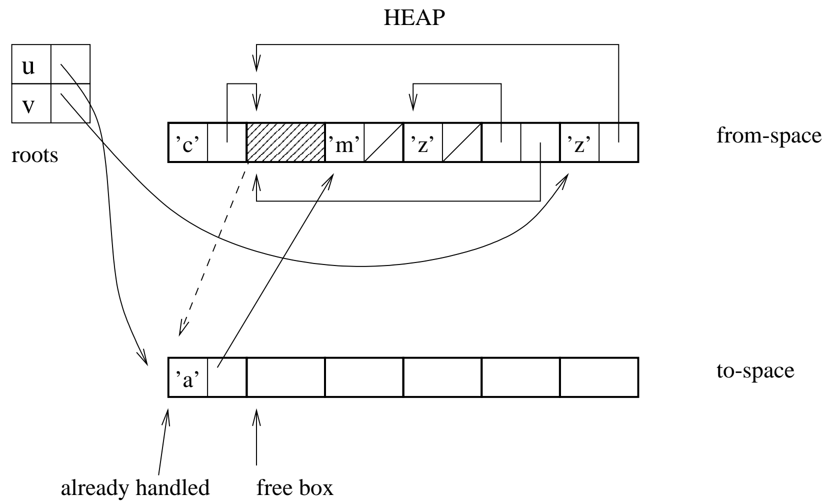


図 9.8: 起領域から至領域への値のコピー

新しく至領域に書き込まれた値もルートとして扱います。まだ処理されていないルートがある限り、値のコピーは続きます。

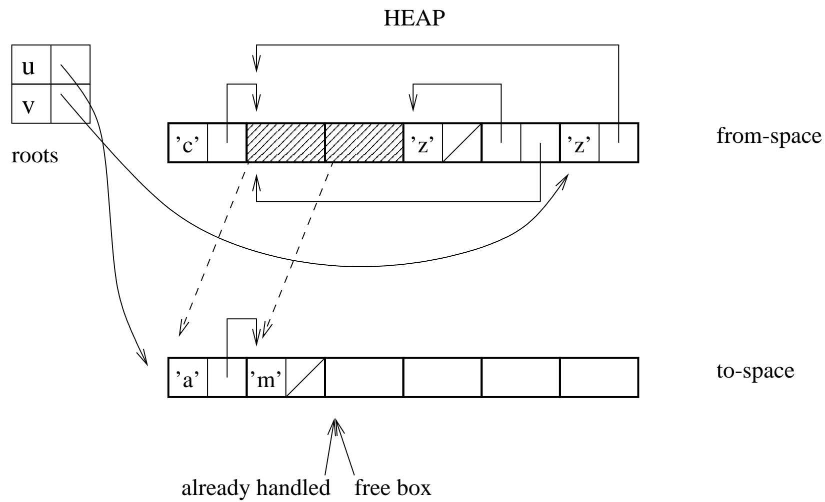


図 9.9: 新しいルート

共有されている値があった場合、言い替えると既にコピーした値を指すポインタがあった場合には新しいアドレスを指すように書き換えます。

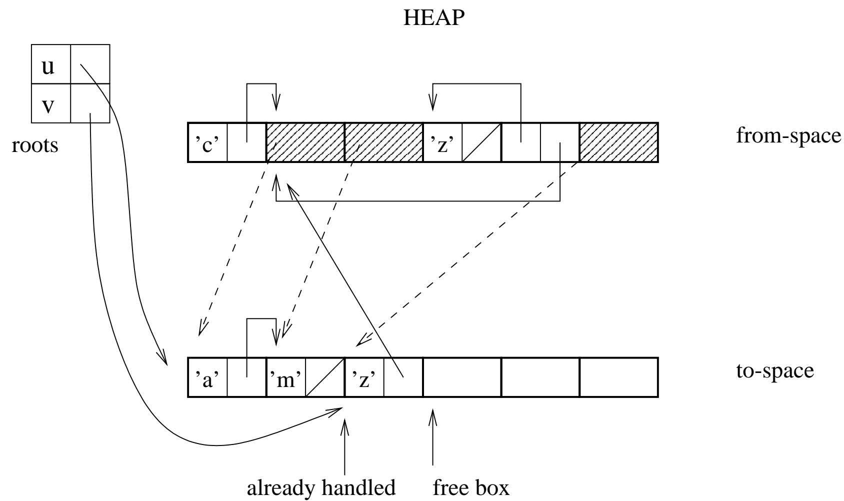


図 9.10: 値の共有

ガーベージコレクションが終了するとすべてのルートの値は更新され、新しい領域のアドレスを参照しています。最後に次のガーベージコレクションに備えて二つの領域の役割を入れ換えます。

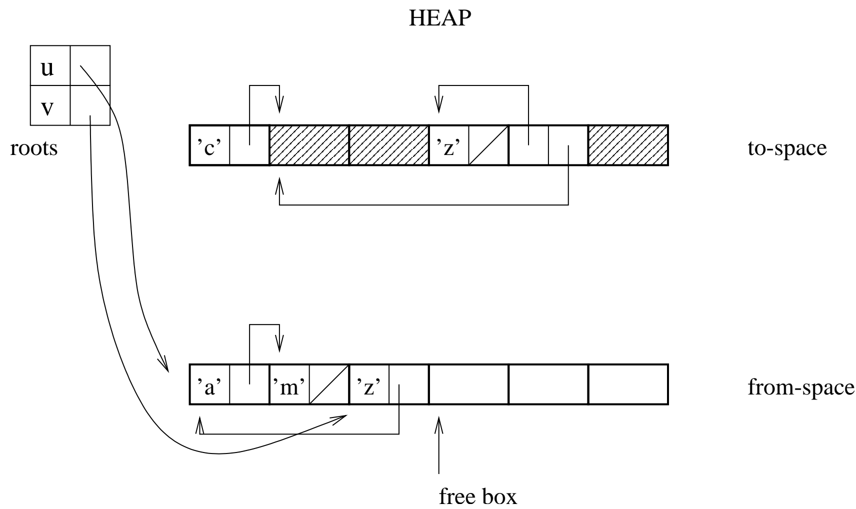


図 9.11: 領域の役割の交換

このガーベージコレクタには次のような特徴があります。

- 計算量は生存している値の大きさに依存します。
- メモリの半分しか利用できません。

- 未使用のメモリは一つの連続領域となります。
- (幅優先探索を行えば) 強い局所性があります。
- スタックのような不定長の領域を必要としません。(ただしメモリを 2 分割したことは除外して考えます。)
- 再帰的なアルゴリズムではありません。
- 値の置かれるアドレスを変更します。

他のガーベージコレクションアルゴリズム

他にも数多くのガーベージコレクションのアルゴリズムが利用されています。多くのものは今紹介した 2 種類のアルゴリズムを拡張したものです。数式処理システムでの巨大配列の操作など、特定のアプリケーションに対して特化されたアルゴリズムやコンパイラ技術と統合させたものなどがあります。多世代ガーベージコレクションは値の寿命に応じた最適化を考えた方式です。保守的ガーベージコレクションは即値とポインタの区別が難しい場合 (例えば C 言語へ変換することによってコンパイルする言語) に使われています。また漸進的ガーベージコレクションはガーベージコレクションが起動された時の待ち時間を減らすことができます。

多世代ガーベージコレクション

関数型言語は頻繁にメモリの確保を行う傾向があります。しかもほとんどの値の生存期間は非常に短いことが分かっています[2]。一方、数回のガーベージコレクションで回収されなかった値はその後非常に長い期間生存する傾向があります。メモリを再利用をする時にマーク・アンド・スイープのようにヒープ全体を辿らずに済ませるために、長い生存期間を持ちガーベージコレクションで回収される可能性の少ない値は辿らないようにする方式が考えられます。オブジェクトに作成時間やガーベージコレクションを経た回数などの情報を書き込んでおき、値の年齢に応じて別のアルゴリズムを使うことができます。

- 若いオブジェクトに対しては高速で、若い世代のみを辿るガーベージコレクションを行います。
- 古いオブジェクトに対するガーベージコレクションは少ない頻度で行い、(少ないと予測される) 使わなくなったオブジェクトを回収するアルゴリズムを使います。

年を取れば取るほどガーベージコレクションで回収されにくくなります。この方式で困難なのは、若いオブジェクトが保存されているメモリ領域のみを辿るようにする方法です。純関数型言語、すなわち代入を持たない言語では若いオブジェクトは古いオブジェクトへの参照を持っていますが、古いオブジェクトは若いオブジェクトへの参照を持っていません。それは、古いオブジェクトは若いオブジェクトが作られる前に作られているからです。したがって古いオブジェクトを探索せずに、すべての若いオブジェクトを探索できます。このような理由から、多世代ガーベージコレクションは純関数型言語に適しています。ただし遅延評価を行う言語では、構造の中身よりも構造の評価を先に行

2. ほとんどの値は最初のガーベージコレクションで回収されてしまいます。

う時があるため古いオブジェクトが若いオブジェクトを参照する場合があり得ます。一方、代入のある関数型言語では古いオブジェクトから若いオブジェクトを指す参照を簡単に作ることができます。そのため古いオブジェクトから指されている若いオブジェクトのリストを常に管理する必要があります。Objective Caml で採用しているアルゴリズムはこの後の節で解説します。

保守的ガーベージコレクション

ここまで出て来たガーベージコレクションの手法ではポインタと即値を区別することが可能であるという前提でアルゴリズムが作られていました。ところが、パラメトリックな多相性を持った関数型言語では値が単一な表現 (一般にメモリの 1 ワード) で表されています。[3]。これは、多相的な関数を型に依存しない一般的なコードで表すために必要な制限です。

しかし単一な表現を使ってしまうと整数とポインタの区別を付けるのが難しくなります。このような場合は保守的なガーベージコレクタを使えば、整数のような即値をポインタとして誤ってマークせずに済みます。保守的なガーベージコレクタではタグビットを使わずにメモリの 1 ワードすべてを値の表現に使えます。実際は整数を表している値をポインタとして誤って辿らないようにするために以下のような観察に基づいた、即値とポインタを区別するアルゴリズムを使います。

- ヒープ領域の開始アドレスと終了アドレスの範囲外の値は即値と判断します。
- すべての割り当てられたオブジェクトはワード境界に置かれています。ワード境界に置かれていない値は即値と判断します。

したがってヒープ上の値はヒープ内の有効なアドレスであれば、例えその値が実際は整数であったとしても、ポインタと判断されその指す領域は回収されません。メモリページごとに同じサイズのオブジェクトを割り当てるようにすれば、即値が有効なアドレスと判断される場合を減らすことができます。未使用領域をすべて回収することはもちろん保証できません。これが保守的なガーベージコレクションの第一の欠点です。しかし、回収されるのは未使用な領域であることは確かです。

一般的に保守的なガーベージコレクションでは値を再配置しません。保守的なガーベージコレクタは即値をポインタと判断することがあるため値の移動は危険な場合があります。それでもルートを作成するアルゴリズムを工夫することで、(明らかにポインタである) ルートに関しては再配置を行えます。

曖昧なルートに対するガーベージコレクションの技術は関数型言語を、ポータブルなアセンブラとしての C 言語にコンパイルした場合にしばしば必要になります。保守的なガーベージコレクションでは C 言語のポインタと即値の表現をそのまま使うことができます。

3. Objective Caml では一つだけ例外があり、浮動小数点数の配列では別の表現を使います。第 12 章 333 ページ参照。

漸進的ガーベージコレクション

ガーベージコレクションに対する批判として良くあるものの一つに、ガーベージコレクションがユーザがはっきり知覚できる時間、プログラムの実行を一時的に停止させてしまう、という批判があります。予測できないタイミングで実行の停止と再開が頻繁に起こるのは、リアルタイムゲームのような高速なインタラクションを行うアプリケーションのユーザにとって受け入れられるものではありません。またガーベージコレクションの停止時間は数秒に達することもあり、制限時間内にイベントを処理しなければならないアプリケーションにとって深刻な問題を引き起こします。例えば乗物などで使われている物理デバイスを制御する組み込みシステムなどでは数秒の遅延は受け入れられません。制限時間内に反応しなければならないリアルタイムシステムでは多くの場合ガーベージコレクションを使いません。

漸進的ガーベージコレクションではメモリの割り当てができる限り任意の時点でガーベージコレクションを中断し、再開できます。これによりリアルタイムゲームであってもガーベージコレクションを利用できます。厳格なリアルタイムシステムであってもガーベージコレクションを使うモジュールと使わないモジュールを分けることでリアルタイム性を保ったままガーベージコレクションを利用可能です。

マーク・アンド・スイープを漸進的にするにはどのように変更すればいいか考えてみましょう。本質的には以下の 2 点について考える必要があります。

1. マークフェーズの途中ですべてのオブジェクトにマークすることを保証するにはどうすればよいか?
2. マークフェーズまたはスイープフェーズの途中で新しいメモリを割り当てるにはどうすればよいか?

マークフェーズの途中でガーベージコレクションが中断された場合、中断されている間に新しく割り当てられた領域はその後のスイープフェーズに誤って回収されてしまうかもしれません。これを防ぐためには中断されている間に割り当てられた領域を黒または灰色で色付けする方法があります。

スイープフェーズの途中でガーベージコレクションが中断された場合は通常通り色の書き換えを再開できます。ただし中断されている間に割り当てられる領域はスイープが中断された時点でフリーリストにある領域から選ばれます。スイープはアドレス順に進行しますが、フリーリストにある領域はスイープが中断した地点よりも前の領域に含まれており、次のガーベージコレクションまでスイープの対象になりません。

図 9.12 はスイープフェーズの途中で新しいメモリの割り当てを行った状態を示しています。新しいルートである w は以下のプログラムによって作られます。

```
# let w = 'f' :: v;;
val w : char list = ['f'; 'z'; 'a'; 'm']
```

Objective Caml でのメモリ管理

Objective Caml のガーベージコレクタはこれまで説明した技法を組み合わせたものです。Objective Caml のガーベージコレクションは 2 世代からなり、若い世代 (マイナー

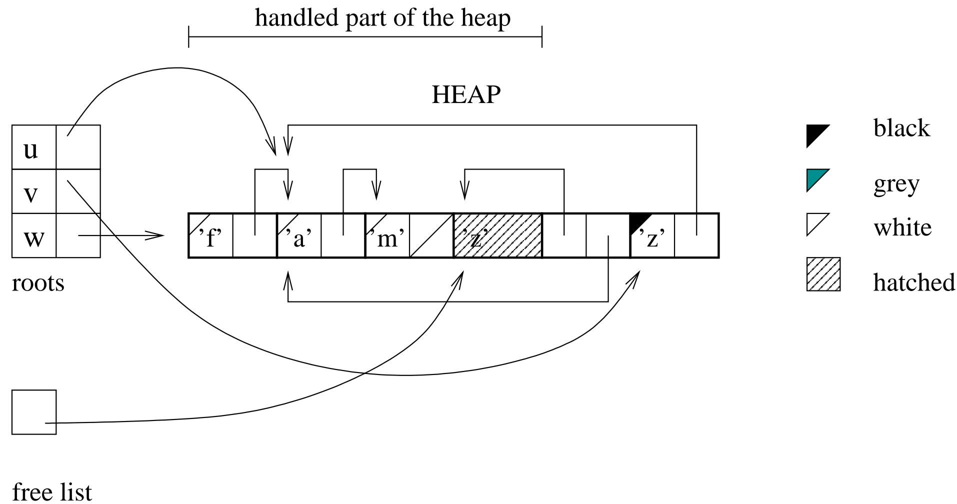


図 9.12: スイープフェーズの途中での割り当て

ガーベージコレクション) では主にストップ・アンド・コピーを使い、古い世代 (メジャーガーベージコレクション) では漸進的なマーク・アンド・スイープを使います。

一度のマイナーガーベージコレクションを生き残った若いオブジェクトは古い世代に移されます。古い世代を対象にストップ・アンド・コピーを使い、ガーベージコレクションの後には `from-space` 全体が解放されます。

多世代ガーベージコレクションについて説明した時に、純粋でない関数型言語での難しさについて注意を行いました。古い世代の値が新しい世代を参照する可能性があります。例えば次のような場合です。

```
# let older = ref [1] ;;
val older : int list ref = {contents = [1]}
(* ... *)
# let newer = [2;5;8] in
    older := newer ;;
- : unit = ()
```

ただしコメント `(* ... *)` によって省略された部分で長い計算があり、その間に `older` は古い世代に移されていると仮定します。マイナーガーベージコレクションでは古い世代から指された若い世代の値を解放してしまわないように配慮しなければなりません。このため古い世代から参照されている若い世代の値の表を作成し、マイナーガーベージコレクションのルートの一部に加えます。実際にこの表のサイズが増大することはほとんどありません。マイナーガーベージコレクションの直後にこの表は空になります。

古い世代に対するマーク・アンド・スイープは漸進的と言いましたが、それはマイナーガーベージコレクションを行う間にメジャーガーベージコレクションを少しだけ実行させるという意味です。メジャーガーベージコレクションのアルゴリズムは 261 ページに説明されている漸進的なマーク・アンド・スイープを基にしています。漸進的アルゴリズムの利点とは、マイナーガーベージコレクションの間にマークフェーズを少しずつ進

めることによってメジャーガーベージコレクションの待ち時間を減らすことにあります。正式なメジャーガーベージコレクションでは、まだマークしていない領域をマークし、再回収を行います。マーク・アンド・スイープでは古い世代にひどいフラグメンテーションを生じる場合があり、その時はメジャーガーベージコレクションの最後にコンパクションを行います。

まとめると、アルゴリズムは以下のようになります。

1. マイナーガーベージコレクション: 若い世代に対してストップ・アンド・コピーを行います。生き残ったオブジェクトは加齢され、古い世代に移されます。古い世代に対してマーク・アンド・スイープを少しだけ進めます。
   古い世代に空き領域がなく、生き残ったオブジェクトをコピーできないときはステップ 2 に進みます。
2. メジャーガーベージコレクションを最後まで進めます。
   これでも空き領域が足らなかった時はステップ 3 に進みます。
3. 漸進的アルゴリズムによってマークされたオブジェクトが解放されていないか調べるため、もう一度メジャーガーベージコレクションを行います。
   これでも空き領域が足らなかった時はステップ 4 に進みます。
4. 連続した空き領域の大きさを最大にするために古い世代に対してコンパクションを行います。もしこれでも空き領域が足らなかった時は諦めてプログラムの実行を停止させます。

GC モジュールを使うとガーベージコレクタの様々なフェーズを起動させることができます。

Objective Caml のメモリ管理機構の中でまだ説明されていないことが一つだけあります。ヒープ全体のサイズは起動時に決定されるのではなく、プログラムの実行状況に応じて増減します。

Gc モジュール

Gc モジュールを使うとヒープに関する統計情報を入手でき、またガーベージコレクションの各フェーズを起動させることなどの制御も行えます。このモジュールでは *stat* と *control* の二つのレコード型が定義されています。*control* のフィールドは変更可能ですが、*stat* はある瞬間のヒープの状態を表しており、フィールドは変更できません。

stat レコードのフィールドは主にカウンタです。

- ガーベージコレクションの回数を表すもの: `minor_collections`, `major_collections`, `compactions`
- プログラムの開始時から割り当てられた総ワード数、コピーされた総ワード数: `minor_words`, `promoted_words`, `major_words`.

control レコードのフィールドは以下のようになっています。

- `minor_heap_size`: 若い世代の大きさ
- `major_heap_increment`: 古い世代の領域を拡張する時の最小の大きさ

- space_overhead: メジャーガーベージコレクションを行わないことによって未使用であるにも関わらず解放されていないメモリの割合。(デフォルト値は 80) この値からメジャーガーベージコレクションを行う頻度を決定します。
- max_overhead: 無駄な (未使用であるが解放されていない) 領域の大きさがこの値を越えた時にコンパクションを行います。この値を 0 にするとメジャーガーベージコレクションの後に必ずコンパクションを行います。この値を 1000000 にするとコンパクションを行わなくなります。
- verbose: ガーベージコレクタの動作を表示させる方法を指定します。

図 9.13 に *stat* と *control* を扱う関数をいくつか示します。

stat	*unit* → *stat*
print_stat	*out_channel* → *unit*
get	*unit* → *control*
set	*control* → *unit*

図 9.13: ヒープを制御し、統計情報を得る関数

以下の 4 つの関数はすべて型が *unit -> unit* であり、Objective Caml のガーベージコレクタの一つまたは複数のフェーズを実行させます。それらは minor (フェーズ 1)、major (フェーズ 1 と 2)、full_major (フェーズ 1 から 3)、compact (フェーズ 1 から 4) の 4 つの関数です。

例

Gc.stat を呼び出した結果は以下のようになります。

```
# Gc.stat();;
- : Gc.stat =
{Gc.minor_words = 1027347; Gc.promoted_words = 131663;
 Gc.major_words = 262302; Gc.minor_collections = 33;
 Gc.major_collections = 3; Gc.heap_words = 253952; Gc.heap_chunks = 4;
 Gc.live_words = 188700; Gc.live_blocks = 43317; Gc.free_words = 65201;
 Gc.free_blocks = 427; Gc.largest_free = 25852; Gc.fragments = 51;
 Gc.compactions = 0; Gc.top_heap_words = 253952}
```

結果にはマイナーガーベージコレクション、メジャーガーベージコレクション、コンパクションの実行回数や、それぞれのメモリ領域で処理されたワード数等が含まれています。compact 関数を呼び出し、ガーベージコレクタの各フェーズを実行させてみましょう。

```
# Gc.compact();;
- : unit = ()
# Gc.stat();;
- : Gc.stat =
{Gc.minor_words = 1043434; Gc.promoted_words = 131663;
```

```
 Gc.major_words = 264281; Gc.minor_collections = 33;
 Gc.major_collections = 5; Gc.heap_words = 253952; Gc.heap_chunks = 4;
 Gc.live_words = 158783; Gc.live_blocks = 35367; Gc.free_words = 95169;
 Gc.free_blocks = 2; Gc.largest_free = 63488; Gc.fragments = 0;
 Gc.compactions = 1; Gc.top_heap_words = 253952}
```

GC.minor_collections フィールドと compactions フィールドの値がそれぞれ一つずつ増えています。一方 Gc.major_collections フィールドの値は 2 つ増えています。*GC.control* 型のレコードのすべてのフィールドは変更可能ですが、言語のランタイムに反映させるためには Gc.set 関数を呼び出さなければなりません。この関数は *control* 型の値を受け取りガーベージコレクタの動作を変更します。

例えば verbose フィールドは 9 種類の指示の組合せの 0 から 1023 までの値を受け取ります。ここでは 31 を代入してみます。

```
# let c = Gc.get();;
val c : Gc.control =
  {Gc.minor_heap_size = 32768; Gc.major_heap_increment = 63488;
   Gc.space_overhead = 80; Gc.verbose = 0; Gc.max_overhead = 500;
   Gc.stack_limit = 262144}
# c.Gc.verbose <- 31;;
- : unit = ()
# Gc.set c;;
- : unit = ()
# Gc.compact();;
<>Starting new major GC cycle
Starting new major GC cycle
Compacting heap...
done.
- : unit = ()
```

ガーベージコレクタの各フェーズの実行が表示されるようになりました。

Weak モジュール

弱いポインタ *weak pointer* とはガーベージコレクタによっていつでも回収される可能性のある領域を指しているポインタです。ポインタの指す値が不意に消滅してしまうかもしれない機能があるのは不思議と感じるかもしれません。しかしこのような弱いポインタは値の一時的な保管機能を実装するのに必須のものです。特に、保存しなければならない要素よりもメモリが少ない時に極めて有用であることが分かります。良く引き合いに出される例はメモリキャッシュです。キャッシュ上の値はある時消滅してしまうかもしれませんが、値がある限り高速にアクセスできます。

Objective Caml では弱いポインタを直接扱うことはできず、弱いポインタの配列を使います。Weak モジュールは抽象型 *'a Weak.t* を定義しています。'a 型の弱いポインタの配列は *'a option array* 型を持ちます。*'a option* 型の定義は以下のようになります。

```
type 'a option = None | Some of 'a;;
```

図 9.14 にこのモジュールの主な関数が定義されています。

関数	型
create	*int -> 'a t*
set	*'a t -> int -> 'a option -> unit*
get	*'a t -> int -> 'a option*
check	*'a t -> int -> bool*

図 9.14: Weak モジュールの主要な関数

create 関数はすべての要素が None で初期化された弱いポインタの配列を生成します。set 関数は *'a option* 型の値を配列の指定されたインデックスの要素に設定します。get 関数は配列の指定されたインデックスの要素を返します。読み出された値を保持している限り、ガーベージコレクションによって回収されません。値が実際に存在しているかどうか確かめるには check 関数を使うか、または get 関数の返り値をパターンマッチして調べます。

順序構造を扱うための標準的な関数も用意されています。配列のサイズを知るには length 関数を使います。配列のすべてまたは一部を埋めるには fill 関数や blit 関数を使います。

例: 画像イメージのキャッシュ

画像処理のアプリケーションでは一度に複数の巨大な画像を扱うことは稀ではありません。ユーザが処理の対象をある画像から別の画像へと移すと、以前の画像をファイルに保存し、新しい画像を読み込みます。ファイルに保存された画像はその名前だけが管理されています。ディスクアクセスをできるだけ減らすと同時にメモリ使用量も抑えるために良くアクセスされる画像をメモリ上に置く画像キャッシュを作ってみましょう。キャッシュの内容は必要に応じていつでも消去されます。この機能を弱いポインタの表を使って実装します。キャッシュの内容の破棄の部分はガーベージコレクタに任せます。画像をロードする時はまず同じ画像があるかどうかキャッシュを探し、もしあればその内容を返し、なければファイルから読み込みます。

キャッシュの表は以下のように定義します。

```
# type table_of_images = {
    size : int;
    mutable ind : int;
    mutable name : string;
    mutable current : Graphics.color array array;
    cache : ( string * Graphics.color array array) Weak.t } ;;
```

size フィールドは表の大きさを表しています。ind フィールドは現在の画像を格納しているインデックスを指しています。name フィールドは現在の画像の名前を表しています。current フィールドは現在の画像を保持しています。cache フィールドは画像を指す弱いポインタの配列を保持しています。この配列の要素は画像の名前と画像本体の組です。

init_table 関数は最初の画像で表を初期化します。

```
# let open_image filename =
    let ic = open_in filename
    in let i = ((input_value ic) : Graphics.color array array)
    in ( close_in ic ; i ) ;;
val open_image : string -> Graphics.color array array = <fun>

# let init_table n filename =
    let i = open_image filename
    in let c =  Weak.create n
    in Weak.set c 0 (Some (filename,i)) ;
       { size=n; ind=0; name = filename; current = i; cache = c } ;;
val init_table : int -> string -> table_of_images = <fun>
```

新しい画像を読み込む前にまず現在の画像を保存します。新しい画像がもしキャッシュにあれば、ファイルから読み込まずキャッシュの内容を返します。

```
# exception Found of int * Graphics.color array array ;;
# let search_table filename table =
    try
      for i=0 to table.size-1 do
        if i<>table.ind then match Weak.get table.cache i with
                Some (n,img) when n=filename  → raise (Found (i,img))
              | _ → ()
        done ;
      None
    with Found (i,img) → Some (i,img) ;;



# let load_table filename table =
    if table.name = filename then ()  (* the image is the current image *)
    else
      match search_table filename table with
        Some (i,img) →
        (* the image found becomes the current image *)
          table.current <- img ;
          table.name <- filename ;
          table.ind <- i
      | None →
        (* the image isn't in the cache, need to load it *)
        (* find an empty spot in the cache *)
          let i = ref 0 in
            while (!i<table.size && Weak.check table.cache !i) do incr i done ;
            (* if none are free, take a full slot *)
              ( if !i=table.size then i:=(table.ind+1) mod table.size ) ;
              (* load the image here and make it the current one *)
              table.current <- open_image filename ;
```

```
          table.ind <- !i ;
          table.name <- filename ;
          Weak.set table.cache table.ind (Some (filename,table.current)) ;;
val load_table : string -> table_of_images -> unit = <fun>
```

load_table 関数はまず要求された画像が現在の画像かどうか調べます。違う時はキャッシュにその画像があるかどうか調べます。キャッシュにない時はディスクから画像を読み込みます。どちらの場合でも要求された画像を現在の画像とします。

このプログラムの動作を確認するためにキャッシュの内容を表示させる関数を使います。

```
# let print_table table =
    for i = 0 to table.size-1 do
      match Weak.get table.cache ((i+table.ind) mod table.size) with
        None → print_string "[] "
      | Some (n,_) → print_string n ; print_string " "
    done ;;
val print_table : table_of_images -> unit = <fun>
```

画像を一枚読み込み、キャッシュの内容を表示させてみましょう。Then we test the following program:

```
# let t = init_table 10 "IMAGES/animfond.caa" ;;
val t : table_of_images =
  {size = 10; ind = 0; name = "IMAGES/animfond.caa";
   current =
    [|[|7371936; 7371936; 7371936; 7371936; 7371936; 7371936; 7371936;
        7371936; 7371936; 7371936; 7371936; 7371936; 7373984; 7373984; ...|];
      ...|];
   cache = ...}
# load_table "IMAGES/anim.caa" t ;;
- : unit = ()
# print_table t ;;
IMAGES/anim.caa [] [] [] [] [] [] [] [] [] - : unit = ()
```

このキャッシュのアイデアは様々なアプリケーションに応用することができます。

練習問題

ヒープの変化の追跡

ヒープの変化を追うためにヒープの情報を以下のようなフォーマットのレコード型として保持する関数を書いてみましょう。

```
# type tr_gc = {state : Gc.stat;
                time : float; number : int};;
```

ここでの時間はプログラムが開始されてからの時間をミリ秒で表した数です。number

フィールドはこのレコードに対して与えられる通し番号です。時間を得るには `Sys.time` 関数 (234 ページ参照) を使います。

1. このようなレコードを生成する `trace_gc` 関数を書きましょう。
2. この関数が *tr_gc* 型のレコードをファイルに永続的な形式で書き込めるように書き換えなさい。 新しい関数は書き出すための出力チャネルを引数として受け取ります。レコードを保存するには 228 ページで記述されている `Marshal` モジュールを使います。
3. *tr_gc* 型のレコードが保存されたファイルを受け取り、時間の経過にそってメジャーガーベージコレクションとマイナーガーベージコレクションの回数を表示する単体のプログラムを書きなさい。
4. トレースファイルをインタープリタを使って作成し、内容を表示させてみましょう。

メモリ使用量とプログラミングスタイル

この演習ではメモリ使用量に対するプログラミングスタイルの影響について比較します。このために第 8 章 245 ページで扱った素数に関する演習を再考してみましょう。この演習ではエラトステネスのふるいを末尾再帰で書いたものと末尾再帰を使わずに書いたものを比べます。

1. 指定された区間に含まれるすべての素数を末尾再帰的に計算する関数 `erart` (この名前は変更した方がよいでしょう) を書きましょう。次に与えられた整数よりも小さい素数を出力する関数を書きましょう。
2. 以上の関数を利用してプログラム `era2main` (この名前は変更した方がいいでしょう) を作りなさい。このプログラムはファイル名と数のリストを受け取り、指定された一つの数ごとにそれよりも小さな素数のリストを計算します。計算の途中でのガーベージコレクションのトレースを指定されたファイルに保存します。トレースを取るには前の演習の結果を利用します。
3. 以上のプログラムをコンパイルして単体のコマンドを作りなさい。 そのコマンドを以下の引数で実行し、トレースを表示させてみましょう。

```
%
erart trace_rt 3000 4000 5000 6000
```

4. 同じことを末尾再帰を使わない関数に対して行いなさい。
5. 両者のトレースを比べてみましょう。

まとめ

この章では、Objective Caml のガーベージコレクタについて解説するという趣旨から自動メモリ再回収機構の代表的なアルゴリズムを紹介しました。Objective Caml のガーベージコレクタは 2 世代を持つ漸進的アルゴリズムを採用しています。古い世代にはマーク・アンド・スイープを行い、若い世代にはストップ・アンド・コピーを行います。ガーベージコレクタと密接に結び付いた 2 つのモジュールを使って、ヒープの振る舞いを制御することができます。`Gc` モジュールは、特定のアプリケーションに対してガーベージ

コレクタを最適化するためにガーベージコレクタに関する統計情報を提供し、ガーベージコレクタのアルゴリズムのパラメータを制御できます。`Weak` モジュールは要素が再回収される可能性のある配列を提供しています。このモジュールはメモリキャッシュを実装するのに重宝します。

もっと知りたい人へ

メモリ管理の技法はプログラミング言語 Lisp の最初の実装から既に 40 年以上に渡って研究されてきました。このため膨大な量の文献があります。

ガーベージコレクションに関する包括的な資料は Jones の本 [Jon98] です。Paul Wilson のチュートリアル [Wil92] は多くの資料を参照しており、この分野への優れた入門書となっています。以下のウェブページもメモリ管理の現状について優れた資料を提供しています。

リンク: ftp://ftp.netcom.com/pub/hb/hbaker/home.html

このページは逐次的なガーベージコレクタについて分かりやすく紹介しています。

リンク: http://www.cs.ukc.ac.uk/people/staff/rej/gc.html

このページは Jones の本 [Jon98] について解説しています。検索可能な巨大な文献リストも備えています。

リンク: http://www.cs.colorado.edu/~zorn/DSA.html

このページはガーベージコレクションをデバグするための様々なツールを紹介しています。

リンク: http://reality.sgi.com/boehm_mti/

このページは C 言語のための保守的なガーベージコレクタのソースコードを提供しています。このガーベージコレクタは C 言語の標準のメモリ割り当て関数 `malloc` を自前の関数 `GC_malloc` で置き換えます。明示的なメモリ領域の解放を行う関数 `free` は何もしない関数と置き換えられます。

リンク: http://www.harlequin.com/mm/reference/links.html

このページにはメモリ管理に関するリンクがあります。

第 12 章で C 言語と Objective Caml のインターフェースについて説明する時に再びメモリ管理の話題について扱います。

10

プログラム解析ツール

プログラム解析ツールはプログラマに対してコンパイラやリンカが提供する情報を補う様々な有益な情報を提供できます。いくつかのツールは静的解析を行いモジュール間の相互依存関係や捕捉されない例外などを調査します。他のツールは実行時の解析を行い、関数の呼び出された回数や引数の値、関数の実行時間などの情報を提供します。デバッガのような対話的なツールもあります。デバッガを使うと特定の地点にブレークポイントを設定でき、その地点での変数の値を調査したり、引数を変えて繰り返し実行させたりできます。

これらのツールは Objective Caml 配布パッケージに含まれています。いくつかのツールは他の言語の標準的なツールと比べて変わった動作をする場合がありますが、この原因には静的な型システムに起因するものが多くあります。静的な型システムのおかげで型チェックなしに安全で効率の良いコードを生成できますが、実行時には値の型情報が失われている場合があります。このため多相的な関数の引数を表示できない等の不都合があります。

この章のあらまし

この短い章では Objective Caml 配布パッケージに含まれているプログラム解析ツールについて紹介します。まず最初に一つのアプリケーションプログラムに含まれるファイルの相互依存関係を調べる `ocamldep` コマンドについて解析します。

次にプログラムの実行トレースを取るツールと Unix 環境で動作するデバッガについて扱います。

最後に最適化を行う時に有益な関数の実行回数や実行時間などの情報を解析するプロファイラについて触れます。

依存性解析

大規模な Objective Caml のアプリケーションは通常多くのプログラムファイルとインターフェースファイルによって構成されています。それらのファイルの間の依存性解析には二つの目的があります。一つはモジュール間の相互依存関係について理解をすること。もう一つはあるファイルを変更したときに再コンパイルする必要のあるファイルを知ることです。

ocamldep コマンドは.ml ファイルと .mli ファイルを引数として受け取り、Makefile[1] フォーマットの依存性情報を出力します。

他のモジュールへの依存は具体的にはドット記法 (M1.f) や他のモジュールを開くこと (**open** M1) によって生じます。

以下のような内容を含んだ二つのファイルがあるとします。

dp.ml :

```
let print_vect v =
    for i = 0 to Array.length v do
      Printf.printf "%f " v.(i)
    done;
    print_newline();;
```

d1.ml :

```
let init n e =
  let v = Array.create 4 3.14 in
    Dp.print_vect v;
    v;;
```

これらのファイル名を ocamldep コマンドに与えると以下のような依存性についての結果を出力します。

```
$ ocamldep dp.ml d1.ml array.ml array.mli printf.ml printf.mli
dp.cmo: array.cmi printf.cmi
dp.cmx: array.cmx printf.cmx
d1.cmo: array.cmi dp.cmo
d1.cmx: array.cmx dp.cmx
array.cmo: array.cmi
array.cmx: array.cmi
printf.cmo: printf.cmi
printf.cmx: printf.cmi
```

バイトコードとネイティブコードの両方に対して依存性情報を出力します。依存性情報は行ごとに一つの項目を表しており、コロン (:) の右側にあるものは左側の項目に依存

1. Makefile は make コマンドによって使われているファイルで、プログラムを変更したときに再コンパイルする手順が記録されています。

していることを表しています。dp.cmo ファイルは array.cmi ファイルと printf.cmi ファイルに依存しています。暗黙の依存関係として拡張子が .cmi のファイルは同じ名前で拡張子が.mli のファイルに依存しています。同様に拡張子が.cmo または .cmx のファイルは.ml ファイルに依存しています。

ディストリビューションに含まれるオブジェクトファイルは依存性情報の中に表示されません。実際もし array.ml ファイルと printf.ml ファイルが現在のディレクトリになければ ocamldep はシステムライブラリからモジュールの情報を取得し、以下のような結果を出力することになるでしょう。

```
$ ocamldep dp.ml d1.ml
d1.cmo: dp.cmo
d1.cmx: dp.cmx
```

ocamldep コマンドがファイルを検索するディレクトリを追加するには-I オプションを使います。

デバッグツール

デバッグのためのツールには二つあります。一つ目のツールはインタープリタで利用できるトレースの機能です。もう一つのツールはいわゆるデバッガです。ブレークポイントでプログラムを停止させたり、引数を変えてプログラムを再実行させたりできます。この二番目のツールは fork システムコール (586 ページ参照) を利用してプロセスを複製するので Unix 環境でのみ利用できます。

Trace（トレース）

ある関数の**トレース**とはプログラムを実行中にその関数が呼ばれた時の引数と帰り値の値の記録のことです。

トレースコマンドはトップレベルのディレクティブです。トレースコマンドを使ってある関数のトレースを行うようにしたり、また止めるように指定できます。トレースコマンドには以下の 3 種類があります。

#trace 関数名	指定された関数のトレースを開始する
#untrace 関数名	指定された関数のトレースを終了する
#untrace_all	すべてのトレースを終了する

簡単な関数を使ってトレースの実演をしてみましょう。

```
# let f x = x + 1;;
val f : int -> int = <fun>
# f 4;;
- : int = 5
```

関数 f のトレースを開始して、引数と帰り値の値が表示されるようにします。

```
#  #trace f;;
f is now traced.
# f 4;;
f <-- 4
f --> 5
- : int = 5
```

関数 f に引数 4 が適用され、帰り値が返されたことが表示されています。関数の引数は左向きの矢印で表され、帰り値は右向きの矢印で表されています。

2 つ以上の引数を持つ関数

複数の引数を持つ関数（部分適用が起こる関数）も同様にトレースできます。すべての引数は表示されますが、何回目の適用であるか示すために*記号を関数に付加しています。4 つの引数 (a, b, q, r) を受け取る関数 verif_div を考えます。この関数は整数の割算が正しいかどうか検算 $a = bq + r$ を行います。

```
# let verif_div a b q r =
    a = b*q + r;;
val verif_div : int -> int -> int -> int -> bool = <fun>
# verif_div 11 5 2 1;;
- : bool = true
```

4 つの値を適用した結果は以下のようになります。

```
#  #trace verif_div;;
verif_div is now traced.
# verif_div 11 5 2 1;;
verif_div <-- 11
verif_div --> <fun>
verif_div* <-- 5
verif_div* --> <fun>
verif_div** <-- 2
verif_div** --> <fun>
verif_div*** <-- 1
verif_div*** --> true
- : bool = true
```

再帰関数

再帰関数の実行トレースには、実行効率についての価値ある情報が含まれています。再帰実行の終了条件に問題があればトレースからそのヒントが得られるでしょう。

関数 belongs_to は与えられた整数のリストに指定された整数が含まれているかどうか判定する再帰的な関数です。この関数の定義は以下のようになります。

```
# let rec belongs_to (e : int) l = match l with
    [] → false
```

```
  | t :: q -> (e = t) || belongs_to e q ;;
val belongs_to : int -> int list -> bool = <fun>
# belongs_to 4 [3;5;7] ;;
- : bool = false
# belongs_to 4 [1; 2; 3; 4; 5; 6; 7; 8] ;;
- : bool = true
```

関数に値を適用した式 belongs_to 4 [3;5;7] の実行結果を見ると、関数呼び出しと関数からの復帰がそれぞれ 4 回ずつ起こっています。

```
#  #trace belongs_to ;;
belongs_to is now traced.
# belongs_to 4 [3;5;7] ;;
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [3; 5; 7]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [5; 7]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [7]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- []
belongs_to* --> false
belongs_to* --> false
belongs_to* --> false
belongs_to* --> false
- : bool = false
```

関数 belongs_to が呼ばれるごとに引数の 4 と検索されるリストが渡されてます。リストが空になった時にこの関数は false を帰り値として返しています。この値は再帰的に呼び出している関数に渡されていっています。

以下の例では値が見付かった場合を示しています。

```
# belongs_to 4 [1; 2; 3; 4; 5; 6; 7; 8] ;;
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [1; 2; 3; 4; 5; 6; 7; 8]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [2; 3; 4; 5; 6; 7; 8]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [3; 4; 5; 6; 7; 8]
belongs_to <-- 4
```

```
belongs_to --> <fun>
belongs_to* <-- [4; 5; 6; 7; 8]
belongs_to* --> true
belongs_to* --> true
belongs_to* --> true
belongs_to* --> true
- : bool = true
```

リストの先頭に 4 が来ると、この関数はすぐに true を返しています。この値は再帰的に呼び出している関数に渡されていっています。

論理和 || の両辺の順序が変わると、関数 belongs_to の呼び出しの結果は同じですが常にリストをすべて検索するようになります。

```
# let rec belongs_to (e : int) = function
      [] → false
    | t :: q →   belongs_to e q || (e = t) ;;
val belongs_to : int -> int list -> bool = <fun>
# #trace belongs_to ;;
belongs_to is now traced.
# belongs_to 3 [3;5;7] ;;
belongs_to <-- 3
belongs_to --> <fun>
belongs_to* <-- [3; 5; 7]
belongs_to <-- 3
belongs_to --> <fun>
belongs_to* <-- [5; 7]
belongs_to <-- 3
belongs_to --> <fun>
belongs_to* <-- [7]
belongs_to <-- 3
belongs_to --> <fun>
belongs_to* <-- []
belongs_to* --> false
belongs_to* --> false
belongs_to* --> false
belongs_to* --> true
- : bool = true
```

リストの先頭に 3 が来ていても、すべてのリストが検索されています。このように実行トレースの情報から再帰関数の効率について考えることができます。

多相関数

パラメータ化された型に対応する引数の値は実行トレースでは表示されません。例として関数 belongs_to を明示的な型宣言なしに書き直してみましょう。

```
# let rec belongs_to e l = match l with
     [] → false
   | t :: q → (e = t) || belongs_to e q ;;
val belongs_to : 'a -> 'a list -> bool = <fun>
```

関数 belongs_to は多相的になりました。実行トレース中では引数の値に代わって (poly)

とだけ表示するようになりました。

```
#  #trace belongs_to ;;
belongs_to is now traced.
# belongs_to 3 [2;3;4] ;;
belongs_to <-- <poly>
belongs_to --> <fun>
belongs_to* <-- [<poly>; <poly>; <poly>]
belongs_to <-- <poly>
belongs_to --> <fun>
belongs_to* <-- [<poly>; <poly>]
belongs_to* --> true
belongs_to* --> true
- : bool = true
```

Objective Caml インタープリタのトップレベルでは単相的な値しか表示できません。トップレベルは大域的な宣言の (推論された) 型しか保存していません。このためトップレベルは関数 belongs_to の型を *'a -> 'a list -> bool* であるとしか認識していません。式 belongs_to 3 [2;3;4] をコンパイルした後は 3 と [2;3;4] の型は忘れられてしまいます。これは Objective Caml が静的な型システムを持っているためであり、このために実行トレース中に多相的な値を表示することができないのです。

Objective Caml で総称的な print 関数 (*'a -> unit*) を定義できないのもまさにこの理由からです。

局所関数

局所関数に対しても上で述べたのと全く同じ理由によりトレースを表示することができません。トップレベルでは大域的な宣言の型のみを保持しています。以下のようなプログラミングスタイルに人気があることは分かっています。

```
# let belongs_to e l =
    let rec bel_aux l = match l with
      []  → false
    | t :: q → (e = t) || (bel_aux q)
    in
      bel_aux l;;
val belongs_to : 'a -> 'a list -> bool = <fun>
```

大域的関数は局所関数を呼び出すだけであり、本質的な仕事は局所関数の中で行われています。

トレースについての註

実行トレースとは実質的にプラットフォームに依存しないデバッグツールとなっています。トレースの弱点とは局所関数と多相関数の値を表示できないことです。この弱点は、特にプログラミング言語は学習していく過程で大きな障害となるかもしれません。

デバッグ

ocamldebug はいわゆる**デバッガ**です。命令のステップ実行、ブレークポイントの挿入、環境の中の値の表示、変更などを行うことができます。

プログラムのステップ実行と言った場合には「ステップとは何か」という定義が前もって必要になります。命令型言語の場合は簡単です。1 ステップとは 1 命令 (または 1 操作) に対応しています。しかし関数型言語ではこの定義ではうまくいきません。関数型言語ではイベントを単位として考えます。イベントとは関数適用、パターンマッチ、条件判定、ループなどを指しています。

警告 このツールは Unix 上でのみ利用できます。

デバッグモードでのコンパイル

-g オプションを付けてコンパイルすると、デバッグのために必要な命令を含んだ.cmo ファイルを生成します。プログラムのデバッグを行うには必ずこのオプションを付けてプログラムのコンパイルを行う必要があります。実行ファイルが生成されたら ocamldebug を使ってプログラムを起動します。

ocamldebug [*options*] executable [*arguments*]

階乗を計算する以下のプログラムを fact.ml というファイルに保存します。

```
let fact n =
  let rec fact_aux p q n =
    if n = 0 then p
    else fact_aux (p+q) p (n-1)
  in
fact_aux 1 1 n;;
```

メインプログラムを main.ml に保存します。このプログラムは非常に長い再帰呼び出しを作り出すことになります。

```
let x = ref 4;;
let go () =
  x := -1;
  Fact.fact !x;;
go();;
```

-g オプションを付けて二つのファイルをコンパイルします。

```
$ ocamlc -g -i -o fact.exe fact.ml main.ml
val fact : int -> int
val x : int ref
val go : unit -> int
```

デバッガの起動

デバッグモードでプログラムがコンパイルされたらデバッガを使って実行ファイルを起動します。

```
$ ocamldebug fact.exe
        Objective Caml Debugger version 3.00

(ocd)
```

実行の制御

実行の制御はプログラムイベントを通して行います。n 個前 (または n 個後) のプログラムイベントまで実行を戻す (または進める) ことができます。ブレークポイントについても同様です。ブレークポイントは関数またはプログラムイベントに設定できます。対応するソースコードはファイルの行数とカラム数によって示されます。

以下の例ではブレークポイントは `Main` モジュールの 4 行目に設定されています。

```
(ocd) step 0
Loading program... done.
Time : 0
Beginning of program.
(ocd)  break @ Main 4
Breakpoint 1 at 5028 : file Main, line 4 column 3
```

モジュールの初期化はプログラムの実行の前に行われます。4 行目に設定されたブレークポイントが 5028 番目の命令になっているのは初期化があるためです。

プログラムイベントまたはブレークポイントを指定してプログラムの実行を進めたり、あるいは巻き戻したりすることができます。`run` によって次のブレークポイントまで実行を進めます。`reverse` によって一つ前のブレークポイントまで実行を巻き戻します。`step` によって次のプログラムイベントまたは n 個先のプログラムイベントまで実行を進めます。`next` は `step` と同じですが、関数呼び出し全体を 1 イベントと数えます。`backstep` と `previous` はそれぞれ `step` と `next` と同じことを逆方向に行います。`finish` は現在の関数から戻るまで実行を進めます。`start` は現在の関数が呼び出される直前まで実行を戻します。

例の中で次のブレークポイントまで実行を進め、その後プログラムイベントを 3 回進めます。

```
(ocd) run
Time : 6 - pc : 4964 - module Main
Breakpoint : 1
4   <|b|>Fact.fact !x;;
(ocd) step
Time : 7 - pc : 4860 - module Fact
```

```
2   <|b|>let rec fact_aux p q n =
(ocd) step
Time : 8 - pc : 4876 - module Fact
6 <|b|>fact_aux 1 1 n;;
(ocd) step
Time : 9 - pc : 4788 - module Fact
3     <|b|>if n = 0 then p
```

値の表示

ブレークポイントでスタック上 (activation record) の変数の値を表示させることができます。`print` と `display` によって変数の値を表示させることができます。`display` は階層的なデータ構造のもっとも浅い部分しか表示しません。

変数 `n` の値を表示させ、それから 3 ステップ実行を戻して、変数 `x` の値を表示させてみましょう。

```
(ocd) print n
n : int = -1
(ocd) backstep 3
Time : 6 - pc : 4964 - module Main
Breakpoint : 1
4   <|b|>Fact.fact !x;;
(ocd) print x
x : int ref = {contents=-1}
```

これらの表示コマンドではレコードのフィールドや配列の要素への参照を書くことができます。

```
(ocd) print x.contents
1 : int = -1
```

実行スタック

関数呼び出しの状態は実行スタックによって表現されています。`backtrace`(または短縮形の `bt`) コマンドによって関数呼び出しの状況を表示させることができます。`up` と `down` コマンドによってそれぞれ次、または直前のスタックフレームを選択することができます。`frame` コマンドによって現在選択されているスタックフレームの内容を表示させることができます。

プロファイリング

このツールはプログラムの実行について、特定の関数の実行回数などの制御構造 (条件文、パターンマッチ、ループ) に関する様々な統計情報を入手できます。結果はファイルに保存されます。この情報を利用すればアルゴリズムの間違いや、最適化に関するボトルネックなどを発見することができるでしょう。

プロファイリングを行うためには、プロファイリングのための専用の命令を含んだコードを生成する必要があります。そのためには特別なオプションを付けてプログラムをコンパイルしなければなりません。機械語でのプロファイリングでは個々の関数の実行時間に関する情報も取得できます。

アプリケーションプログラムのプロファイリングは次のような手順で行います。

1. プロファイリングオプションを付けてアプリケーションをコンパイルする。
2. プログラムを実行する。
3. プロファイリングの結果を出力する。

コンパイル方法

プロファイリングのためのコードを生成するには次のように指定します。

- `ocamlcp -p` バイトコードコンパイラの場合。
- `ocamlopt -p` 機械語コードコンパイラの場合。

これらのツールは通常のコンパイラによって生成されるファイル (第 7 章参照) と全く同じ種類のファイルを生成します。サブオプションには図 10.1 に挙げたものがあります。

f	関数呼び出し
i	**if** による分岐
l	**while** と **for** によるループ
m	**match** による分岐
t	**try** による分岐
a	以上のすべて

図 10.1: プロファイリングツールのオプション

これらのオプションは情報を取得する対象となる制御構造を指定しています。デフォルトでは `fm` オプションが付いていると解釈されています。

プログラムの実行

バイトコードコンパイラの場合

プロファイリングオプションを付けてコンパイルされたプログラムは終了した時に、プロファイリングに必要な情報の入っている `ocamlprof.dump` というファイルを生成します。

整数のリストの乗算の例を使って説明します。以下のプログラムが `f1.ml` というファイルに保存されているとします。

```
let rec interval a b =
  if b < a then []
```

```
  else a :: (interval (a+1) b);;

exception Found_zero ;;

let mult_list l =
 let rec mult_rec l = match l with
     [] → 1
   | 0::_ → raise Found_zero
   | n :: x → n * (mult_rec x)
 in
  try mult_rec l with Found_zero → 0
;;
```

同様に以下のプログラムが f2.ml というファイルに保存されているとします。

```
let l1 = F1.interval 1 30;;
let l2 = F1.interval 31 60;;
let l3 = l1 @ (0 :: l2);;

print_int (F1.mult_list l1);;
print_newline();;

print_int (F1.mult_list l3);;
print_newline();;
```

これらのファイルをプロファイリングモードでコンパイルすると次のようになります。

```
ocamlcp -i -p a -c f1.ml
val profile_f1_ : int array
val interval : int -> int -> int list
exception Found_zero
val mult_list : int list -> int
```

-p オプションを付けるとコンパイラは新しい関数 (profile_f1_) を追加します。この関数はプロファイリングのためのカウンタの初期化を行います。f2.ml ファイルのコンパイルでも同様です。

```
ocamlcp -i -p a -o f2.exe f1.cmo f2.ml
val profile_f2_ : int array
val l1 : int list
val l2 : int list
val l3 : int list
```

機械語コードコンパイラの場合

機械語コードコンパイラでコンパイルすると以下のような結果になります。The native code compilation gives the following result:

```
$ ocamlopt -i -p  -c f1.ml
val interval : int -> int -> int list
exception Found_zero
val mult_list : int list -> int
$ ocamlopt -i -p -o f2nat.exe f1.cmx f2.ml
```

ここではサブオプションのない -p オプションを使っています。f2nat.exe の実行が終了すれば gmon.out というファイルを生成します。このファイルは Unix の標準的なツール (286 ページ参照) で使えるフォーマットになっています。

結果の出力

バイトコードと機械語コードで生成されるファイルが異なっているので、結果を出力させる方法も異なっています。バイトコードの場合は制御構造を通過した回数がプログラム中にコメントとして挿入されます。一方、機械語コードの場合は関数ごとの呼び出された回数と実行時間のグラフが出力されます。

バイトコードコンパイラの場合

結果を見るためには ocamlprof コマンドを使います。このコマンドはプログラムのソースファイルを引数として取り、camlprof.dump ファイルの内容を調べ、その情報をコメントとして付加したソースファイルを出力します。

整数のリストの乗算の例では以下のような結果になります。

```
ocamlprof f1.ml

let rec interval a b =
  (* 62 *) if b < a then (* 2 *) []
  else (* 60 *) a::(interval (a+1) b);;

exception Found_zero ;;

let mult_list l =
 (* 2 *) let rec mult_rec l = (* 62 *) match l with
    [] -> (* 1 *) 1
  | 0::_ -> (* 1 *) raise Found_zero
  | n::x -> (* 60 *) n * (mult_rec x)
 in
  try mult_rec l with Found_zero -> (* 1 *) 0
;;
```

解析結果には F2 モジュールの結果も反映されています。mult_list が呼び出されたのは 2 回であり、補助関数 mult_rec は 62 回呼び出されています。パターンマッチの部分では一般の場合が 60 回、[] パターンは 1 度だけ実行されています。リストの先頭が 0 であったのは正確に 1 度だけであり、これは **try** 文の catch 節が 1 度実行されていることと一致しています。

ocamlprof コマンドには 2 つのオプションがあります。-f オプションではプロファイリングの情報を含んだファイルを指定します。-F オプションは挿入する数字の前に指定された文字列を追加するように指示します。

機械語コードコンパイラの場合

リストの要素を掛け合わせるプログラムで、それぞれの関数の実行時間を知る方法について考えます。まず以下のプログラムを f3.ml というファイルに書き込みます。

```
let l1 = F1.interval 1 30;;
let l2 = F1.interval 31 60;;
let l3 = l1 @ (0 :: l2);;

for i=0 to 100000 do
  F1.mult_list l1;
  F1.mult_list l3
done;;

print_int (F1.mult_list l1);;
print_newline();;

print_int (F1.mult_list l3);;
print_newline();;
```

これはループを 100000 回繰り返すことを除いて f2.ml と同じです。

プログラムを実行すると gmon.out というファイルが生成されます。このファイルは gprof という Unix のツールによって読むことができます。gprof を使うと以下のように個々の関数の実行時間に関する情報が出力されます。この出力は非常に長くなるので、ここでは関数の実行時間についての情報を含んでいる最初のページだけを示します。

```
$ gprof  f3nat.exe
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 92.31      0.36     0.36   200004     1.80     1.80  F1_mult_rec_45
  7.69      0.39     0.03   200004     0.15     1.95  F1_mult_list_43
  0.00      0.39     0.00     2690     0.00     0.00  oldify
  0.00      0.39     0.00      302     0.00     0.00  darken
  0.00      0.39     0.00      188     0.00     0.00  gc_message
  0.00      0.39     0.00      174     0.00     0.00  aligned_malloc
  0.00      0.39     0.00      173     0.00     0.00  alloc_shr
  0.00      0.39     0.00      173     0.00     0.00  fl_allocate
  0.00      0.39     0.00       34     0.00     0.00  caml_alloc3
  0.00      0.39     0.00       30     0.00     0.00  caml_call_gc
  0.00      0.39     0.00       30     0.00     0.00  garbage_collection
...
```

このプログラムでは実行時間のほとんどが F1_mult_rec_45 という関数の実行によって占められています。この関数は f1.ml の中で定義された F1.mult_rec という関数に対応しています。リストには他にも呼び出された関数を見ることができますが、このほとんどはメモリ管理のためのランタイムライブラリに含まれているものです (第 9 章参照)。

練習問題

関数適用のトレース

この練習問題では関数適用の時点での引数の評価状況について調べてみましょう。

1. 関数 List.fold_left に以下の引数を与えて評価した時のトレース結果を調べなさい。

   List.fold_left (-) 1 [2; 3; 4; 5];;

   どのようなトレースが得られましたか。

2. 関数 List.fold_left と同じ働きをする関数 fold_left_int を定義しなさい。ただしこの関数の型は (*int* → *int* → *int*) → *int* → *int list* → *int* とします。この関数のトレースを取り、 なぜ元の関数とトレースが変わるのか答えなさい。

性能解析

第 9 章 (249 ページ) の練習問題の続きを行います。その練習問題では素数を計算する二つのプログラム (末尾再帰を使ったものと使わなかったもの) のメモリの仕様状況を比較しましたが、ここではプロファイリングツールを使って関数の実行時間の比較を行います。この練習問題ではインライン展開 (第 7 章参照) の重要性について学びます。

1. erart と eranrt の二つのプログラムをプロファイリングオプション付きでコンパイルしなさい。 コンパイルするにはバイトコードコンパイラと機械語コードコンパイラの両方で行いなさい。
2. 引数に 3000 4000 5000 6000 を指定してプログラムを実行しなさい。
3. 結果を ocamlprof と gprof 使って表示させ、 分析しなさい。

まとめ

この章では Objective Caml パッケージに含まれているプログラミング支援ツールを紹介しました。

最初のツールはファイル同士の依存関係を解析します。この情報は Makefile に供給され、分割コンパイルを容易に実現します。

他のツールはプログラムの実行に関する情報を提供します。インタープリタはプログラムの実行のトレースを提供しますが本文中で既に示されているように、値を表示する時

に多相性が障害となります。しかし単相的な引数や再帰的な関数のトレースについては全く問題ありません。

残りはデバッガとプロファイラで Unix 環境で伝統的に使われているツールです。デバッガではプログラムをステップ実行させることができます。プロファイラは性能に関する情報を得ることができます。どちらも Unix 環境でのみ利用できます。

もっと知りたい人へ

`ocamldep` コマンドの出力結果は `ocamldot` でグラフィカルに表示させることができます。`ocamldot` についての詳細は以下のページを御覧ください。

リンク: http://www.cis.upenn.edu/~tjim/ocamldot/index.html

`ocamldot` は別のツールである `dot` を利用しています。このツールは以下のページからダウンロードできます。

リンク: http://www.research.att.com/sw/tools/graphviz/

Objective Caml でソフトウェア開発を行うときのプロジェクト管理について `Makefile` のテンプレートが用意されています。

リンク: http://caml.inria.fr/FAQ/Makefile_ocaml-eng.html

リンク: http://www.ai.univie.ac.at/~markus/ocaml_sources

これらは `ocamldep` の出力を利用しています。

文献 [HF+96] では 20 種類の関数型言語 (ML を含む) の性能評価を行っています。ベンチマークは巨大なデータ構造の数値計算です。

11

字句解析と構文解析のための ツール

字句解析と構文解析のためのツールの整備は情報科学の重要な研究分野であり続けています。この活動の一つとして字句・構文解析器の自動生成を行う lex と yacc が生み出されました。この章では、それらのツールに由来する camllex と camlyacc について解説を行います。これら二つのツールは字句解析器と構文解析器を実装する事実上標準的なやり方ですが、他にも行単位の処理などを行う**正規表現**ライブラリ str があります。もっともこれは強力な解析を必要としないで済むアプリケーションに向いています。

このようなツールは現代的なプログラミング言語の分野でとりわけ重宝されていますが、他のアプリケーションにも適用することができます。たとえばデータベースシステムでは問い合わせを構文解析する場合がありますし、スプレッドシートでは数式を評価した結果を利用してセルの内容を定義することがあります。そこまでいかなくてもシステムコンフィグレーションファイルのように、プレーンテクストにデータを保存しておくことは良く行われています。このように限定された場合ではありますが、字句・構文解析を必要とする処理を行っています。

これらすべての例において字句・構文解析が解決しなければならないのは単純な文字の列を、語やレコードまたはプログラムの抽象構文木などの豊かな構造を持ったデータへと変換する問題です。

どんな言語でも語彙（字句の集合）と字句をどのように組み合わせ方を記述した文法を持っています。コンピュータのプログラムが言語を正しく処理するためには正確な字句・構文規則に従わなければなりません。コンピュータは自然言語の曖昧性を解消できる細やかな意味上の理解ができません。この限界があるためにコンピュータ言語は一般に、例外なく明確に規定された規則に従う必要があります。そのため、このような言語の字句・文法の持つ構造は形式的な手法によって定義されています。この章では形式的な定義法について簡単に述べ、その後でその使い方について説明を行います。

この章のあらまし

この章では Objective Caml の配付パッケージに含まれている字句・構文解析のためのツールについて紹介します。構文解析用のツールは通常、字句解析用のツールと一緒に使うことを想定しています。第一節では Genlex モジュールによって提供されている、字句解析のためのツールについて紹介します。Next we give details about the definition of sets of lexical units by introducing the formalism of **regular expressions**. We illustrate their behavior within module Str and the ocamllex tool. In section two we define grammars and give details about sentence production rules for a language to introduce two types of parsing: bottom-up and top-down. They are further illustrated by using *Stream* and the ocamlyacc tool. These examples use context-free grammars. We then show how to carry out contextual analysis with *Streams*. In the third section we go back to the example of a BASIC interpreter from page 159, using ocamllex and ocamlyacc to implement the lexical analysis and parsing functions.

語彙

字句解析は文字列処理の第一歩です。字句解析は文字の列を分割して語 **lexemes** の列へと変換します。

Genlex モジュール

このモジュールは、字句解析を行う基本的な機能を提供しています。文字の列を解析するためのいくつかの語彙があらかじめ定義されています。それには次のようなものがあります。

```
# type token =
     Kwd of string
   | Ident of string
   | Int of int
   | Float of float
   | String of string
   | Char of char ;;
```

したがってこのモジュールを使うと文字の列の中から整数（Int 構築子）を認識することができ、その値を取り出すことができます（Int 構築子の引数）。また慣習的な記法である（"）に囲まれた文字列や（'）に囲まれた文字を取り出すこともできます。浮動小数点数は 0.01 のような通常の記法でも、1E-2 のような指数仮数表現でも認識できます。

Ident 構築子は**識別子**のカテゴリーを示しています。識別子とは例えばプログラミング言語の変数名や関数名のことです。識別子は文字や数字、下線（_）、アクセント記号（'）などを含んでいます。識別子は数字で始まってはいけません。このモジュールでは演算子（+、*、>、=など）もまた識別子であると考えています。最後に Kwd 構築子は特別な識別子や文字を含んでいるキーワードのカテゴリーを示しています。

引数によって制御できるのはキーワードだけです。次の関数はキーワードのリストを受け取って字句解析器（lexer）を生成します。

```
# Genlex.make_lexer ;;
- : string list -> char Stream.t -> Genlex.token Stream.t = <fun>
```

make_lexer にキーワードのリストを適用した結果は、文字列を受け取り、語（*token* 型）の列を返す関数になります。

したがってこれを利用すると BASIC インタープリタを簡単に作ることができます。まずキーワードを宣言します。

```
# let keywords =
    [ "REM"; "GOTO"; "LET"; "PRINT"; "INPUT"; "IF"; "THEN";
      "-"; "!"; "+"; "-"; "*"; "/"; "%";
      "="; "<"; ">"; "<="; ">="; "<>";
      "&"; "|" ] ;;
```

この定義をした後に字句解析器を次のように定義します。

```
# let line_lexer l = Genlex.make_lexer keywords (Stream.of_string l) ;;
val line_lexer : string -> Genlex.token Stream.t = <fun>
# line_lexer "LET x = x + y * 3" ;;
- : Genlex.token Stream.t = <abstr>
```

line_lexer 関数は文字の流れを受け取り、語 lexeme の列を返します。

文字の流れの制御

文字の流れを直接扱って字句解析を行うこともできます。

以下の例では数式の字句解析器を扱います。関数 lexer は文字の流れを受け取り、*lexeme Stream.t* 型の語の列を返します[1]。空白文字、タブ、改行コードは除去されます。問題を簡単にするために変数と負の数は扱わないことにします。

```
# let rec spaces s =
    match s with parser
      [<'' ' ; rest >] → spaces rest
    | [<''\t' ; rest >] → spaces rest
    | [<''\n' ; rest >] → spaces rest
```

1. *lexeme* 型は 163 ページで定義されます。

```
    | [<>] → ();;
Characters 46-48:
      [<'' '  ; rest  >] -> spaces rest
      ^^
Syntax error
# let rec lexer s =
    spaces s;
    match s with parser
      [< ''('  >] → [< 'Lsymbol "(" ; lexer s >]
    | [< '')'  >] → [< 'Lsymbol ")" ; lexer s >]
    | [< ''+'  >] → [< 'Lsymbol "+" ; lexer s >]
    | [< ''-'  >] → [< 'Lsymbol "-" ; lexer s >]
    | [< ''*'  >] → [< 'Lsymbol "*" ; lexer s >]
    | [< ''/'  >] → [< 'Lsymbol "/" ; lexer s >]
    | [< ''0'..'9' as c;
         i,v = lexint (Char.code c - Char.code('0')) >]
        → [<'Lint i ; lexer v>]
  and lexint r s =
    match s with parser
      [< ''0'..'9' as c >]
      → let u = (Char.code c) - (Char.code '0') in  lexint (10*r + u) s
    | [<>] → r,s
  ;;
Characters 56-58:
      [< ''('  >] -> [< 'Lsymbol "(" ; lexer s >]
      ^^
Syntax error
```

関数 lexint は文字の流れの中にある整数定数に対する字句解析を行います。この関数は lexer 関数が入力の中に数字を見付けたときに呼び出されます。lexint 関数はそのときに数字の列を消費して対応する整数値を求めます。

正規表現

2

字句解析での基本的な単位についての問題をちょっとばかり抽象化して理論的な観点から考えてみましょう。

我々の観点では字句解析の基本的な単位は「語」になります。語とは文字の集合「アルファベット」の中の特定の文字を並べて結合させたものです。ここではアルファベットとしてアスキー文字集合の部分集合を考えています。理論的には 0 文字からなる「空語[3]」

2. アカデミックの世界の用語では「正則」表現とするのが正しいのであるが、我々はプログラマの伝統に従い「正規」表現と呼ぶことにする。
3. 慣習にしたがって空語はギリシャ文字 ϵ によって表現されます。

を考えることもできます。アルファベットの要素から字句解析の基本的な単位（lexeme）を構成する方法について理論的な研究が長い間行われ、その成果の一つとして「正規表現」として知られている単純明解な形式が考案されました。

定義　正規表現は語の集合を定義します。例えば、正しい識別子の集合を定義することができます。正規表現はいくつかの集合理論的操作によって定義されます。M と N を語の集合とすると、

1. M と N の和集合を M | N と書きます。
2. M の補集合を ^M と書きます。補集合とは M に含まれないすべての語の集合です。
3. M と N の連接を MN と書きます。M と N の連接とは M に含まれる語と N に含まれる語を並べて結合したすべての語の集合です。
4. M に含まれる語を有限個並べて出来るすべての語の集合を M+ と書きます。
5. 記述上の慣習として語の集合 M に空語を加えた集合を M? と書きます。

一つ一つの文字はその一文字だけを要素として含む集合を表します。そのため a | b | c という式は a と b と c の三語を含む集合を表します。特にこのような集合を定義する場合は [abc] というより簡潔な表現も使います。我々の考えているアルファベットには順序があります（アスキーコードによって指定された順序）ので文字を範囲で指定することができます。例えば数字の集合を [0-9] と書きます。また演算子の優先順位を変更するために括弧を使います。

もし演算子のために使われている文字を文字として使いたい場合にはエスケープ文字 \ をその使いたい文字の直前に置きます。例えば (*)* はアスタリスク文字の繰り返しを表しています。

例　アルファベットとして数字（0, 1, 2, 3, 4, 5, 6, 7, 8, 9）、プラス記号（+）、マイナス記号（-）、ドット記号（.）と英文字 E を考えます。これを利用して数を表す語の集合 *num* を定義します。まず整数を表す集合 *integers* を [0-9]+ と定義しましょう。符号のない数を表す集合 *unum* を次のように定義します。

integers?(.*integers*)?(E(\+|-)?*integers*)?

符号付き数はしたがって次のような定義になります。

unum | -*unum* or with $-?unum$

認識　正規表現はそれ自体とても役に立つ形式ですが、実際にはある文字列が、正規表現によって指定された語の集合の要素であるかどうか判定するプログラムを実装したい場合があります。そのためには集合の形式的な定義から、表現を処理して認識を行うプログラムを生成する必要があります。正規表現の場合は、この変換を自動的に行うことができます。このような技術は以下に続く節で紹介を行う ocamllex ツールと、Str ライブラリの中の Genlex モジュールが提供しています。

Str ライブラリ

このモジュールには正規表現を表す抽象データ型 *regexp* と、おおよそ上で説明した構文に従う正規表現を表す文字列を受け取り、その抽象的な表現を返す関数 regexp が含まれています。

このモジュールはまた正規表現を利用して文字列を操作する多くの関数を含んでいます。Str ライブラリでの正規表現の構文は図 11.1 のようになっています。

.	\n 以外の任意の一文字
*	直前の表現の零回以上の繰り返し
+	直前の表現の一回以上の繰り返し
?	直前の表現であるかまたは空語
[..]	文字の集合（例 [abc]）
	区間（例 [0-9]）
	補集合（例 [^A-Z]）
^	行頭（補集合を表す ^ と間違えないように）
$	行末
|	選択
(..)	式のグループ化（以下で説明しますが、グループ化した式は番号で参照できます。）
i	整数定数。i 番目のグループとマッチした文字列を指しています。
\	エスケープ文字（正規表現のために予約されている文字をマッチングで使いたい場合）

図 11.1: 正規表現

例 なんらかのテキストファイルに含まれている英米流の日付表示をフランス流に変換する関数を書いてみましょう。そのテキストファイルは行ごとに何らかのデータが書かれており、英米流の日付表示ではドットによって項目が区切られていると仮定します。ファイルから読まれた一行を表す文字列を引数として受け取り、日付を分離してフランス流の表示に変換する関数を定義してみましょう。

```
# let french_date_of d =
   match d with
    [mm; dd; yy] → dd^"/"^mm^"/"^yy
   | _ → failwith "Bad date format" ;;
val french_date_of : string list -> string = <fun>

# let english_date_format = Str.regexp "[0-9]+\.[0-9]+\.[0-9]+" ;;
Characters 45-47:
Warning: Illegal backslash escape in string
Characters 53-55:
Warning: Illegal backslash escape in string
```

```
  let english_date_format = Str.regexp "[0-9]+\.[0-9]+\.[0-9]+" ;;
                                                    ^^
  let english_date_format = Str.regexp "[0-9]+\.[0-9]+\.[0-9]+" ;;
                                                            ^^
val english_date_format : Str.regexp = <abstr>

# let trans_date l =
    try
     let i=Str.search_forward english_date_format l 0 in
     let d1 = Str.matched_string l in
     let d2 = french_date_of (Str.split (Str.regexp "\.") d1) in
       Str.global_replace english_date_format d2 l
    with Not_found → l ;;
Characters 165-167:
Warning: Illegal backslash escape in string
    let d2 = french_date_of (Str.split (Str.regexp "\.") d1) in
                                                        ^^
    val trans_date : string -> string = <fun>

# trans_date "..............06.13.99............" ;;
- : string = "..............13/06/99............"
```

ocamllex ツール

ocamllex ツールとは、C 言語のための字句解析器生成器である lex ツールをモデルとして Objective Caml のために作られたものです。このツールは正規表現として認識される字句要素を記述したファイルを入力として受け取り、その字句要素を解析する Objective Caml のソースプログラムを出力として生成します。プログラマは字句要素の記述に**意味動作 semantic action** を付け加えることができます。生成されたプログラムは Lexing モジュールで定義されている *lexbuf* 抽象型を利用しています。プログラマはこのモジュールを使って、バッファの処理を制御することもできます。

字句解析の記述ファイルの拡張子は通常 .mll となっています。lex_file.mll という記述ファイルから Objective Caml のプログラムを生成するには次のように打ち込みます。

```
ocamllex lex_file.mll
```

この結果、lex_file.ml というファイルが生成されます。このファイルは、記述ファイルに指定されている字句解析を行うプログラムを含んでいます。この後に Objective Caml アプリケーションの他のモジュールと一緒にこのファイルをコンパイルすることができます。字句解析ルールごとに同じ名前を持った関数が定義されています。その関数は字句解析のバッファ（*Lexing.lexbuf* 型）を引数として受け取り、意味動作で定義された値を返します。したがって、同じルールの意味動作はすべて同じ型の値を返さなければなりません。

ocamllex の記述ファイルは一般に次のようなフォーマットになっています。

```
{
  ヘッダー
}

let ident = regexp
  ...
rule ruleset1 = parse
      regexp { 意味動作 }
     | ...
     | regexp { 意味動作 }
and ruleset2 = parse
     ...
and ...
{

  後処理
}
```

ヘッダーと後処理の部分はなくてもかまいません。それらの部分では字句解析の時に必要になる Objective Caml の型、関数などを定義するために使います。後処理の部分では、中間部分から生成される字句解析のための関数を使うことができます。ルール定義の直前にある宣言リストは正規表現に名前を付けるのに使います。その名前はルールの中で参照できます。

例 BASIC の題材をまた考えてみましょう。今回は、返される語の型を改良しましょう。前回（163 ページ）と同様に *lexeme* 型の値を返す `lexer` 関数を定義します。ただし、今回は *Lexing.lexbuf* 型のバッファを引数に取ります。

```
{
 let string_chars s =
  String.sub s 1 ((String.length s)-2) ;;
}

let op_ar = ['-' '+' '*' '%' '/']
let op_bool = ['!' '&' '|']
let rel = ['=' '<' '>']

rule lexer = parse
   [' ']   { lexer lexbuf }

 | op_ar   { Lsymbol (Lexing.lexeme lexbuf) }
 | op_bool { Lsymbol (Lexing.lexeme lexbuf) }

 | "<="    { Lsymbol (Lexing.lexeme lexbuf) }
 | ">="    { Lsymbol (Lexing.lexeme lexbuf) }
 | "<>"    { Lsymbol (Lexing.lexeme lexbuf) }
```

```
| rel     { Lsymbol (Lexing.lexeme lexbuf) }

| "REM"   { Lsymbol (Lexing.lexeme lexbuf) }
| "LET"   { Lsymbol (Lexing.lexeme lexbuf) }
| "PRINT" { Lsymbol (Lexing.lexeme lexbuf) }
| "INPUT" { Lsymbol (Lexing.lexeme lexbuf) }
| "IF"    { Lsymbol (Lexing.lexeme lexbuf) }
| "THEN"  { Lsymbol (Lexing.lexeme lexbuf) }

| '-'? ['0'-'9']+   { Lint (int_of_string (Lexing.lexeme lexbuf)) }
| ['A'-'z']+        { Lident (Lexing.lexeme lexbuf) }
| '"' [^ '"']* '"'  { Lstring (string_chars (Lexing.lexeme lexbuf)) }
```

ocamllex によってこのファイルを変換すると *Lexing.lexbuf -> lexeme* 型を持つ lexer 関数が生成されます。このあと構文解析について考える時（308 ページ）にこの lexer 関数の使い方について説明します。

構文

字句解析を利用すれば入力流の構造を切り分けるのが簡単にできます。そうやって得られた語を組み立てて、想定されている言語の文法的に正しい文を形成するのが次の問題です。それを行うのが構文解析の役割です。文を組み立てる規則は**文法規則**によって定義されます。この形式化手法はもともとは言語学の分野で発展してきたものですが、やがて数理言語学者とコンピュータ科学者にとっても極めて役に立つことが分かって来ています。文法規則については既に 160 ページで Basic 言語についての例を見てきましたが、ここではその例をさらに進めながら文法についての基本的な概念を勉強することにしましょう。

文法

形式的には文法とは次の四要素によってできています。

1. **終端記号**と呼ばれるシンボルの集合。終端記号とは対象としている言語の字句要素（語）を表しています。Basic 言語の場合、字句要素とはマイナス演算子-、算術的、論理的記号（+、&、<、<=等）、キーワード（GOTO、PRINT、IF、THEN）、整数、変数になります。
2. **非終端記号**と呼ばれるシンボルの集合。非終端記号とは対象としている言語の構文要素を表しています。例えば Basic のプログラムは行（Line）から成り立っており、行は式（Expression）を含んでいます。
3. **生成規則**の集合。生成規則とは終端記号と非終端記号からどのように構文要素を生成するかを示しているものです。Basic の行は行番号に続いて命令が書かれていますが、これは次のような規則によって表現されます。

$$\text{Line} \quad ::= \quad \textit{integer}\ \text{Instruction}$$

同じ終端記号は異なった記号列から生成されることがありえます。その時は個々のケースを区別するために — という記号を使います。例えば

INSTRUCTION	::=	LET *variable* = EXPRESSION
	—	GOTO *integer*
	—	PRINT EXPRESSION

のようになります。

4. **開始記号**と呼ばれる特別な非終端記号。開始記号は対象としている言語の正しい文全体（プログラム）を指定します。開始記号の生成規則が構文解析のスタート地点となります。

生成と認識

生成規則は、言語の字句要素の列をどう認識するかという問題と深く関わっています。

ここでは、数式のための単純な言語を考えてみましょう。

EXP	::=	*integer*	(R1)
	—	EXP + EXP	(R2)
	—	EXP * EXP	(R3)
	—	(EXP)	(R4)

ただし右辺にある (R1)、(R2)、(R3)、(R4) はそれぞれの規則の名前を表しています。1*(2+3) という式を字句解析すると、次のような語の列が得られます。

integer * (*integer* + *integer*)

この文を解析して、数式言語に間違いなく属していることを確認するためには、数式の生成規則を右から左に見ていきます。もしある部分式が生成規則のどれかの右辺とマッチすれば、その部分式を対応する右辺と置き換えます。この書き換えを繰り返し適用して、最終的に式を非終端記号 *start* （ここでは EXP）まで還元します。ここに還元の例を示します。[4]

$$
\begin{array}{lcl}
\underline{integer} * (\ integer + integer\ ) & \stackrel{(R1)}{\longleftarrow} & \text{EXP} * (\ \underline{integer} + integer\ ) \\
& \stackrel{(R1)}{\longleftarrow} & \text{EXP} * (\ \text{EXP} + \underline{integer}\ ) \\
& \stackrel{(R1)}{\longleftarrow} & \text{EXP} * (\ \underline{\text{EXP} + \text{EXP}}\ ) \\
& \stackrel{(R2)}{\longleftarrow} & \text{EXP} * \underline{(\ \text{EXP}\ )} \\
& \stackrel{(R4)}{\longleftarrow} & \underline{\text{EXP} * \text{EXP}} \\
& \stackrel{(R3)}{\longleftarrow} & \text{EXP}
\end{array}
$$

4. 還元する部分式には下線が引いてあります。またその時に使った規則名も書いてあります。

Exp だけを含む最終行から始めて、矢印に従って下から上に読んでいけば、開始記号 Exp からどのように最初の式が生成されたのか理解することができます。つまり、最初の式はこの文法によって定義された言語に含まれていることが分かります。

文法から、その文法によって定義された言語に属する文を認識するプログラムを生成するのは、正規言語の場合よりもかなり複雑な問題になります。過去の研究の結果から分かっているのは正規言語の形式によって定義された言語は**決定性有限オートマトン**と本質的に等価であるということです。そしてその決定性有限オートマトンとして動作するプログラムを書くのはかなり簡単です。しかし、一般の文法に対してはそのような結果は得られませんでした。例えばある文法のクラスに対しては決定性有限オートマトンよりも複雑な**プッシュダウンオートマトン**と等価であることが分かっています。ただしここではこの話題にこれ以上関わりません。オートマトンの厳密な定義も行いません。ここでは単に、我々の構文解析器生成器で使われている文法について明確に説明を行います。

トップダウン解析

前段で登場した `1*(2+3)` という式の構文解析は一通りだけではありません。*integers* の還元を右から左に行うこともできました。その場合さらに `2+3` を規則（R2）で先に還元することもできたでしょう。これらの二通りの方式はトップダウン解析（右から左）とボトムアップ解析（左から右）と呼ばれているものと対応しています。トップダウン解析は `Stream` モジュールを使えば簡単に実装できます。一方ボトムアップ解析は `ocamlyacc` ツールによって実装されています。ボトムアップ解析は、Basic の構文解析の時に既に解説したようにスタックを利用しています。構文解析にどちらの方式を選択するかという判断は重要です。対象となる言語を定義している文法によってはトップダウン解析が不可能な場合もあります。

簡単な場合

トップダウン解析の最も標準的な例とは次のように定義される数式の前置記法です。

$$
\begin{array}{rcl}
\textsc{Expr} & ::= & \textit{integer} \\
 & — & + \ \textsc{Expr}\ \textsc{Expr} \\
 & — & * \ \textsc{Expr}\ \textsc{Expr}
\end{array}
$$

この場合、先頭の語さえ分かればどの生成規則を使うのか決まります。したがってトップダウン解析を行うプログラムは単純な再帰呼び出しを行う関数によって実装することができます。`Genlex` モジュールと `Stream` モジュールを使って実装した例を以下に示します。関数 `infix_of` は前置記法の式を受け取り、意味が等価な中置記法の式を返します。

```
# let lexer s =
    let ll = Genlex.make_lexer ["+";"*"]
    in  ll (Stream.of_string s) ;;
val lexer : string -> Genlex.token Stream.t = <fun>
# let rec stream_parse s =
    match s with parser
```

```
    [<'Genlex.Ident x>] → x
  | [<'Genlex.Int n>] → string_of_int n
  | [<'Genlex.Kwd "+"; e1=stream_parse; e2=stream_parse>] → "("^e1^"+"^e2^")"
  | [<'Genlex.Kwd "*"; e1=stream_parse; e2=stream_parse>] → "("^e1^"*"^e2^")"
  | [<>] → failwith "Parse error"
 ;;
Characters 52-54:
      [<'Genlex.Ident x>] -> x
      ^^
Syntax error
# let infix_of s = stream_parse (lexer s) ;;
Characters 17-29:
  let infix_of s = stream_parse (lexer s) ;;
                   ^^^^^^^^^^^^
Unbound value stream_parse
# infix_of "* +3 11 22";;
Characters 0-8:
  infix_of "* +3 11 22";;
  ^^^^^^^^
Unbound value infix_of
```

ただし、この構文解析器はかなり融通のないものになっています。語の間に空白がないと正しく解釈ができません。

```
# infix_of "*+3 11 22";;
Characters 1-9:
  infix_of "*+3 11 22";;
  ^^^^^^^^
Unbound value infix_of
```

やや単純でない例

ストリームを使った構文解析の動作はかなり予測しやすいものになっています。このような構文解析を行うためには文法は二つの条件を満たしていなければなりません。

1. 文法は**左再帰**規則を含んでいてはなりません。左再帰規則とは右辺の式が左辺と同じ非終端記号から始まっている規則です。例えば Exp ::= Exp + Exp このような規則です。
2. 二つの規則が同じ式から始まってはなりません。

298 ページの通常の数式の文法に対してはトップダウン解析を直接適用することはできません。この文法はトップダウン解析の制約を満たしていません。したがってトップダウン解析を行うためには文法を書き換えて左再帰と非決定性を除去する必要があります。例えば次のような文法になります。

EXPR	::=	ATOM NEXTEXPR
NEXTEXPR	::=	+ ATOM
	—	- ATOM
	—	* ATOM
	—	/ ATOM
	—	ϵ
ATOM	::=	*integer*
	—	(EXPR)

NEXTEXPR の定義に空語 ϵ がないと整数だけの式を解釈できないことに注意してください。

この文法は、生成規則を直接実装した構文解析器を実装できるようになっています。以下の実装では数式を表す抽象構文木を生成します。

```
# let rec rest = parser
      [< 'Lsymbol "+"; e2 = atom >]  → Some (PLUS,e2)
    | [< 'Lsymbol "-"; e2 = atom >]  → Some (MINUS,e2)
    | [< 'Lsymbol "*"; e2 = atom >]  → Some (MULT,e2)
    | [< 'Lsymbol "/"; e2 = atom >]  → Some (DIV,e2)
    | [< >] → None
  and atom = parser
      [< 'Lint i >] → ExpInt i
    | [< 'Lsymbol "("; e = expr ; 'Lsymbol ")" >] → e
  and expr s =
    match s with parser
      [< e1 = atom >] →
        match rest s with
            None → e1
          | Some (op,e2) → ExpBin(e1,op,e2) ;;
Characters 28-30:
      [< 'Lsymbol "+"; e2 = atom >]  -> Some (PLUS,e2)
      ^^
Syntax error
```

トップダウン解析の主要な問題とは、極めて制限された形式の文法を使わなければならないということです。もし対象としている言語が左再帰文法を使って自然に書ける（例えば中置記法の式）場合はその文法と等価（すなわち同じ言語を定義している）であって、トップダウン解析の制約を満たしている文法を見付けるのは必ずしも自明ではありません。このため yacc と ocamlyacc はボトムアップ解析を採用しています。ボトムアップ解析ではより人間が自然に見える文法を定義することができます。ただし、ボトムアップ解析であってもやはり限界はあります。

ボトムアップ解析

本書の 165 ページから始まる部分でボトムアップ解析の動作に関する直観的な解説を行いました。ボトムアップ解析は、スタックの状態を書き換える**シフト**と**還元**という二つの動作を基本としています。文法がボトムアップ解析の制約を満たしている限り、シフトと還元の動作列から文の認識することができます。ここでもやはり問題は文法が持つ非決定性です。この節では広く使われている後置記法と中置記法の数式を例に取り、ボトムアップ解析の内部で行われている動作と、その失敗の原因について解説します。

ボトムアップ解析の利点 The simplified grammar for postfix arithmetic expressions is:

$$\begin{array}{lcll} \textsc{Expr} & ::= & \textit{integer} & \text{(R1)} \\ & | & \textsc{Expr}\ \textsc{Expr}\ \texttt{+} & \text{(R2)} \\ & | & \textsc{Expr}\ \textsc{Expr}\ \texttt{*} & \text{(R3)} \end{array}$$

この文法は前置記法の数式の文法のちょうど逆になっています。解析の最後になるまでどの規則を適用すべきだったのか分かりませんが、その時に適用できる規則は一つだけしかありません。実際にこのような式のボトムアップ解析はスタックを利用した数式の計算過程ととてもよく似ています。計算結果をスタックに積む代わりに文法記号を積むことになります。ボトムアップ解析は空の状態のスタックから始まり、入力がすべて解析されると開始記号だけを含む状態で終了します。シフトする時は現在の非終端記号をスタックに積みます。還元は、ある規則の右辺の要素が逆順でスタックの先頭に積まれている時に起こります。この場合、それらの要素を対応する左辺の非終端記号で置き換えます。

図 11.2 は `1 2 + 3 * 4 +` という式のボトムアップ解析が進行する過程を説明しています。入力となる語には下線が引かれています。入力列の最後を `$` 記号で表しています。

ボトムアップ解析の限界 文法から言語を認識するプログラムを作り出す際の困難は適用する動作を決定する点に尽きています。この問題について三種類の非決定性を持つ例を使って説明します。

最初の例は足し算だけを含む式の文法です。

$$\begin{array}{lcll} \text{E0} & ::= & \textit{integer} & \text{(R1)} \\ & | & \text{E0}\ \texttt{+}\ \text{E0} & \text{(R2)} \end{array}$$

この文法の非決定性は規則（R2）に起因しています。次のような状況を考えてみましょう。

動作	入力	スタック
⋮		
	<u>+</u>. . .	[E0 + E0 . . .]
⋮		

動作	入力	スタック
	12+3*4+$	[]
シフト		
	2+3*4+$	[1]
還元（R1）		
	2+3*4+$	[Expr]
シフト		
	+3*4+$	[2Expr]
還元（R1）		
	+3*4+$	[Expr Expr]
シフト、還元（R2）		
	3*4+$	[Expr]
シフト、還元（R1）		
	*4+$	[Expr Expr]
シフト、還元（R3）		
	4+$	[Expr]
シフト、還元（R1）		
	+$	[Expr Expr]
シフト、還元（R2）		
	$	[Expr]

図 11.2: ボトムアップ解析の例

このような場合シフト動作を行って + 記号をスタックに積むべきなのか、規則（R2）を適用してスタックの中の E0 + E0 を還元するべきなのか、決定できません。これをシフトと還元の衝突 shift-reduce conflict と呼びます。この例の場合は *integer* + *integer* + *integer* という式が二通りの導出ができることに原因があります。

$$
\begin{array}{lll}
\text{導出 1} & \text{E0} & \xrightarrow{(R2)} \text{E0 + \underline{E0}} \\
 & & \xrightarrow{(R1)} \underline{\text{E0}} + integer \\
 & & \xrightarrow{(R2)} \text{E0 + \underline{E0}} + integer \\
 & \text{etc.} &
\end{array}
$$

$$
\begin{array}{lll}
\text{導出 2} & \text{E0} & \xrightarrow{(R2)} \text{E0 + \underline{E0}} \\
 & & \xrightarrow{(R2)} \text{E0 + E0 + \underline{E0}} \\
 & & \xrightarrow{(R1)} \text{E0 + \underline{E0}} + integer \\
 & \text{etc.} &
\end{array}
$$

それぞれの導出によって得られる式は式の値の計算という観点では同じように見えるかもしれません。

$$(integer + integer) + integer \text{ と } integer + (integer + integer)$$

しかし構文木の上では両者は異なっています（166 ページの図 6.3 を参照）。

シフトと還元の衝突が起こる文法の二番目の例は、式に暗黙に付けられている括弧に関係するという意味では一番目の例と同じですが、括弧の付け方によって式の意味が変わってしまうような場合です。次のような文法を考えてみましょう。

$$\begin{array}{lcll} \mathrm{E1} & ::= & integer & (\mathrm{R1}) \\ & | & \mathrm{E1} + \mathrm{E1} & (\mathrm{R2}) \\ & | & \mathrm{E1} * \mathrm{E1} & (\mathrm{R3}) \end{array}$$

この文法では一番目の例で述べた衝突が + 記号と * 記号に関して起こることは分かりますが、それに加えて + 記号と * 記号の間でも衝突が発生します。異なった導出方法によって同じ式を生成することが可能です。次の式を見てください。

$$integer + integer * integer$$

$$\begin{array}{lll} \text{導出 1} & \mathrm{E1} & \xrightarrow{(R3)} \mathrm{E1} * \underline{\mathrm{E1}} \\ & & \xrightarrow{(R1)} \underline{\mathrm{E1}} * integer \\ & & \xrightarrow{(R2)} \mathrm{E1} + \underline{\mathrm{E1}} * integer \end{array}$$

以下略

$$\begin{array}{lll} \text{導出 2} & \mathrm{E1} & \xrightarrow{(R2)} \mathrm{E1} + \underline{\mathrm{E1}} \\ & & \xrightarrow{(R3)} \mathrm{E1} + \mathrm{E1} * \underline{\mathrm{E1}} \\ & & \xrightarrow{(R1)} \mathrm{E1} + \underline{\mathrm{E1}} * integer \end{array}$$

以下略

この場合（暗黙の）括弧によって式の意味が変わってしまいます。

$$(integer + integer) * integer \neq integer + (integer * integer)$$

この問題は Basic の式に関しても既に指摘してありました（165 ページ参照）。その時は演算子に別々の優先順位を付加することで解決しました。規則（R3）を規則（R2）より先に必ず還元することにします。これはすなわち掛け算に括弧を付けることに相当します。

+ 記号と * 記号の選択に関する問題は、文法を書き換えることでも解決できます。新しい非終端記号 T （項を表す）と F （因数を表す）を導入した、次のような文法を考えます。

$$\begin{array}{lcll} \mathrm{E} & ::= & \mathrm{E} + \mathrm{T} & (\mathrm{R1}) \\ & | & \mathrm{T} & (\mathrm{R2}) \\ \mathrm{T} & ::= & \mathrm{T} * \mathrm{F} & (\mathrm{R3}) \\ & | & \mathrm{F} & (\mathrm{R4}) \\ \mathrm{F} & ::= & integer & (\mathrm{R5}) \end{array}$$

この文法では *integer* + *integer* * *integer* を解釈するには規則（R1）を使うしかありません。

三番目の例はプログラミング言語の条件文と関係があります。Pascal などの言語では `if .. then` と `if .. then .. else` の二種類の条件文があります。次のような文法を考えてみましょう。

INSTR	::=	`if` EXP `then` INSTR	(R1)
	—	`if` EXP `then` INSTR `else` INSTR	(R2)
	—	（以下略）	

ここで次のような状況を考えてみましょう。

動作	入力	スタック
⋮		
	`else`...	[INSTR `then` EXP `if`...]
⋮		

この時にスタックの先頭部分が規則（R1）の条件文なのか（その場合、還元しなければならない）それとも規則（R2）の一番目の INSTR なのか（その場合、シフトしなければならない）決定できません。

シフトと還元の衝突に加えて、ボトムアップ解析は還元と還元が衝突することもあります。

We now introduce the `ocamlyacc` tool which uses the bottom-up parsing technique and may find these conflicts.

`ocamlyacc` ツール

`ocamlyacc` ツールは `ocamllex` と使い方は同じです。`ocamlyacc` ツールは、文法規則とその意味動作を記述したファイルを入力として受け取り、Objective Caml のソースプログラムを生成します。生成されるのは構文解析を行うプログラムを含むファイル（`.ml` ファイル）とそのインターフェース（`.mli`）です。

フォーマット　`ocamlyacc` の文法記述ファイルの拡張子は今までの慣習で `.mly` になっています。文法記述ファイルは次のような構造でなければなりません。

```
%{
    ヘッダ
}%
    宣言
%%
    文法規則
%%
    後処理
```

文法規則は次のようになっています。

```
非終端記号  :  記号. . . 記号  { 意味動作 }
            |  . . .
            |  記号. . . 記号  { 意味動作 }
            ;
```

「記号」は終端記号か非終端記号のどちらでも構いません。ocamllex の場合と同様にヘッダ部と後処理部は補助的な定義を書くのに使います。ただしヘッダ部で定義した内容は文法規則からのみ参照できます。したがってヘッダ部でモジュールをオープンしても宣言部ではその効果がありません。そのため宣言部では型名などを常に正式名称で記述しなければなりません。

意味動作　意味動作とは、ある文法規則に結びつけられた Objective Caml のプログラムであり、構文解析器がその文法規則を還元した時に実行されます。意味動作の中では文法規則の右辺の要素を番号で参照することができます。番号は 1 から始まり、左から右へ一つずつ増えていきます。最初の要素は $1 で参照でき、二番目の要素は $2 で参照できます。

開始記号　宣言部に（複数の）開始記号を宣言することができます。

```
%start 非終端記号 .. 非終端記号
```

これらの非終端記号ごとに構文解析を行う関数が生成されます。非終端記号の型は宣言部に記述しなければなりません。

```
%type <output-type> 非終端記号
```

この output-type は正式名称でなければなりません。

警告	非終端記号は構文解析を行う関数の名前になります。そのため大文字で始めることはできません（大文字で始まる記号は構築子と見なされるため）。

字句要素　文法規則は終端記号である字句要素を参照する必要があります。

終端記号を宣言するには次のように記述します。

```
%token PLUS MINUS MULT DIV MOD
```

識別子のようなある種の字句要素は文字列の集合を表しています。識別子は、それがどんな文字列であるのかという情報が重要です。ある字句要素が含んでいる内容の型は`<`と`>`で囲んで指定します。

```
%token <string> IDENT
```

警告 ocamlyacc はこれらの宣言を処理して *token* 型の構築子に変換します。したがって字句要素は大文字で始めなければなりません。

文字列を暗黙の終端記号として使うこともできます。

```
expr : expr "+" expr   { ... }
     | expr "*" expr   { ... }
     | ...
     ;
```

この場合、それ自身がその内容を表しているので宣言を行う意味はありません。暗黙の終端記号は字句解析器に渡されることなく、構文解析器が直接処理をします。字句解析器との整合性を保つためこの表記法は推奨されていません。

優先順位、結合律　ボトムアップ解析で衝突が発生する原因の多くは演算子の間の暗黙の結合法則、優先順位の扱いにあることを見てきました。このような衝突を扱うために演算子の優先順位と結合法則（右結合、左結合、非結合）を宣言することができます。以下の宣言では`+`演算子（記号 `PLUS`）と`*`演算子（記号 `MULT`）を左結合の演算子として宣言しています。また `MULT` が `PLUS` の後に宣言しているので`*`の方が`+`よりも高い優先順位を持っています。

```
%left PLUS
%left MULT
```

同じ行で宣言された演算子は同じ優先順位を持ちます。

コマンドオプション　ocamlyacc のオプションは二つあります。

- `-b name`: 生成される Objective Caml のファイルの名前が `name.ml` と `name.mli` になります。
- `-v`: `.output` という拡張子のファイルを生成します。このファイルの中には文法規則に割り振られた番号、文法を認識するオートマトンの状態、衝突の原因が記録されます。

`ocamllex` **との連携** `ocamllex` と `ocamlyacc` を一緒に組み合わせて、入力された文字列を字句の列に変換し、それを構文解析器に送ることもできます。これを行うためには字句を表す型 OCtypelexeme を双方のツールが知っていなければなりません。この型は、`.mly` の拡張子を持つファイルから `ocamlyacc` によって生成された `.mli` と `.ml` ファイルで定義されています。`.mll` ファイルはこの型をインポートし、`ocamllex` はこのファイルを *`Lexing.lexbuf -> lexeme`* という型を持つ関数に変換します。309 ページの例はこの相互依存の仕方について具体的に手順を踏んで説明しています。

文脈依存文法

`ocamlyacc` が処理する文法によって生成される言語のクラスは**文脈自由**文法です。文脈自由文法の構文解析器はある字句要素を処理する時にその結果が以前に処理された構文要素に依存していません。たとえば次の L のような言語では以前の処理の結果に依存する文法になっています。

$$L ::= wCw \mid w \text{ ただし } w \in (A|B)^*$$

ここで A、B、C は終端記号です。$(A|B)^*C(A|B)^*$ ではなくて wCw と書いたのは中央の C の左右に**同じ**語を置くことを表したかったからです。

L に含まれる語を認識するためには語 C 以前に読んだものを覚えておき、語 C 以後の部分と全く同じかどうか調べる必要があります。ここではストリームを利用した解決法を説明します。基本的なアイデアは全く同じ語を認識する構文解析関数を語 C より前の部分を認識する過程で作り出すというものです。

字句を表す型を定義します。

```
# type token = A | B | C ;;
```

関数 parse_w1 はストリーム解析関数を利用して最初の w を覚えておく関数を作り出します。その関数は一文字だけを認識するストリーム関数のリストとして定義されています。

```
# let rec parse_w1 s =
    match s with parser
      [<'A; l = parse_w1  >] → (parser [<'A >] → "a") :: l
    | [<'B; l = parse_w1  >] → (parser [<'B >] → "b") :: l
    | [< >] → [] ;;
Characters 48-50:
      [<'A; l = parse_w1  >] -> (parser [<'A >] -> "a")::l
      ^^
Syntax error
```

parse_w1 が返す関数の返り値は字句に対応する文字列です。

関数 parse_w2 は関数 parse_w1 が生成したリストを受け取り、個々の要素を一つの解析関数へと組み合わせます。

```
# let rec parse_w2 l =
    match l with
      p :: pl → (parser [< x = p; l = (parse_w2 pl)  >] → x^l)
    | []  → parser [<>] → "" ;;
Characters 58-60:
      p::pl -> (parser [< x = p; l = (parse_w2 pl)  >] -> x^l)
                       ^^
Syntax error
```

parse_w2 を適用した結果は語 w を表す文字列となります。parse_w2 の定義より、この関数は parse_w1 が読み込んだ語だけを認識し、それ以外の語を認識することはありません。

ストリームの中間結果を束縛する機構を使うと、言語 L の語を認識する関数を次のように書くことができます。

```
# let parse_L = parser  [< l = parse_w1 ; 'C; r = (parse_w2 l) >] → r ;;
Characters 23-25:
  let parse_L = parser  [< l = parse_w1 ; 'C; r = (parse_w2 l) >] -> r ;;
                        ^^
Syntax error
```

例として二つの語の解析を行います。最初の結果では C を囲む語を返しています。二番目の例では C の両側の語が違っているため認識に失敗しています。

```
# parse_L [< 'A; 'B; 'B; 'C; 'A; 'B; 'B >];;
Characters 9-11:
  parse_L [< 'A; 'B; 'B; 'C; 'A; 'B; 'B >];;
          ^^
Syntax error
# parse_L [< 'A; 'B; 'C; 'B; 'A >];;
Characters 8-10:
  parse_L [< 'A; 'B; 'C; 'B; 'A >];;
          ^^
Syntax error
```

Basic 再考

ocamllex と ocamlyacc を使って 169 ページに記載されている Basic のための構文解析関数を書き直してみましょう。今回は字句定義と文法定義ファイルから生成された関数を利用して、抽象的に記述することができます。

ここでは以前の字句定義を再利用することはできません。演算子、命令、キーワードを区別できるようにより正確な型を定義しなければなりません。

同様に、抽象構文を定義する型宣言 (sentences 型と sentences 型を定義するのに必要な型) を basic_types.mli ファイルの中に保存します。

basic_parser.mly ファイル

ヘッダー部 ヘッダー部では抽象構文を定義する型宣言をインポートします。それから文字列を抽象構文要素へと変換する補助関数を二つ定義します。

```
open Basic_types ;;

let phrase_of_cmd c =
 match c with
   "RUN" → Run
 | "LIST" → List
 | "END" → End
 | _ → failwith "line : unexpected command"
;;

let bin_op_of_rel r =
 match r with
   "=" → EQUAL
 | "<" → INF
 | "<=" → INFEQ
 | ">" → SUP
 | ">=" → SUPEQ
 | "<>" → DIFF
 | _ → failwith "line : unexpected relation symbol"
;;
```

宣言部 宣言部は字句宣言と字句の結合規則と優先順位、開始記号の三つの部分からなります。この例では開始記号は命令を意味する line です。

字句宣言部は次のようになります。

```
%token <int> Lint
%token <string> Lident
%token <string> Lstring
%token <string> Lcmd
%token Lplus Lminus Lmult Ldiv Lmod
%token <string> Lrel
%token Land Lor Lneg
%token Lpar Rpar
%token <string> Lrem
%token Lrem Llet Lprint Linput Lif Lthen Lgoto
%token Lequal
%token Leol
```

名前から意味を想像できると思います。これらの定義は basic_lexer.mll に記述されています (312 ページ参照)。

演算子の優先順位規則は priority_uop 関数と priority_binop 関数で定義されていました (160 ページ参照) が、ここでは以下のようになります。

```
%right Lneg
%left Land Lor
%left Lequal Lrel
%left Lmod
%left Lplus Lminus
%left Lmult Ldiv
%nonassoc Lop
```

最後のシンボル Lop は単項マイナス演算子を表すのに使います。このシンボルは終端記号ではありませんが、同じ記号が文脈によって別の意味と優先順位を持つ時に疑似的な終端記号として導入します。この点には文法規則を説明する時にまた触れます。

開始記号が line なので生成された関数は行を表す構文木を返すことになります。

```
%start line
%type <Basic_types.phrase> line
```

文法規則 Basic の文法規則は三つの非終端記号で表されています。行を表す line、命令を表す inst と式を表す exp です。文法規則と関連した意味動作は単純に文法規則に対応する抽象構文を組み立てているだけです。

```
%%
line :
   Lint inst Leol              { Line {num=$1; inst=$2} }
 | Lcmd Leol                   { phrase_of_cmd $1 }
 ;

inst :
   Lrem                        { Rem $1 }
 | Lgoto Lint                  { Goto $2 }
 | Lprint exp                  { Print $2 }
 | Linput Lident               { Input $2 }
 | Lif exp Lthen Lint          { If ($2, $4) }
 | Llet Lident Lequal exp      { Let ($2, $4) }
 ;

exp :
   Lint                        { ExpInt $1 }
 | Lident                      { ExpVar $1 }
 | Lstring                     { ExpStr $1 }
 | Lneg exp                    { ExpUnr (NOT, $2) }
 | exp Lplus exp               { ExpBin ($1, PLUS, $3) }
 | exp Lminus exp              { ExpBin ($1, MINUS, $3) }
 | exp Lmult exp               { ExpBin ($1, MULT, $3) }
 | exp Ldiv exp                { ExpBin ($1, DIV, $3) }
 | exp Lmod exp                { ExpBin ($1, MOD, $3) }
```

```
 | exp Lequal exp                { ExpBin ($1, EQUAL, $3) }
 | exp Lrel exp                 { ExpBin ($1, (bin_op_of_rel $2), $3) }
 | exp Land exp                 { ExpBin ($1, AND, $3) }
 | exp Lor exp                  { ExpBin ($1, OR, $3) }
 | Lminus exp %prec Lop        { ExpUnr(OPPOSITE, $2) }
 | Lpar exp Rpar                { $2 }
 ;
 %%
```

これらの文法規則の意味について特に説明はいらないでしょう。ただし次の規則

```
exp :
 ...
 | Lminus exp %prec Lop { ExpUnr(OPPOSITE, $2) }
```

については説明が必要かもしれません。この規則は単項マイナス演算子と関係しています。%prec キーワードはこの命令が Lop の優先順位を持つべきであると指定しています。

basic_lexer.mll ファイル

字句解析はひとつしか規則がありません。この lexer という規則は以前の関数 lexer (165 ページ) と密接に対応しています。それぞれの字句要素と結びつけられた動作規則は単純にその字句要素のコンストラクタを呼び出しているだけです。字句要素の型は文法規則のファイルで定義されてるので、そのファイルをインポートします。また単にダブルコーテーション記号を取り除く関数をここで定義します。

```
{
 open Basic_parser ;;

 let string_chars s = String.sub s 1 ((String.length s)-2) ;;
}

rule lexer = parse
   [' ' '\t']          { lexer lexbuf }

 | '\n'                { Leol }

 | '!'                 { Lneg }
 | '&'                 { Land }
 | '|'                 { Lor }
 | '='                 { Lequal }
 | '%'                 { Lmod }
 | '+'                 { Lplus }
 | '-'                 { Lminus }
 | '*'                 { Lmult }
 | '/'                 { Ldiv }

 | ['<' '>']           { Lrel (Lexing.lexeme lexbuf) }
```

```
| "<="                  { Lrel (Lexing.lexeme lexbuf) }
| ">="                  { Lrel (Lexing.lexeme lexbuf) }

| "REM" [^ '\n']*       { Lrem (Lexing.lexeme lexbuf) }
| "LET"                 { Llet }
| "PRINT"               { Lprint }
| "INPUT"               { Linput }
| "IF"                  { Lif }
| "THEN"                { Lthen }
| "GOTO"                { Lgoto }

| "RUN"                { Lcmd (Lexing.lexeme lexbuf) }
| "LIST"               { Lcmd (Lexing.lexeme lexbuf) }
| "END"                { Lcmd (Lexing.lexeme lexbuf) }

| ['0'-'9']+           { Lint (int_of_string (Lexing.lexeme lexbuf)) }
| ['A'-'z']+           { Lident (Lexing.lexeme lexbuf) }
| '"' [^ '"']* '"'     { Lstring (string_chars (Lexing.lexeme lexbuf)) }
```

ここで =記号は式と代入の両方で使われます。

やや複雑な正規表現を使っている二つの規則についてだけ説明します。最初の規則 (`"REM" [^ '\n']*`) はコメント行に関するものです。この規則はキーワード REM の後で改行 `'\n'` を含まない文字列とマッチします。また最後の規則 (`'"' [^ '"']* '"'`) はダブルコーテーション記号に囲まれた文字列を認識します。

コンパイルとリンク

字句解析器と構文解析器のコンパイルは決まった順序で行わなければなりません。なぜなら字句などのデータ構造の定義が相互に依存しているからです。この例のプログラムをコンパイルするには次のような順序でコマンドを実行する必要があります。

```
ocamlc -c basic_types.mli
ocamlyacc basic_parser.mly
ocamllex basic_lexer.mll
ocamlc -c basic_parser.mli
ocamlc -c basic_lexer.ml
ocamlc -c basic_parser.ml
```

この結果二つのファイル `basic_lexer.cmo` と `basic_parser.cmo` が生成されます。これらのファイルをリンクしてアプリケーションから利用できます。

以上で Basic の構文解析器を書き直す準備はすべて整いました。

「字句解析」(163 ページ) と「構文解析」(165 ページ) に記述されている型や関数の定義はここでは省略します。174 ページに記述されている関数 `one_command` の中の次の式

```
match parse (input_line stdin) with
```

を以下のように書き換えてください。

```
match line lexer (Lexing.from_string ((input_line stdin)^"\n")) with
```

ここで行の終わりに改行コード '\n' を付け加える必要があります。(input_line 関数は改行コードを取り除いてしまいます。) 改行コードは命令行の終わりを表すシンボル (Leol) として必要なものです。

練習問題

コメントの除去

Objective Caml ではコメントは階層的です。つまりコメントを含むコメントを記述できます。コメントは (* で始まり *) で終わります。例えば以下のようなコメントを記述できます。

```
(* comment spread
      over several
        lines *)

 let succ x = (* successor function *)
   x + 1;;

(* level 1 commented text
  let old_succ y = (* level 2 successor function level 2 *)
    y +1;;
    level 1 *)
  succ 2;;
```

この練習問題ではコメントを除去したテキストを書き出すプログラムを作ります。どんな字句解析器を使っても構いません。

1. Objective Caml のコメントを認識できる字句解析器 を書きなさい。コメントの始まりと終わりがきちんと対応していなければなりません。コメント以外の部分に含まれている語については自由とします。
2. ファイルを開き、コメントを除去し、残りの部分を新しいファイルに買きだすプログラム を書きなさい。
3. Objective Caml では文字列の中に任意の文字を含むことができます。例えば "what(*ever te*)xt" という文字列は有効で、この場合 (* と *) はコメントとは解釈されません。文字列を正しく解釈する字句解析器 を書きなさい。
4. 新しい字句解析器と プログラムを統合したプログラム を作りなさい。

評価器

ocamlyacc を式の評価に利用することができます。基本的なアイデアは式の評価を文法規則で直接実行することです。

ここでは必ず括弧で括られた前置記法の数式を対象にします。例えば（ADD e1 e2 .. en）という式は e1 + e2 + .. + en と等価です。ただし加算と乗算は右結合、減算と除算は左結合とします。

1. 式の構文解析と評価規則を opn_parser.mly ファイルに定義しなさい。
2. 式の字句解析器を opn_lexer.mll ファイルに定義しなさい。
3. 標準入力から式を一行読み込み、評価して結果を表示するプログラム opn を書きなさい。

まとめ

この章では Objective Caml で利用できる字句解析と構文解析のためのツールを紹介しています。

- 正規表現を扱うモジュール Str
- 単純な字句解析器を定義できるモジュール Genlex
- lex ツールに型システムを導入したツール ocamllex
- yacc ツールに型システムを導入したツール ocamlyacc
- 文脈依存構文解析器を含むトップダウン解析器を定義できるストリーム

159 ページで定義した Basic 言語の構文解析器は ocamllex と ocamlyacc を利用するとより簡潔で直観的に分かりやすい形式で記述できます。

もっと知りたい人へ

字句解析と構文解析のための標準的な本は親しみを込めて「ドラゴンブック」と呼ばれています。この本は正しくは *Compilers: principles, techniques and tools* ([ASU86]) という題名であり、表紙にはドラゴンの絵が使われています。この本はコンパイラの設計と実装のすべての面をカバーしています。文脈自由言語を解釈するオートマトンの構築法とその最小化の技法なども明解に説明されています。lex と yacc について詳しく説明されている本は何冊かありますが文献 [LMB92] を参照してください。ocamllex と ocamlyac がオリジナルのツールと比べて興味深い点は Objective Caml 言語と完全に統合されていることであり、とりわけ型付きの字句解析器と構文解析器を定義できることです。ストリームに関しては Michel Mauny と Daniel de Rauglaudre が明解な操作的意味を与え [MdR92] 、[CM98] が実装法について解説しています。新しい文法を Objective Caml 言語自体に組み込む方法については camlp4 ツールを参照してください。

リンク: http://caml.inria.fr/camlp4/

12

Interoperability with C

Developing programs in a given language very often requires one to integrate libraries written in other languages. The two main reasons for this are:

- to use libraries that cannot be written in the language, thus extending its functionality;
- to use high-performance libraries already implemented in another language.

A program then becomes an assembly of software components written in various languages, where each component has been written in the language most appropriate for the part of the problem it addresses. Those software components interoperate by exchanging values and requesting computations.

The Objective Caml language offers such a mechanism for interoperability with the C language. This mechanism allows Objective Caml code to call C functions with Caml-provided arguments, and to get back the result of the computation in Objective Caml. The converse is also possible: a C program can trigger an Objective Caml computation, then work on its result.

The choice of C as interoperability language is justified by the following reasons:

- it is a standardized language (ISO C);
- C is a popular implementation language for operating systems (Unix, Windows, MacOS, etc.);
- a great many libraries are written in C;
- most programming languages offer a C interface, thus it is possible to interface Objective Caml with these languages by going through C.

The C language can therefore be viewed as the esperanto of programming languages.

Cooperation between C and Objective Caml raises a number of difficulties that we review below.

- **Machine representation of data**
  For instance, values of base types (*int*, *char*, *float*) have different machine representations in the two languages. This requires conversion between the representations, in both directions. The same holds for data structures such as records, sum types[1], or arrays.
- **The Objective Caml garbage collector**
  Standard C does not provide garbage collection. (However, garbage collectors are easily written in C.) Moreover, calling a C function from Objective Caml must not modify the memory in ways incompatible with the Objective Caml GC.
- **Aborted computations**
  Standard C does not support exceptions, and provides different mechanisms for aborting computations. This complicates Objective Caml's exception handling.
- **Sharing common resources**
  For instance, files and other input-output devices are shared between Objective Caml and C, but each language maintains its own input-output buffers. This may violate the proper sequencing of input-output operations in mixed programs.

Programs written in Objective Caml benefit from the safety of static typing and automatic memory management. This safety must not be compromised by improper use of C libraries and interfacing with other languages through C. The programmer must therefore adhere to rather strict rules to ensure that both languages coexist peacefully.

Chapter outline

This chapter introduces the tools that allow interoperability between Objective Caml and C by building executables containing code fragments written in both languages. These tools include functions to convert between the data representations of each language, allocation functions using the Objective Caml heap and garbage collector, and functions to raise Objective Caml exceptions from C.

The first section shows how to call C functions from Objective Caml and how to build executables and interactive toplevel interpreters including the C code implementing those functions. The second section explores the C representation of Objective Caml values. The third section explains how to create and modify Objective Caml values from C. It discusses the interactions between C allocations and the Objective Caml garbage collector, and presents the mechanisms ensuring safe allocation from C. The fourth section describes exception handling: how to raise exceptions and how to handle them. The fifth section reverses the roles: it shows how to include Objective Caml code in an application whose main program is written in C.

1. Objective Caml's sum types are discriminated unions. Refer to chapter 2, page 44 for a full description.

注意

This chapter assumes a working knowledge of the C language. Moreover, reading chapter 9 can be helpful in understanding the issues raised by automatic memory management.

Communication between C and Objective Caml

Communication between parts of a program written in C and in Objective Caml is accomplished by creating an executable (or a new toplevel interpreter) containing both parts. These parts can be separately compiled. It is therefore the responsibility of the linking phase[2] to establish the connection between Objective Caml function names and C function names, and to create the final executable. To this end, the Objective Caml part of the program contains external declarations describing this connection.

Figure 12.1 shows a sample program composed of a C part and an Objective Caml part. Each part comprises code (function definitions and toplevel expressions for Objective

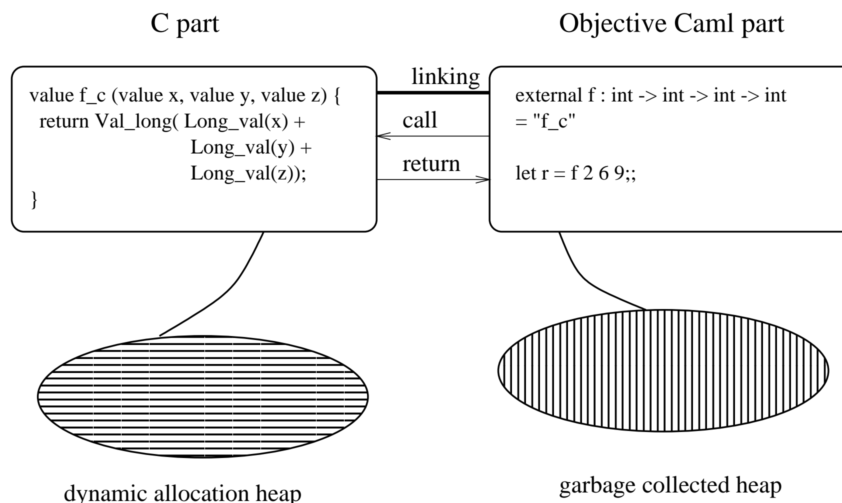


図 12.1: Communication between Objective Caml and C.

Caml) and a memory area for dynamic allocation. Calling the function `f` with three Objective Caml integer arguments triggers a call to the C function `f_c`. The body of the C function converts the three Objective Caml integers to C integers, computes their sum, and returns the result converted to an Objective Caml integer.

2. Linking is performed differently for the bytecode compiler and the native-code compiler.

We now introduce the basic mechanisms for interfacing C with Objective Caml: external declarations, calling conventions for C functions invoked from Objective Caml, and linking options. Then, we show an example using input-output.

External declarations

External function declarations in Objective Caml associate a C function definition with an Objective Caml name, while giving the type of the latter.

The syntax is as follows:

構文 :

```
external caml_name : type = "C_name"
```

This declaration indicates that calling the function *caml_name* from Objective Caml code performs a call to the C function *C_name* with the given arguments. Thus, the example in figure 12.1 declares the function `f` as the Objective Caml equivalent of the C function `f_c`.

An external function can be declared in an interface (*i.e.*, in an `.mli` file) either as an external or as a regular value:

構文 :

```
external caml_name : type = "C_name"
val caml_name : type
```

In the latter case, calls to the C function first go through the general function application mechanism of Objective Caml. This is slightly less efficient, but hides the implementation of the function as a C function.

Declaration of the C functions

C functions intended to be called from Objective Caml must have the same number of arguments as described in their external declarations. These arguments have type `value`, which is the C type for Objective Caml values. Since those values have uniform representations (第 9 章参照), a single C type suffices to encode all Objective Caml values. On page 325, we will present the facilities for encoding and decoding values, and illustrate them by a function that explores the representations of Objective Caml values.

The example in figure 12.1 respects the constraints mentioned above. The function `f_c`, associated with an Objective Caml function of type *int* -> *int* -> *int* -> *int*, is indeed a function with three parameters of type `value` returning a result of type `value`.

The Objective Caml bytecode interpreter evaluates calls to external functions differently, depending on the number of arguments[3]. If the number of arguments is less than or equal to five, the arguments are passed directly to the C function. If the number of arguments is greater than five, the C function's first parameter will get an array containing all of the arguments, and the C function's second parameter will get the number of arguments. These two cases must therefore be distinguished for external C functions that can be called from the bytecode interpreter. On the other hand, the Objective Caml native-code compiler always calls external functions by passing all the arguments directly, as function parameters.

External functions with more than five arguments

For external functions with more than five arguments, the programmer must provide two C functions: one for bytecode and the other for native-code. The syntax of external declarations allows the declaration of one Objective Caml function associated with two C functions:

構文 : **external** *caml_name* **:** *type* **=** "*C_name_bytecode*" "*C_name_native*"

The function *C_name_bytecode* takes two parameters: an array of values of type `value` (*i.e.* a C pointer of type `value*`) and an integer giving the number of elements in this array.

Example

The following C program defines two functions for adding together six integers: `plus_native`, callable from native code, and `plus_bytecode`, callable from the bytecode compiler. The C code must include the file `mlvalues.h` containing the definitions of C types, Objective Caml values, and conversion macros.

```
#include <stdio.h>
#include <caml/mlvalues.h>

value plus_native (value x1,value x2,value x3,value x4,value x5,value x6)
{
  printf("<< NATIVE PLUS >>\n") ; fflush(stdout) ;
  return Val_long ( Long_val(x1) + Long_val(x2) + Long_val(x3)
                  + Long_val(x4) + Long_val(x5) + Long_val(x6)) ;
}

value plus_bytecode (value * tab_val, int num_val)
{
  int i;
  long res;
```

3. Recall that a function such as `fst`, of type `'a * 'b -> 'a`, does not have two arguments, but only one that happens to be a pair; on the other hand, a function of type `int -> int -> int` has two arguments.

```
  printf("<< BYTECODED PLUS >> : ") ; fflush(stdout) ;
  for (i=0,res=0;i<num_val;i++) res += Long_val(tab_val[i]) ;
  return Val_long(res) ;
}
```

The following Objective Caml program `exOCAML.ml` calls these two C functions.

```
external plus : int → int → int → int → int → int → int
             = "plus_bytecode" "plus_native" ;;
print_int (plus 1 2 3 4 5 6) ;;
print_newline () ;;
```

We now compile these programs with the two Objective Caml compilers and a C compiler that we call `cc`. We must give it the access path for the `mlvalues.h` include file.

```
$ cc -c -I/usr/local/lib/ocaml  exC.c

$ ocamlc -custom exC.o exOCAML.ml -o ex_byte_code.exe
$ ex_byte_code.exe
<< BYTECODED PLUS >> : 21

$ ocamlopt exC.o exOCAML.ml -o ex_native.exe
$ ex_native.exe
<< NATIVE PLUS >> : 21
```

注意

To avoid writing the C function twice (with the same body but different calling conventions), it suffices to implement the bytecode version as a call to the native-code version, as in the following sketch:

```
value prim_nat (value x1, ..., value xn) { ... }
value prim_bc (value *tbl, int n)
{ return prim_nat(tbl[0],tbl[1],...,tbl[n-1]) ; }
```

Linking with C

The linking phase creates an executable from C and Objective Caml files compiled with their respective compilers. The result of the native-code compiler is shown in figure 12.2.

The compilation of the C and Objective Caml sources generates machine code that is stored in the static allocation area of the program. The dynamic allocation area contains the execution stack (corresponding to the function calls in progress) and the heaps for C and Objective Caml.

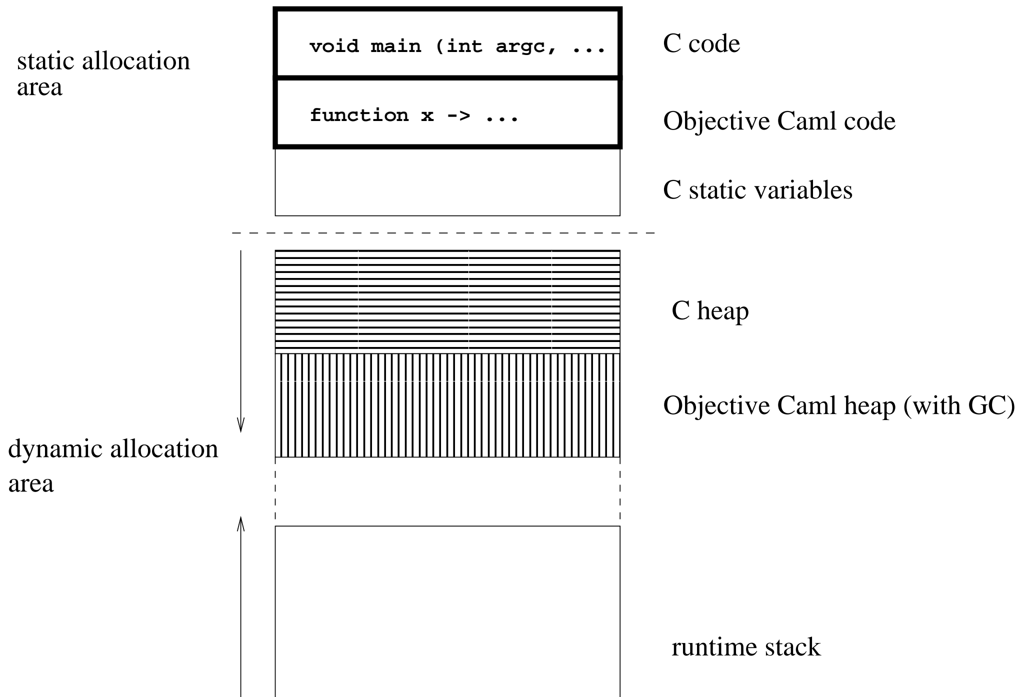


図 12.2: Mixed-language executable.

Run-time libraries

The C functions that can be called from a program using only the standard Objective Caml library are contained in the execution library of the abstract machine (see figure 7.3 page 200). For such a program, there is no need to provide additional libraries at link-time. However, when using Objective Caml libraries such as `Graphics`, `Num` or `Str`, the programmer must explicitly provide the corresponding C libraries at link-time. This is the purpose of the `-custom` compiler option (see 第 7 章参照, page 207). Similarly, when we wish to call our C functions from Objective Caml, we must provide the object file containing those C functions at link-time. The following example illustrates this.

The three linking modes

The linking commands differ slightly between the native-code compiler, the bytecode compiler, and the construction of toplevel interactive loops. The compiler options relevant to these linking modes are described in chapter 7.

To illustrate these linking modes, we consider again the example in figure 12.1. Assume the Objective Caml source file is named `progocaml.ml`. It uses the external function `f_c` defined in the C file `progC.c`. In turn, the function `f_c` refers to a C library

a_C_library.a. Once all these files are compiled separately, we link them together using the following commands:

- bytecode:
  `ocamlc -custom -o vbc.exe progC.o a_C_library.a progocaml.cmo`
- native code:
  `ocamlopt progC.o -o vn.exe a_C_library.a progocaml.cmx`

We obtain two executable files: vbc.exe for the bytecode version, and vn.exe for the native-code version.

Building an enriched abstract machine

Another possibility is to augment the run-time library of the abstract machine with new C functions callable from Objective Caml. This is achieved by the following commands:

```
ocamlc -make-runtime -o new_ocamlrun progC.o a_C_library.a
```

We can then build a bytecode executable vbcnam.exe targeted to the new abstract machine:

```
ocamlc -o vbcnam.exe -use-runtime new_ocamlrun progocaml.cmo
```

To run this bytecode executable, either give it as the first argument to the new abstract machine, as in new_ocaml vbcnam.exe , or run it directly as vbcnam.exe

注意

> Linking in -custom mode scans the object files (.cmo) to build a table of all external functions mentioned. The bytecode required to use them is generated and added to the bytecode corresponding to the Objective Caml code.

Building a toplevel interactive loop

To be able to use an external function in the toplevel interactive loop, we must first build a new toplevel interpreter containing the C code for the function, as well as an Objective Caml file containing its declaration.

We assume that we have compiled the file progC.c containing the function f_c. We then build the toplevel loop ftop as follows:

```
ocamlmktop -custom -o ftop progC.o a_C_library.a ex.ml
```

The file ex.ml contains the external declaration for the function f. The new toplevel interpreter ftop then knows this function and contains the corresponding C code, as found in progC.o.

Mixing input-output in C and in Objective Caml

The input-output functions in C and in Objective Caml do not share their file buffers. Consider the following C program:

```
#include <stdio.h>
#include <caml/mlvalues.h>
value hello_world (value v)
  { printf("Hello World !!");  fflush(stdout);  return v; }
```

Writes to standard output must be flushed explicitly (`fflush`) to guarantee that they will be printed in the intended order.

```
# external caml_hello_world : unit → unit = "hello_world"  ;;
external caml_hello_world : unit -> unit = "hello_world"
# print_string "<< " ;
  caml_hello_world () ;
  print_string " >>\n" ;
  flush stdout ;;
The external function ‘hello_world’ is not available
```

The outputs from C and from Objective Caml are not intermingled as expected, because each language buffers its outputs independently. To get the correct behavior, the Objective Caml part must be rewritten as follows:

```
# print_string "<< " ; flush stdout ;
  caml_hello_world () ;
  print_string " >>\n" ; flush stdout ;;
The external function ‘hello_world’ is not available
```

By flushing the Objective Caml output buffer after each write, we ensure that the outputs from each language appear in the expected order.

Exploring Objective Caml values from C

The machine representation of Objective Caml values differs from that of C values, even for fundamental types such as integers. This is because the Objective Caml garbage collector needs to record additional information in values. Since Objective Caml values are represented uniformly, their representations all belong to the same C type, named (unsurprisingly) `value`.

When Objective Caml calls a C function, passing it one or several arguments, those arguments must be decoded before using them in the C function. Similarly, the result of this C function must be encoded before being returned to Objective Caml.

These conversions (decoding and encoding) are performed by a number of macros and C functions provided by the Objective Caml runtime system. These macros and functions are declared in the include files listed in figure 12.3. These include files are part of the

Objective Caml installation, and can be found in the directory where Objective Caml libraries are installed[4]

`caml/mlvalues.h`	definition of the `value` type and basic value conversion macros.
`caml/alloc.h`	functions for allocating Objective Caml values.
`caml/memory.h`	macros for interfacing with the Objective Caml garbage collector.

図 12.3: Include files for the C interface.

Classification of Objective Caml representations

An Objective Caml representation, that is, a C datum of type `value`, is one of:

- an immediate value (represented as an integer);
- a pointer into the Objective Caml heap;
- a pointer pointing outside the Objective Caml heap.

The Objective Caml heap is the memory area that is managed by the Objective Caml garbage collector. C code can also allocate and manipulate data structures in its own memory space, and communicate pointers to these data structures to Objective Caml.

Figure 12.4 shows the macros for classifying representations and converting between C integers and their Objective Caml representation. Note that C offers several integer

`Is_long(v)`	is `v` an Objective Caml integer?
`Is_block(v)`	is `v` an Objective Caml pointer?
`Long_val(v)`	extract the integer contained in `v`, as a C "long"
`Int_val(v)`	extract the integer contained in `v`, as a C "int"
`Bool_val(v)`	extract the boolean contained in `v` (0 if `false`, non-zero if `true`)

図 12.4: Classification of representations and conversion of immediate values.

types of varying sizes (`short`, `int`, `long`, etc), while Objective Caml has only one integer type, `int`.

4. Under Unix, this directory is `/usr/local/lib/ocaml` by default, or sometimes `/usr/lib/ocaml`. Under Windows, the default location is `C:\OCAML\LIB`, or the value of the environment variable `CAMLLIB`, if set.

Accessing immediate values

All Objective Caml immediate values are represented as integers:

- integers are represented by their value;
- characters are represented by their ASCII code[5];
- constant constructors are represented by an integer corresponding to their position in the datatype declaration: the nth constant constructor of a datatype is represented by the integer $n - 1$.

The following program defines a C function `inspect` that inspects the representation of its argument:

```
#include <stdio.h>
#include <caml/mlvalues.h>
value inspect (value v)
{
  if (Is_long(v))
    printf ("v is an integer (%ld) : %ld", (long) v, Long_val(v));
  else if (Is_block(v))
    printf ("v is a pointer");
  else
    printf ("v is neither an integer nor a pointer (???)");
  printf("   ");
  fflush(stdout) ;
  return v ;
}
```

The function `inspect` tests whether its argument is an Objective Caml integer. If so, it prints the integer twice, first viewed as a C long integer (without conversion), then converted by the `Long_val` macro, which extracts the actual integer represented in the argument.

On the following example, we see that the machine representation of integers in Objective Caml differs from that of C:

```
# external inspect : 'a → 'a = "inspect" ;;
external inspect : 'a -> 'a = "inspect"
# inspect 123 ;;
The external function 'inspect' is not available
# inspect max_int;;
The external function 'inspect' is not available
```

We can also inspect values of other predefined types, such as *char* and *bool*:

```
# inspect 'A' ;;
The external function 'inspect' is not available
# inspect true ;;
The external function 'inspect' is not available
```

5. More precisely, by their ISO Latin-1 code, which is an 8-bit character encoding extending ASCII with accented letters and signs for Western languages. Objective Caml does not yet handle wider internationalized character sets such as Unicode.

```
# inspect false ;;
The external function ‘inspect’ is not available
# inspect [] ;;
The external function ‘inspect’ is not available
```

Consider the Objective Caml type `foo` defined thus:

```
# type foo = C1 | C2 of int | C3 | C4 ;;
```

The `inspect` function shows that constant constructors and non-constant constructors of this type are represented differently:

```
# inspect C1 ;;
The external function ‘inspect’ is not available
# inspect C4 ;;
The external function ‘inspect’ is not available
# inspect (C2 1) ;;
The external function ‘inspect’ is not available
```

When the function `inspect` detects an immediate value, it prints first the “physical” representation of this value (*i.e.* the representation viewed as a word-sized C integer of C type `long`); then it prints the “logical” contents of this value (*i.e.* the Objective Caml integer it represents, as returned by the decoding macro `Long_val`). The examples above show that the “physical” and the “logical” contents differ. This difference is due to the tag bit[6] used by the garbage collector to distinguish immediate values from pointers (see chapter 9, page 255).

Representation of structured values

Non-immediate Objective Caml values are said to be structured values. Those values are allocated in the Objective Caml heap and represented as a pointer to the corresponding memory block. All memory blocks contain a header word indicating the kind of the block as well as its size expressed in machine words. Figure 12.5 shows the structure of a block for a 32-bit machine. The two “color” bits are used by the garbage collector for walking the memory graph (see chapter 9, page 256). The “tag” field, or “tag” for short, contains the kind of the block. The “size” field contains the size of the block, in words, excluding the header. The macros listed in figure 12.6 return the tag and size of a block. The tag of a memory block can take the values listed in figure 12.7. Depending on the block tag, different macros are used to access the contents of the blocks. These macros are described in figure 12.8. When the tag is less than `No_scan_tag`, the heap block is structured as an array of Objective Caml value representations. Each element of the array is called a “field” of the memory block. In accordance with C and Objective Caml conventions, the first field is at index 0, and the last field is at index `Wosize_val(v) - 1`.

6. Here, the tag bit is the least significant bit.

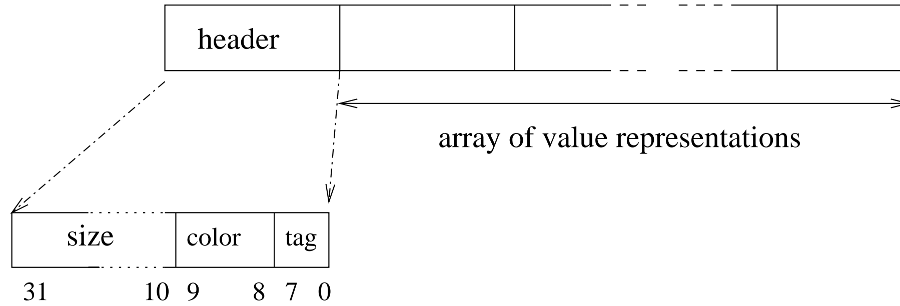


図 12.5: Structure of an Objective Caml heap block.

`Wosize_val(v)`	return the size of the block `v` (header excluded)
`Tag_val(v)`	return the tag of the block `v`

図 12.6: Accessing header information in memory blocks.

As we did earlier for immediate values, we now define a function to inspect memory blocks. The C function `print_block` takes an Objective Caml value representation, tests whether it is an immediate value or a memory block, and in the latter case prints the kind and contents of the block. It is called from the wrapper function `inspect_block`, which can be called from Objective Caml.

```
#include <stdio.h>
#include <caml/mlvalues.h>

void margin (int n)
  { while (n-- > 0) printf(".");  return; }

void print_block (value v,int m)
{
```

from `0` to `No_scan_tag-1`	an array of Objective Caml value representations
`Closure_tag`	a function closure
`String_tag`	a character string
`Double_tag`	a double-precision float
`Double_array_tag`	an array of float
`Abstract_tag`	an abstract data type
`Final_tag`	an abstract data type equipped with a finalization function

図 12.7: Tags of memory blocks.

`Field(v,n)`	return the $\mathtt{n}^{th}$ field of `v`.
`Code_val(v)`	return the code pointer for a closure.
`string_length(v)`	return the length of a string.
`Byte(v,n)`	return the $\mathtt{n}^{th}$ character of a string, with C type `char`.
`Byte_u(v,n)`	same, but result has C type `unsigned char`.
`String_val(v)`	return the contents of a string with C type `(char *)`.
`Double_val(v)`	return the float contained in `v`.
`Double_field(v,n)`	return the $\mathtt{n}^{th}$ float contained in the float array `v`.

図 12.8: Accessing the content of a memory block.

```
  int size, i;
  margin(m);
  if (Is_long(v))
    { printf("immediate value (%d)\n", Long_val(v));  return; };
  printf ("memory block: size=%d  -  ", size=Wosize_val(v));
  switch (Tag_val(v))
   {
    case Closure_tag :
        printf("closure with %d free variables\n", size-1);
        margin(m+4); printf("code pointer: %p\n",Code_val(v)) ;
        for (i=1;i<size;i++)  print_block(Field(v,i), m+4);
        break;
    case String_tag :
        printf("string: %s (%s)\n", String_val(v),(char *) v);
        break;
    case Double_tag:
        printf("float: %g\n", Double_val(v));
        break;
    case Double_array_tag :
        printf ("float array: ");
        for (i=0;i<size/Double_wosize;i++)  printf("  %g", Double_field(v,i));
        printf("\n");
        break;
    case Abstract_tag : printf("abstract type\n"); break;
    case Final_tag : printf("abstract finalized type\n"); break;
    default:
        if (Tag_val(v)>=No_scan_tag) { printf("unknown tag"); break; };
        printf("structured block (tag=%d):\n",Tag_val(v));
        for (i=0;i<size;i++)  print_block(Field(v,i),m+4);
   }
  return ;
}
```

```
value inspect_block (value v)
  { print_block(v,4); fflush(stdout); return v; }
```

Each possible tag for a block corresponds to a case of the `switch` construct. In the case of a block containing an array of Objective Caml values, we recursively call `print_block` on each field of the array. We then redefine the `inspect` function:

```
# external inspect : 'a -> 'a = "inspect_block" ;;
external inspect : 'a -> 'a = "inspect_block"
```

We can now explore the representations of Objective Caml structured values. We must be careful not to apply `inspect_block` to a cyclic value, since the recursive traversal of the value would then loop indefinitely.

Arrays, tuples, and records

Arrays and tuples are represented by structured blocks. The nth field of the block contains the representation of the nth element of the array or tuple.

```
# inspect [| 1; 2; 3 |] ;;
The external function 'inspect_block' is not available
# inspect ( 10 , true , () ) ;;
The external function 'inspect_block' is not available
```

Records are also represented as structured blocks. The values of the record fields appear in the order given at record declaration time. Mutable fields and immutable fields are represented identically.

```
# type foo = { fld1: int ; mutable fld2: int } ;;
type foo = { fld1 : int; mutable fld2 : int; }
# inspect { fld1=10 ; fld2=20 } ;;
The external function 'inspect_block' is not available
```

警告 Nothing prevents a C function from physically modifying an immutable record field. It is the programmers' responsibility to make sure that their C functions do not introduce inconsistencies in Objective Caml data structures.

Sum types

We previously saw that constant constructors are represented like integers. A non-constant constructor is represented by a block containing the constructor's arguments, with a tag identifying the constructor. The tag associated with a non-constant constructor represents its position in the type declaration: the first non-constant constructor has tag 0, the second one has tag 1, and so on.

```
# type foo = C1 of int * int * int | C2 of int | C3 | C4 of int * int ;;
type foo = C1 of int * int * int | C2 of int | C3 | C4 of int * int
# inspect (C1 (1,2,3)) ;;
```

```
The external function ‘inspect_block’ is not available
# inspect (C4 (1,2)) ;;
The external function ‘inspect_block’ is not available
```

注意

> The type *list* is a sum type whose declaration is:
> **type** *'a list* = [] | :: **of** *'a * 'a list*. This type has only one non-constant constructor (::). Thus, a non-empty list is represented by a memory block with tag 0.

Character strings

Characters inside strings occupy one byte each. Thus, the memory block representing a string uses one word per group of four characters (on a 32-bit machine) or eight characters (on a 64-bit machine).

警告

> Objective Caml strings can contain the null character whose ASCII code is 0. In C, the null character represents the end of a string, and cannot appear inside a string.

```
#include <stdio.h>
#include <caml/mlvalues.h>

value explore_string (value v)
{
  char *s;
  int i,size;
  s = (char *) v;
  size = Wosize_val(v) * sizeof(value);
  for (i=0;i<size;i++)
    {
      int p = (unsigned int) s[i] ;
      if ((p>31) && (p<128)) printf("%c",s[i]); else printf("(#%u)",p);
    }
  printf("\n");
  fflush(stdout);
  return v;
}
```

The length and position of last character of an Objective Caml string are determined not by looking for a terminating null character, as in C, but by combining the size of the memory block that contains the string with the last byte of the last word of this block, which indicates the number of *unused* bytes in the last word. The following examples clarify the role played by this last byte.

```
# external explore : string -> string = "explore_string" ;;
external explore : string -> string = "explore_string"
```

```
# ignore(explore "");
  ignore(explore "a");
  ignore(explore "ab");
  ignore(explore "abc");
  ignore(explore "abcd");
  ignore(explore "abcd\000") ;;
The external function ‘explore_string’ is not available
```

In the last two examples ("abcd" and "abcd\000"), the strings are of length 4 and 5 respectively. This explains why the last byte takes two different values, although the other bytes of the string representations are identical.

Floats and float arrays

Objective Caml offers only one type (*float*) of floating-point numbers. This type corresponds to 64-bit, double-precision floating point numbers in C (type `double`). Values of type *float* are heap-allocated and represented by a memory block of size 2 words (on a 32-bit machine) or 1 word (on a 64-bit machine).

```
# inspect 1.5 ;;
The external function ‘inspect_block’ is not available
# inspect 0.0;;
The external function ‘inspect_block’ is not available
```

Arrays of floats are represented specially to reduce their memory occupancy: the floats contained in the array are stored consecutively in the memory block, rather than having each float heap-allocated separately. Therefore, float arrays possess a specific tag and specific access macros.

```
# inspect [| 1.5 ; 2.5 ; 3.5 |] ;;
The external function ‘inspect_block’ is not available
```

This optimized representation encourages the use of Objective Caml for numerical computations that manipulate many float arrays: operations on array elements are much more efficient than if each float was heap-allocated separately.

警告 When allocating an Objective Caml float array from C, the size of the block should be the number of array elements multiplied by `Double_wosize`. The `Double_wosize` macro represents the number of words occupied by a double-precision float (2 words on a 32-bit machine, but only 1 word on a 64-bit machine).

With the exception of float arrays, floating-point numbers contained in other data structures are always treated as a structured, heap-allocated value. The following example shows the representation of a list of floats.

```
# inspect [ 3.14; 1.2; 7.6];;
The external function ‘inspect_block’ is not available
```

The list is viewed as a block with size 2, containing its head and its tail. The head of the list is a float, which is also a block of size 2.

Closures

A function value is represented by the code to be executed when the function is applied, and by its environment (see chapter 2, page 23). There are two ways to build a function value: either by explicit abstraction (as in `fun x -> x+1`) or by partial application of a curried function (as in `(fun x -> fun y -> x+y) 1`).

The environment of a closure can contain three kinds of variables: those declared globally, those declared locally, and the function parameters already instantiated by a partial application. The implementation treats those three kinds differently. Global variables are stored in a global environment that is not explicitly part of any closure. Local variables and instantiated parameters can appear in closures, as we now illustrate.

A closure with an empty environment is simply a memory block containing a pointer to the code of the function:

```
# let f = fun x y z → x+y+z ;;
val f : int -> int -> int -> int = <fun>
# inspect f ;;
The external function ‘inspect_block’ is not available
```

Functions with free local variables are represented by closures with non-empty environments. Here, the closure contains both a pointer to the code of the function, and the values of its free local variables.

```
# let g = let x = 1 and y = 2 in fun z → x+y+z ;;
val g : int -> int = <fun>
# inspect g ;;
The external function ‘inspect_block’ is not available
```

The Objective Caml virtual machine treats partial applications of functions specially for better performance. A partial application of an abstraction is represented by a closure containing a value for each of the instantiated parameters, plus a pointer to the closure for the initial abstraction.

```
# let a1 = f 1 ;;
val a1 : int -> int -> int = <fun>
# inspect (a1) ;;
The external function ‘inspect_block’ is not available
# let a2 = a1 2 ;;
val a2 : int -> int = <fun>
# inspect (a2) ;;
The external function ‘inspect_block’ is not available
```

Figure 12.9 depicts the result of the inspection above.

The function `f` has no free variables, hence the environment part of its closure is empty. The code pointer for a function with several arguments points to the code that should be called when all arguments are provided. In the case of `f`, this is the code corresponding to `x+y+z`. Partial applications of this function result in intermediate closures that point to a shared code (it is the same code pointer for `a1` and `a2`). The role of this code is

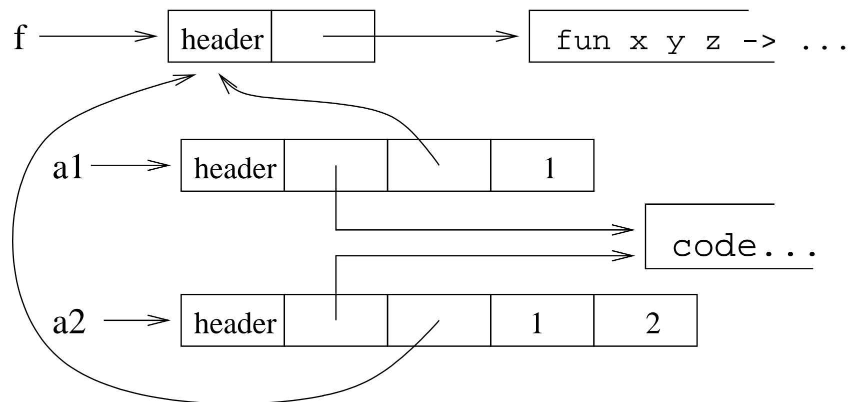


図 12.9: Closure representation.

to accumulate the arguments and detect when all arguments have been provided. If so, it pushes all arguments and calls the actual code for the function body; if not, it creates a new closure. For instance, the application of `a1` to 2 fails to provide all arguments to the function `f` (the last argument is still missing), hence a closure is created containing the first two arguments, 1 and 2. Notice that the closures resulting from partial applications always contain, in the first environment slot, a pointer to the original closure. The original closure will be called when all arguments have been gathered.

Mixing local declarations and partial applications results in the following representation:

```
# let g x = let y=2 in fun z → x+y+z ;;
val g : int -> int -> int = <fun>
# let a1 = g 1 ;;
val a1 : int -> int = <fun>
# inspect a1 ;;
The external function ‘inspect_block’ is not available
```

Abstract types

Values of an abstract type are represented like those of its implementation type. Actually, type information is used only during type-checking and compilation. During execution, the types are not needed – only the memory representation (tag bits on values, size and tag fields on memory blocks) needs to be communicated to the garbage collector.

For instance, a value of the abstract type *'a Stack.t* is represented as a reference to a list, since the type *'a Stack.t* is implemented as *'a list ref*.

```
# let p = Stack.create();;
val p : '_a Stack.t = <abstr>
```

```
# Stack.push 3 p;;
- : unit = ()
# inspect p;;
The external function 'inspect_block' is not available
```

On the other hand, some abstract types are implemented by representations that cannot be expressed in Objective Caml. Typical examples include arrays of weak pointers and input-output channels. Often, values of those abstract types are represented as memory blocks with tag Abstract_tag.

```
# let w = Weak.create 10;;
val w : '_a Weak.t = <abstr>
# Weak.set w 0 (Some p);;
- : unit = ()
# inspect w;;
The external function 'inspect_block' is not available
```

Sometimes, a finalization function is attached to those values. Finalization functions are C functions which are called by the garbage collector just before the value is collected. They are very useful to free external resources, such as an input-output buffer, just before the memory block referring to those resources disappears. For instance, inspection of the "standard output" channel reveals that the type *out_channel* is represented by abstract memory blocks with a finalization function:

```
# inspect (stdout) ;;
The external function 'inspect_block' is not available
```

Creating and modifying Objective Caml values from C

A C function called from Objective Caml can modify its arguments in place, or return a newly-created value. This value must match the Objective Caml type for the function result. For base types, several C macros are provided to convert a C datum to an Objective Caml value. For structured types, the new value must be allocated in the Objective Caml heap, with the correct size, and its fields initialized with values of the correct types. Considerable care is required here: it is easy to construct bad values from C, and these bad values may crash the Objective Caml program.

Any allocation in the Objective Caml heap can trigger a garbage collection, which will deallocate unused memory blocks and may move live blocks. Therefore, any Objective Caml value manipulated from C must be registered with the Objective Caml garbage collector, if they are to survive the allocation of a new block. These values must be treated as extra memory roots by the garbage collector. To this end, several macros are provided for registering extra roots with the garbage collector.

Finally, C code can allocate Objective Caml heap blocks that contain C data instead of Objective Caml values. This C data will then benefit from Objective Caml's automatic memory management. If the C data requires explicit deallocation, a finalization function can be attached to the heap block.

Modifying Objective Caml values

The following macros allow the creation of immediate Objective Caml values from the corresponding C data, and the modification of structured values in place.

`Val_long(l)`	return the value representing the long integer `l`
`Val_int(i)`	return the value representing the integer `l`
`Val_bool(x)`	return `false` if `x=0`, `true` otherwise
`Val_true`	the representation of `true`
`Val_false`	the representation of `false`
`Val_unit`	the representation of `()`
`Store_field(b,n,v)`	store the value `v` in the `n`-th field of block `b`
`Store_double_field(b,n,d)`	store the float `d` in the `n`-th field of the float array `b`

図 12.10: Creation of immediate values and modification of structured blocks.

Moreover, the macros `Byte` and `Byte_u` can be used on the left-hand side of an assignment to modify the characters of a string. The `Field` macro can also be used for assignment on blocks with tag `Abstract_tag` or `Final_tag`; use `Store_field` for blocks with tag between `0` and `No_scan_tag-1`. The following function reverses a character string in place:

```
#include <caml/mlvalues.h>
value swap_char(value v, int i, int j)
  { char c=Byte(v,i); Byte(v,i)=Byte(v,j); Byte(v,j)=c; }
value swap_string (value v)
{
  int i,j,t = string_length(v) ;
  for (i=0,j=t-1; i<t/2; i++,j--)  swap_char(v,i,j) ;
  return v ;
}
```

```
# external mirror : string → string = "swap_string" ;;
external mirror : string -> string = "swap_string"
# mirror "abcdefg" ;;
The external function ‘swap_string’ is not available
```

Allocating new blocks

The functions listed in figure 12.11 allocate new blocks in the Objective Caml heap. The function `alloc_array` takes an array of pointers `a`, terminated by a null pointer, and a conversion function `f` taking a pointer and returning a value. The result of `alloc_array` is an Objective Caml array containing the results of applying `f` in turn to each pointer in

`alloc(n, t)`	return a new block of size `n` words and tag `t`
`alloc_tuple(n)`	same, with tag 0
`alloc_string(n)`	return an uninitialized string of length `n` characters
`copy_string(s)`	return a string initialized with the C string `s`
`copy_double(d)`	return a block containing the double float `d`
`alloc_array(f, a)`	return a block representing an array, initialized by applying the conversion function `f` to each element of the C array of pointers `a`, null-terminated.
`copy_string_array(p)`	return a block representing an array of strings, obtained from the C string array `p` (of type `char **`), null-terminated.

図 12.11: Functions for allocating blocks.

`a`. In the following example, the function `make_str_array` uses `alloc_array` to convert a C array of strings.

```
#include <caml/mlvalues.h>
value make_str (char *s) { return copy_string(s); }
value make_str_array (char **p) { return alloc_array(make_str,p) ; }
```

It is sometimes necessary to allocate blocks of size 0, for instance to represent an empty Objective Caml array. Such a block is called an **atom**.

```
# inspect [| |] ;;
The external function 'inspect_block' is not available
```

Because atoms are allocated statically and do not reside in the dynamic part of the Objective Caml heap, the allocation functions in figure 12.11 must not be used to allocate atoms. Instead, atoms are created in C by the macro `Atom(t)`, where `t` is the desired tag for the block of size 0.

Storing C data in the Objective Caml heap

It is sometimes convenient to use the Objective Caml heap to store arbitrary C data that does not respect the constraints imposed by the garbage collector. In this case, blocks with tag `Abstract_tag` must be used.

A natural example is the manipulation of native C integers (of size 32 or 64 bits) in Objective Caml. Since these integers are not tagged as the Objective Caml garbage collector expects, they must be kept in one-word heap blocks with tag `Abstract_tag`.

```
#include <caml/mlvalues.h>
#include <stdio.h>
```

```
value Cint_of_OCAMLint (value v)
{
  value res = alloc(1,Abstract_tag) ;
  Field(res,0) = Long_val(v) ;
  return res ;
}

value OCAMLint_of_Cint (value v)  { return Val_long(Field(v,0)) ; }

value Cplus (value v1,value v2)
{
  value res = alloc(1,Abstract_tag) ;
  Field(res,0) = Field(v1,0) + Field(v2,0) ;
  return res ;
}

value printCint (value v)
{
  printf ("%d",(long) Field(v,0)) ; fflush(stdout) ;
  return Val_unit ;
}
```

```
# type cint
  external cint_of_int : int → cint = "Cint_of_OCAMLint"
  external int_of_cint : cint → int = "OCAMLint_of_Cint"
  external plus_cint : cint → cint → cint = "Cplus"
  external print_cint : cint → unit = "printCint" ;;
```

We can now work on native C integers, without losing the use of the tag bit, while remaining compatible with Objective Caml's garbage collector. However, such integers are heap-allocated, instead of being immediate values, which renders arithmetic operations less efficient.

```
# let a = 1000000000 ;;
val a : int = 1000000000
# a+a ;;
- : int = -147483648
# let c = let b = cint_of_int a in plus_cint b b ;;
The external function ‘Cint_of_OCAMLint’ is not available
# print_cint c ; print_newline () ;;
Characters 11-12:
  print_cint c ; print_newline () ;;
             ^
This expression has type Gc.control but is here used with type cint
The type constructor cint would escape its scope
# int_of_cint c ;;
Characters 12-13:
  int_of_cint c ;;
```

```
              ^
This expression has type Gc.control but is here used with type cint
The type constructor cint would escape its scope
```

Finalization functions

Abstract blocks can also contain pointers to memory blocks allocated outside the Objective Caml heap. We know that Objective Caml blocks that are no longer used by the program are deallocated by the garbage collector. But what happens to a block allocated in the C heap and referenced by an abstract block that was reclaimed by the GC? To avoid memory leaks, we can associate a **finalization function** to the abstract block; this function is called by the GC before reclaiming the abstract block.

An abstract block with an attached finalization function is allocated via the function `alloc_final (n, f, used, max)` .

- `n` is the size of the block, in words. The first word of the block is used to store the finalization function; hence the size occupied by the user data must be increased by one word.
- `f` is the finalization function itself, with type `void f (value)`. It receives the abstract block as argument, just before this block is reclaimed by the GC.
- `used` represents the memory space (outside the Objective Caml heap) occupied by the C data. `used` must be ¡= `max`.
- `max` is the maximum memory space outside the Objective Caml heap that we tolerate not being reclaimed immediately.

For efficiency reasons, the Objective Caml garbage collector does not reclaim heap blocks as soon as they become unused, but some time later. The ratio `used/max` controls the proportion of finalized abstract blocks that the garbage collector may leave allocated while they are no longer used. A ratio of 0 (that is, `used = 0`) lets the garbage collector work at its usual pace; higher ratios (no greater than 1) cause it to work harder and spend more CPU time finding unused finalized blocks and reclaiming them.

The following program manipulates arrays of C integers allocated in the C heap via `malloc`. To allow the Objective Caml garbage collector to reclaim these arrays automatically, the `create` function wraps them in a finalized abstract block, containing both a pointer to the array and the finalization function `finalize_it`.

```
#include <malloc.h>
#include <stdio.h>
#include <caml/mlvalues.h>

typedef struct {
        int size ;
        long * tab ; } IntTab ;

IntTab *alloc_it (int s)
```

```
{
  IntTab *res = malloc(sizeof(IntTab)) ;
  res->size = s ;
  res->tab = (long *) malloc(sizeof(long)*s) ;
  return res ;
}
void free_it (IntTab *p)  { free(p->tab) ; free(p) ; }
void put_it (int n,long q,IntTab *p)  { p->tab[n] = q ; }
long get_it (int n,IntTab *p)  { return p->tab[n]; }

void finalize_it (value v)
{
  IntTab *p = (IntTab *) Field(v,1) ;
  int i;
  printf("reclamation of an IntTab by finalization [") ;
  for (i=0;i<p->size;i++) printf("%d ",p->tab[i]) ;
  printf("]\n"); fflush(stdout) ;
  free_it ((IntTab *) Field(v,1)) ;
}
value create (value s)
{
  value block ;
  block = alloc_final (2, finalize_it,Int_val(s)*sizeof(IntTab),100000) ;
  Field(block,1) = (value) alloc_it(Int_val(s)) ;
  return block ;
}
value put (value n,value q,value t)
{
  put_it (Int_val(n), Long_val(q), (IntTab *) Field(t,1)) ;
  return Val_unit ;
}
value get (value n,value t)
{
  long res = get_it (Int_val(n), (IntTab *) Field(t,1)) ;
  return Val_long(res) ;
}
```

The C functions visible from Objective Caml are: `create`, `put` and `get`.

```
# type c_int_array
  external cia_create : int → c_int_array = "create"
  external cia_get : int → c_int_array → int = "get"
  external cia_put : int→ int → c_int_array → unit = "put" ;;
```

We can now manipulate our new data structure from Objective Caml:

```
# let tbl = cia_create 10 and tbl2 = cia_create 10
  in for i=0 to 9 do cia_put i (i*2) tbl done ;
     for i=0 to 9 do print_int (cia_get i tbl) ; print_string " " done ;
     print_newline () ;
```

```
    for i=0 to 9 do cia_put (9-i) (cia_get i tbl) tbl2 done ;
    for i=0 to 9 do print_int (cia_get i tbl2) ; print_string " " done ;;
The external function ‘create’ is not available
```

We now force a garbage collection to check that the finalization function is called:

```
# Gc.full_major () ;;
- : unit = ()
```

In addition to freeing C heap blocks, finalization functions can also be used to close files, terminate processes, etc.

Garbage collection and C parameters and local variables

A C function can trigger a garbage collection, either during an allocation (if the heap is full), or voluntarily by calling `void Garbage_collection_function ()`.

Consider the following example. Can you spot the error?

```
#include <caml/mlvalues.h>
#include <caml/memory.h>

value identity (value x)
{
  Garbage_collection_function() ;
  return x;
}
```

```
# external id : 'a → 'a = "identity" ;;
external id : 'a -> 'a = "identity"
# id [1;2;3;4;5] ;;
The external function ‘identity’ is not available
```

The list passed as parameter to `id`, hence to the C function `identity`, can be moved or reclaimed by the garbage collector. In the example, we forced a garbage collection, but any allocation in the Objective Caml heap could have triggered a garbage collection as well. The anonymous list passed to `id` was reclaimed by the garbage collector, because it is not reachable from the set of known roots. To avoid this, any C function that allocates anything in the Objective Caml heap must tell the garbage collector about the C function's parameters and local variables of type `value`. This is achieved by using the macros described next.

For parameters, these macros are used within the body of the C function as if they were additional declarations:

`CAMLparam1(v)`	:	for one parameter `v` of type `value`
`CAMLparam2(v1,v2)`	:	for two parameters
...		...
`CAMLparam5(v1,...,v5)`	:	for five parameters
`CAMLparam0 ;`	:	required when there are no `value` parameters.

If the C function has more than five `value` parameters, the first five are declared with the `CAMLparam5` macro, and the remaining parameters with the macros `CAMLxparam1`, ..., `CAMLxparam5`, used as many times as necessary to list all `value` parameters.

`CAMLparam5(v1,...,v5);`		
`CAMLxparam5(v6,...,v10);`		
`CAMLxparam2(v11,v12);`	:	for 12 parameters of type `value`

For local variables, these macros are used instead of normal C declarations of the variables. Local variables of type `value` must also be registered with the garbage collector, using the macros `CAMLlocal1`, ..., `CAMLlocal5`. An array of values is declared with `CAMLlocalN(tbl,n)` where `n` is the number of elements of the array `tbl`. Finally, to return from the C function, we must use the macro `CAMLreturn` instead of C's `return` construct.

Here is the corrected version of the previous example:

```
#include <caml/mlvalues.h>
#include <caml/memory.h>
value identity2 (value x)
{
  CAMLparam1(x) ;
  Garbage_collection_function() ;
  CAMLreturn x;
}
```

```
# external id : 'a -> 'a = "identity2" ;;
external id : 'a -> 'a = "identity2"
# let a = id [1;2;3;4;5] ;;
The external function 'identity2' is not available
```

We now obtain the expected result.

Calling an Objective Caml closure from C

To apply a closure (*i.e.* an Objective Caml function value) to one or several arguments from C, we can use the functions declared in the header file `callback.h`.

`callback(f,v)`	:	apply the closure `f` to the argument `v`,
`callback2(f,v1,v2)`	:	same, to two arguments,
`callback3(f,v1,v2,v3)`	:	same, to three arguments,
`callbackN(f,n,tbl)`	:	same, to `n` arguments stored in the array `tbl`.

All these functions return a `value`, which is the result of the application.

Registering Objective Caml functions with C

The `callback` functions require the Objective Caml function to be applied as a closure, that is, as a value that was passed as an argument to the C function. We can also register a closure from Objective Caml, giving it a name, then later refer to the closure by its name in a C function.

The function `register` from module `Callback` associates a name (of type `string`) with a closure or with any other Objective Caml value (of any type, that is, *'a*). This closure or value can be recovered from C using the C function `caml_named_value`, which takes a character string as argument and returns a pointer to the closure or value associated with that name, if it exists, or the null pointer otherwise.

An example is in order:

```
# let plus x y = x + y ;;
val plus : int -> int -> int = <fun>
# Callback.register "plus3_ocaml" (plus 3);;
- : unit = ()
```

```
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/callback.h>

value plus3_C (value v)
{
  CAMLparam1(v);
  CAMLlocal1(f);
  f = *(caml_named_value("plus3_ocaml")) ;
  CAMLreturn callback(f,v) ;
}
```

```
# external plusC : int → int = "plus3_C" ;;
external plusC : int -> int = "plus3_C"
# plusC 1 ;;
The external function ‘plus3_C’ is not available
# Callback.register "plus3_ocaml" (plus 5);;
- : unit = ()
# plusC 1 ;;
The external function ‘plus3_C’ is not available
```

Do not confuse the declaration of a C function with **`external`** and the registration of an Objective Caml closure with the function `register`. In the former case, the

declaration is static, the correspondence between the two names is established at link time. In the latter case, the binding is dynamic: the correspondence between the name and the closure is performed at run time. In particular, the name–closure binding can be modified dynamically by registering a different closure with the same name, thus modifying the behavior of C functions using that name.

Exception handling in C and in Objective Caml

Different languages have different mechanisms for raising and handling exceptions: C relies on `setjmp` and `longjmp`, while Objective Caml has built-in constructs for exceptions (**`try ... with`**, **`raise`**). Of course, these mechanisms are not compatible: they do not keep the same information when setting up a handler. It is extremely hard to safely implement the nesting of exception handlers of different kinds, while ensuring that an exception correctly "jumps over" handlers. For this reason, only Objective Caml exceptions can be raised and handled from C; `setjmp` and `longjmp` in C cannot be caught from Objective Caml, and must not be used to skip over Objective Caml code.

All functions and macros introduced in this section are defined in the header file `fail.h`.

Raising a predefined exception

From a C function, it is easy to raise one of the exceptions `Failure`, `Invalid_argument` or `Not_found` from the `Pervasives` module: just use the following functions.

`failwith(s)`	:	raise the exception `Failure(s)`
`invalid_argument(s)`	:	raise the exception `Invalid_argument(s)`
`raise_not_found()`	:	raise the exception `Not_found`

In the first two cases, `s` is a C string (`char *`) that ends up as the argument to the exception raised.

Raising a user-defined exception

A registration mechanism similar to that for closures enables user-defined exceptions to be raised from C. We must first register the exception using the `Callback` module's `register_exception` function. Then, from C, we retrieve the exception identifier using the `caml_named_value` function (see page 344). Finally, we raise the exception, using one of the following functions:

`raise_constant(e)`	raise the exception `e` with no argument,
`raise_with_arg(e,v)`	raise the exception `e` with the value `v` as argument,
`raise_with_string(e,s)`	same, but the argument is taken from the C string `s`.

Here is an example C function that raises an Objective Caml exception:

```
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/fail.h>

value divide (value v1,value v2)
{
  CAMLparam2(v1,v2);
  if (Long_val(v2) == 0)
    raise_with_arg(*caml_named_value("divzero"),v1) ;
  CAMLreturn Val_long(Long_val(v1)/Long_val(v2)) ;
}
```

And here is an Objective Caml transcript showing the use of that C function:

```
# external divide : int → int → int = "divide" ;;
external divide : int -> int -> int = "divide"
# exception Division_zero of int ;;
exception Division_zero of int
# Callback.register_exception "divzero" (Division_zero 0) ;;
- : unit = ()
# divide 20 4 ;;
The external function ‘divide’ is not available
# divide 22 0 ;;
The external function ‘divide’ is not available
```

Catching an exception

In a C function, we cannot catch an exception raised from another C function. However, we can catch Objective Caml exceptions arising from the application of an Objective Caml function (callback). This is achieved via the functions `callback_exn`, `callback2_exn`, `callback3_exn` and `callbackN_exn`, which are similar to the standard `callback` functions, except that if the callback raises an exception, this exception is caught and returned as the result of the callback. The result value of the `callback_exn` functions must be tested with `Is_exception_result(v)`; this predicate returns “true” if the result value represents an uncaught exception, and “false” otherwise. The macro `Extract_exception(v)` returns the exception value contained in an exceptional result value.

The C function `divide_print` below calls the Objective Caml function `divide` using `callback2_exn`, and checks whether the result is an exception. If so, it prints a message and raises the exception again; otherwise it prints the result.

```
#include <stdio.h>
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/callback.h>
#include <caml/fail.h>

value divide_print (value v1,value v2)
{
  CAMLparam2(v1,v2) ;
  CAMLlocal3(div,dbz,res) ;
  div = * caml_named_value("divide") ;
  dbz = * caml_named_value("div_by_0") ;
  res = callback2_exn (div,v1,v2) ;
  if (Is_exception_result(res))
    {
      value exn=Extract_exception(res);
      if (Field(exn,0)==dbz)  printf("division by 0\n") ;
      else printf("other exception\n");
      fflush(stdout);
      if (Wosize_val(exn)==1)  raise_constant(Field(exn,0)) ;
      else raise_with_arg(Field(exn,0),Field(exn,1)) ;
    }
  printf("result = %d\n",Long_val(res)) ;
  fflush(stdout) ;
  CAMLreturn Val_unit ;
}
```

```
# Callback.register "divide" (/) ;;
- : unit = ()
# Callback.register_exception "div_by_0" Division_by_zero ;;
- : unit = ()
# external divide_print : int → int → unit = "divide_print" ;;
external divide_print : int -> int -> unit = "divide_print"
# divide_print 42 3 ;;
The external function ‘divide_print’ is not available
# divide_print 21 0 ;;
The external function ‘divide_print’ is not available
```

As the examples above show, it is possible to raise an exception from C and catch it in Objective Caml, and also to raise an exception from Objective Caml and catch it in C. However, a C program cannot by itself raise and catch an Objective Caml exception.

Main program in C

Until now, the entry point of our programs was in Objective Caml; the program could then call C functions. Nothing prevents us from writing the entry point in C, and having the C code call Objective Caml functions when desired. To do this, the program must define the usual C `main` function. This function will then initialize the Objective Caml runtime system by calling the function `caml_main(char **)`, which takes as an argument the array of command-line arguments that corresponds to the `Sys.argv` array in Objective Caml. Control is then passed to the Objective Caml code using callbacks (see page 343).

Linking Objective Caml code with C

The Objective Caml compiler can output C object files (with extension `.o`) instead of Objective Caml object files (with extension `.cmo` or `.cmx`). All we need to do is set the `-output-obj` compiler flag.

```
ocamlc -output-obj files.ml
ocamlopt -output-obj.cmxa files.ml
```

From the Objective Caml source files, an object file with default name `camlprog.o` is produced.

The final executable is obtained by linking, using the C compiler, and adding the library `-lcamlrun` if the Objective Caml code was compiled to bytecode, or the library `-lasmrun` if it was compiled to native code.

```
cc camlprog.o filesC.o -lcamlrun
cc camlprog.o filesC.o -lasmrun
```

Calling Objective Caml functions from the C program is performed as described previously, via the `callback` functions. The only difference is that the initialization of the Objective Caml runtime system is performed via the function `caml_startup` instead of `caml_main`.

Exercises

Polymorphic Printing Function

We wish to define a printing function `print` with type *'a -> unit* able to print any Objective Caml value. To this end, we extend and improve the `inspect` function.

1. In C, write the function `print_ws` which prints Objective Caml as follows:

- immediate values: as C integers;
- strings: between quotes;
- floats: as usual;
- arrays of floats: between `[| |]`
- closures: as `< code, env >`
- everything else: as a tuple, between `( )`

The function should handle structured types recursively.

2. To avoid looping on circular values, and to display sharing properly, modify this function to keep track of the addresses of heap blocks it has already seen. If an address appears several times, name it when it is first printed (`v = name`), and just print the name when this address is encountered again.
   (a) Define a data structure to record the addresses, determine when they occur several times, and associate a name with each address.
   (b) Traverse the value once first to determine all the addresses it contains and record them in the data structure.
   (c) The second traversal prints the value while naming addresses at their first occurrences.
   (d) Define the function `print` combining both traversals.

Matrix Product

1. Define an abstract type *`float_matrix`* for matrices of floating-point numbers.
2. Define a C type for these matrices.
3. Write a C function to convert values of type *`float array array`* to values of type *`float_matrix`*.
4. Write a C function performing the reverse conversion.
5. Add the C functions computing the sum and the product of these matrices.
6. Interface them with Objective Caml and use them.

Counting Words: Main Program in C

The Unix command `wc` counts the number of characters, words and lines in a file. The goal of this exercise is to implement this command, while counting repeated words only once.

1. Write the program `wc` in C. This program will simply count words, lines and characters in the file whose name is passed on the command line.
2. Write in Objective Caml a function `add_word` that uses a hash table to record how many times the function was invoked with the same character string as argument.
3. Write two functions `num_repeated_words` and `num_unique_words` counting respectively the number of word repetitions and the number of unique words, as determined from the hash table built by `add_word`.

4. Register the three previous functions so that they can be called from a C program.
5. Rewrite the main function of the `wc` program so that it prints the number of unique words instead of the number of words.
6. Write the `main` function and the commands required to compile this program as an Objective Caml program.
7. Write the `main` function and the commands required to compile this program as a C program.

Summary

This chapter introduced the interface between the Objective Caml language and the C language. This interface allows C functions to operate on Objective Caml values. Using abstract Objective Caml types, the converse is also possible. An important feature of this interface is the ability to use the Objective Caml garbage collector to perform automatic reclamation of values created in C. This interface supports the combination, in the same program, of components developed in the two languages. Finally, Objective Caml exceptions can be raised and (with some limitations) handled from C.

To Learn More

For a better understanding of the C language, especially argument passing and data representations, the book *C: a reference manual* [HS94] is highly recommended.

Concerning exceptions and garbage collection, several works add these missing features to C. The technical report [Rob89] describes an implementation of exceptions in C, based on open macros and on the `setjmp` and `longjmp` functions from the C library. Hans Boehm distributes a conservative collector with ambiguous roots that can be added (as a library) to any C program:

リンク: http://www.hpl.hp.com/personal/Hans_Boehm/gc/

Concerning interoperability between Objective Caml and C, the tools described in this chapter are rather low-level and difficult to use. However, they give the programmer full control on copying or sharing of data structures between the two languages. A higher-level tool called `CamlIDL` is available; it automatically generates the Objective Caml "stubs" (encapsulation functions) for calling C functions and converting data types. The C types and functions are described in a language called IDL (Interface Definition Language), similar to a subset of C++ and C. This description is then passed through the `CamlIDL` compiler, which generates the corresponding `.mli`, `.ml` and `.c` files. This tool is distributed from the following page:

リンク: http://caml.inria.fr/camlidl/

Other interfaces exist between Objective Caml and languages other than C. They are available on the "Caml hump" page:

リンク: http://caml.inria.fr/humps/index.html

They include several versions of interfaces with Fortran, and also an Objective Caml bytecode interpreter written in Java.

Finally, interoperability between Objective Caml and other languages can also be achieved via data exchanges between separate programs, possibly over the network. This approach is described in the chapter on distributed programming (see chapter 20).

13

Applications

この章では、３章で説明しなかったプログラミングの概念を使って、２つのアプリケーションを作ってみます。

最初に Awi(Application Window Interface) と呼ばれるグラフィックコンポーネントライブラリを作り、それを使って日本円からユーロへの換算ソフトを作ります。コンポーネントライブラリはユーザーの入力に反応してイベントハンドラを呼び出します。アルゴリズムとしては簡単なアプリケーションですが、コンポーネント間通信の仕組みとしてクロージャを使う利点が分かります。実際、様々なイベントハンドラが、環境を通して値を共有しています。Awi の構造を理解するには、基となる Graphics ライブラリを理解しておくのがいいでしょう。（5 章, 117 ページ参照）.

２つ目は、有向グラフにおける最小コスト経路を探索するアプリケーションです。これには、起点に繋がっている全てのノードについて最小コスト経路を計算するダイクストラ法を使います。weak pointers (see page 267) の表によるキャッシュ機能を使って探索を高速化します。GC はこの表の要素をいつでも解放することができますが、表の要素は必要に応じて再計算されます。グラフを表示する部分では、始点と終点を指定するために Awi ライブラリに含まれる簡単なボタンを使います。その後、キャッシュを使った時と使わない時のアルゴリズムの実行効率を比較します。簡単に計測できるように、グラフ構造、始点、終点の情報が含まれるファイルを用意し、それをプログラムの引数に渡します。最後に、探索プログラムに簡単なグラフィカルインターフェースを付け加えます。

グラフィカルインターフェースの構築

Graphics ライブラリのように十分強力とは言えないツールしか使えない場合、プログラムにグラフィカルなインターフェースを実装することは退屈な作業です。プログラムの使いやすさの一部はそのインターフェースによります。グラフィカルインターフェースを作る作業を簡単にするために、まず Graphics の上位に位置する Awi と呼ばれる新

しいライブラリを作ることから始めましょう。私達はこの単純なモジュールを使ってアプリケーションのインターフェースを作る事にします。

このグラフィカルインターフェースは**コンポーネント**を扱います。コンポーネントとはメインウィンドウ内のある範囲であり、グラフィカルなものを表示したり送られたイベントを処理したりします。コンポーネントには 2 種類あり、確認ボタンやテキスト入力フィールドなどの単純なコンポーネントと、他のコンポーネントを含むことができる**コンテナ**に分かれます。1 つのコンポーネントはただ 1 つのコンテナに所属することができます。このため、アプリケーションのインターフェースはメインコンテナ（グラフィックスウィンドウ）に対応するルート要素を持つ木として構築されます。節もコンテナであり、葉は単純なコンポーネントまたは空のコンテナです。この木のような構造はユーザーとのやりとりから生じるイベントを伝播させるのに適しています。あるコンテナがイベントを受け取ると、コンテナは自分の子要素がイベントを処理できるなら子要素にイベントを送信し、処理できないなら自分で処理します。

コンポーネントはこのライブラリに欠くことのできない要素です。私達はコンポーネントを、サイズ、グラフィックコンテキスト、親/子のコンポーネント、自分を描画する関数、イベントを処理する関数を含むレコードとして定義します。コンテナは自分が持つコンポーネントを描画させる関数を含みます。*component* 型を定義するために、グラフィックコンテキスト、イベント、初期化オプションのための型を定義することにします。グラフィカルコンテキストは、背景色、前景色や表示位置、用いるフォントのような “グラフィカルスタイル” を保持するのに使います。そして私達はコンポーネントに送られうるいろいろな種類のイベントを定義する必要があります。これらは基になっている `Graphics` ライブラリで定義されているイベントよりも多くの種類からなります。私達はグラフィックコンテキストやコンポーネントを構成しやすいように単純なオプションの仕組みを取り入れます。

一般的なイベント処理ループは `Graphics` ライブラリの入力関数から物理的なイベントを受け取り、これらのイベントの結果他のイベントが引き起こされるべきかどうかを決め、それらをルートコンテナに送ります。ここではコンポーネントとしてテキスト、ボタン、リストボックス、入力範囲、リッチコンポーネントを考えることにします。次では、フランからユーロへの換算プログラムを用いてグラフィカルインターフェースを構築するためのコンポーネントの組み立て方を見ていきます。

グラフィックスコンテキスト、イベント、オプション

グラフィックスコンテキスト、イベント、オプションの初期化/変更関数を定義する前に、まずそれらの基本となる型を定義しましょう。グラフィカルオブジェクトを作る関数をパラメタライズしやすいように、オプション型も定義することにします。

グラフィックスコンテキスト

グラフィックスコンテキストは前景色、背景色、フォント、フォントサイズ、カーソル位置、線の幅を保持することができます。これより、次の型が定義できます。

```
type  g_context = {
    mutable bcol : Graphics.color;
```

```
    mutable fcol : Graphics.color;
    mutable font :  string;
    mutable font_size : int;
    mutable lw : int;
    mutable x : int;
    mutable y : int };;
```

make_default_context 関数はデフォルト値で初期化された新しいグラフィックスコンテキストを生成します[1]。

```
# let default_font = "fixed"
  let default_font_size = 12
  let make_default_context () =
    { bcol = Graphics.white; fcol = Graphics.black;
      font = default_font;
      font_size = default_font_size;
      lw = 1;
      x = 0; y = 0;};;
val default_font : string = "fixed"
val default_font_size : int = 12
val make_default_context : unit -> g_context = <fun>
```

各フィールドにアクセスする関数は、その型の定義を知ることなく値を取得する事ができます。

```
# let get_gc_bcol gc  = gc.bcol
  let get_gc_fcol gc  = gc.fcol
  let get_gc_font  gc  = gc.font
  let get_gc_font_size  gc  = gc.font_size
  let get_gc_lw gc  = gc.lw
  let get_gc_cur gc = (gc.x,gc.y);;
val get_gc_bcol : g_context -> Graphics.color = <fun>
val get_gc_fcol : g_context -> Graphics.color = <fun>
val get_gc_font : g_context -> string = <fun>
val get_gc_font_size : g_context -> int = <fun>
val get_gc_lw : g_context -> int = <fun>
val get_gc_cur : g_context -> int * int = <fun>
```

各フィールドの変更を行う関数は同じ原理で動きます。

```
# let set_gc_bcol gc c = gc.bcol <- c
```

1. フォント名はシステム環境に依存します。

```
  let set_gc_fcol gc c = gc.fcol <- c
  let set_gc_font  gc f = gc.font <- f
  let set_gc_font_size  gc s = gc.font_size <- s
  let set_gc_lw gc i = gc.lw <- i
  let set_gc_cur gc (a,b) = gc.x<- a; gc.y<-b;;
val set_gc_bcol : g_context -> Graphics.color -> unit = <fun>
val set_gc_fcol : g_context -> Graphics.color -> unit = <fun>
val set_gc_font : g_context -> string -> unit = <fun>
val set_gc_font_size : g_context -> int -> unit = <fun>
val set_gc_lw : g_context -> int -> unit = <fun>
val set_gc_cur : g_context -> int * int -> unit = <fun>
```

これにより、私達は *g_context* 型の新しいコンテキストを生成し、各フィールドへの読み書きすることができます。

use_gc 関数はグラフィックコンテキストをグラフィカルウィンドウに適用します。

```
# let use_gc gc =
      Graphics.set_color (get_gc_fcol gc);
      Graphics.set_font (get_gc_font gc);
      Graphics.set_text_size (get_gc_font_size gc);
      Graphics.set_line_width (get_gc_lw gc);
      let (a,b) = get_gc_cur gc in Graphics.moveto a b;;
val use_gc : g_context -> unit = <fun>
```

背景色のような一部のデータは `Graphics` により直接使われないため、use_gc 関数には現れません。

イベント

`Graphics` ライブラリには、マウスのクリック/移動、キーの押下という限られたイベントしか定義されていません。コンポーネントから発生する統合されたイベントの種類を豊富にしたいので、*rich_event* を定義することにします。

```
# type rich_event =
    MouseDown | MouseUp | MouseDrag | MouseMove
  | MouseEnter | MouseExit | Exposure
  | GotFocus | LostFocus | KeyPress | KeyRelease;;
```

このようなイベントを生成するには、以前のイベントを覚えておく必要があります。`MouseDown`、`MouseMove` イベントはマウスのクリック/移動に対応するイベントで、`Graphics` ライブラリにより生成されます。他のマウスイベントは、前回のイベントが `MouseUp` か、物理的な `MouseExit` イベントを扱った最後のコンポーネントによって生成されます。`Exposure` イベントはコンポーネントの再描画要求に対応します。**focus** の概念は、コン

ポーネントがある種のイベントに関連づけられるというものです。典型的には、フォーカスが設定されたコンポーネントへテキストが入力されるという事は、そのコンポーネントだけが KeyPress/KeyRelease イベントを処理することができるという意味です。テキスト入力コンポーネントに対する MouseDown イベントは、フォーカスをそのコンポーネントに設定し、これまでフォーカスを持っていたコンポーネントからフォーカスを取り除きます。

これらの新しいイベントは 362 で述べるイベント処理ループの中で生成されます。

オプション

グラフィカルインターフェースには、グラフィカルオブジェクト（コンポーネント/グラフィカルコンテキスト）の生成オプションを記述するためのルールが必要です。ある色を持つグラフィックコンテキストを作りたいなら、現状ではそれをまずデフォルトの値で作成し、その後で色を変更する 2 つの関数を呼ぶ必要があります。もっと複雑なグラフィックオブジェクトの場合、この作業はすぐに退屈になります。私達はコンポーネントライブラリを強化するためにこれらのオプションを拡張したいので、“拡張可能な”直和型が必要となります。そのようなもののうち Objective Caml から提供されている唯一のものが、例外に使われる *exn* 型です。オプションを扱うために *exn* を使うとプログラムの明快さに影響を及ぼすので、この型は本当の例外のためにだけ使うことにします。その代わりに、文字列で表される疑似コンストラクタを使うことで拡張可能な直和型をシミュレートすることにします。オプションの値のために、*opt_val* を定義します。オプションはタプルであり、最初の要素にオプション名、次の要素にその値が入ります。*lopt* 型はこのようなオプションのリストを持つものとします。

```
# type opt_val = Copt of Graphics.color | Sopt of string
               | Iopt of int | Bopt of bool;;
# type lopt = (string * opt_val) list ;;
```

オプションを解読する関数は、オプションのリストとオプション名とデフォルト値を引数として受け取ります。もしオプション名がリストに所属していたら、対応する値を返します。そうでなければデフォルト値を返します。ここでは整数値と真偽値のオプション値を返す関数を示しますが、他の型についても同様です。

```
# exception OptErr;;
exception OptErr
# let theInt lo name default =
    try
      match List.assoc name lo with
        Iopt i → i
      | _  → raise OptErr
    with Not_found → default;;
val theInt : ('a * opt_val) list -> 'a -> int -> int = <fun>
# let theBool lo name default =
    try
```

```
        match List.assoc name lo with
          Bopt b → b
        | _  → raise OptErr
     with Not_found → default;;
val theBool : ('a * opt_val) list -> 'a -> bool -> bool = <fun>
```

これで、次のようにオプションのリストを指定してグラフィックコンテキストを生成する関数を書くことが出来るようになりました。

```
# let set_gc gc lopt =
     set_gc_bcol gc (theColor lopt "Background" (get_gc_bcol gc));
     set_gc_fcol gc (theColor lopt "Foreground" (get_gc_fcol gc));
     set_gc_font gc (theString lopt "Font" (get_gc_font gc));
     set_gc_font_size gc (theInt lopt "FontSize" (get_gc_font_size gc));
     set_gc_lw gc (theInt lopt "LineWidth" (get_gc_lw gc));;
val set_gc : g_context -> (string * opt_val) list -> unit = <fun>
```

これで、オプションの指定順序を気にしなくていいようになりました。

```
# let dc = make_default_context () in
     set_gc dc [ "Foreground", Copt Graphics.blue;
                 "Background", Copt Graphics.yellow];
     dc;;
- : g_context =
{bcol = 16776960; fcol = 255; font = "fixed"; font_size = 12; lw = 1;
 x = 0; y = 0}
```

あいにく部分的に型体系から逃げていますが、これでかなり柔軟なシステムになりました。オプション名は *string* 型なので存在しない名前のオプションを作ることも可能ですが、単にその値は無視されるだけです。

コンポーネントとコンテナ

コンポーネントはこのライブラリの本質的な要素です。私達は、コンポーネントを生成しそれを使って簡単にインターフェースを構築できるようにしたいです。コンポーネントは自分自身を表示し、自分に送られたイベントを認識し、それを処理しなければなりません。コンテナは他のコンポーネントとの間でイベントの受け渡しをする必要があります。コンポーネントはただ 1 つのコンテナにだけ属することができるものとします。

コンポーネントの構築

component 型の値は次のものを持ちます。サイズ（w、h）、メインウィンドウ内での絶対座標（x、y）、描画される時に使われるグラフィックスコンテキスト（gc）、コンテ

ナかどうかを表すフラグ（container）、コンテナに所属している場合はそのコンテナ（parent）、子コンポーネントのリスト（children）、コンポーネントの位置を操作するための 4 つの関数。子コンポーネントをどう配置するか（layout）、コンポーネントがどう表示されるか（display）、与えられた点がコンポーネントの表示範囲に含まれるかどうかを返す関数（mem）、イベントを処理したら true、しなかったら false を返すイベントハンドラ（listener）。listener の引数は rich_status 型で、これは Graphics モジュールからから来る低レベルのイベント情報、キーボードフォーカス、一般フォーカス、最後にイベントを処理したコンポーネントを含んでいます。というわけで、次のような相互再帰的な定義になります。

```
# type  component =
    { mutable info : string;
      mutable x : int;  mutable y : int;
      mutable w :int ; mutable h : int;
      mutable gc : g_context;
      mutable container : bool;
      mutable parent :   component list;
      mutable children :  component list;
      mutable layout_options : lopt;
      mutable layout : component ->  lopt -> unit;
      mutable display : unit  -> unit;
      mutable mem : int * int -> bool;
      mutable listener : rich_status  -> bool }
  and rich_status =
    { re : rich_event;
      stat : Graphics.status;
      mutable key_focus : component;
      mutable gen_focus : component;
      mutable last : component};;
```

コンポーネントのデータフィールドへのアクセスにはこれらの関数を使います。

```
# let get_gc c = c.gc;;
val get_gc : component -> g_context = <fun>
# let is_container c = c.container;;
val is_container : component -> bool = <fun>
```

次の 3 つの関数はコンポーネントのデフォルトの振る舞いを定義しています。与えられたマウスポインタの座標があるコンポーネントの in_rect に適用されると、その座標がコンポーネントで定義された矩形に含まれるかどうかを返します。デフォルトの描画関数（display_rect）はコンポーネントのグラフィックコンテキストに設定された背景色によりコンポーネントの矩形を塗り潰します。デフォルトのレイアウト関数（direct_layout）はコンテナの左上を基準とする相対座標にコンポーネントを配置します。有効なオプション名は"PosY"と"PosX"で、これはコンテナ座標を基準とした相対座標に対応します。

```
# let in_rect  c  (xp,yp)   =
    (xp >= c.x) && (xp < c.x + c.w) && (yp >= c.y) && (yp < c.y + c.h) ;;
val in_rect : component -> int * int -> bool = <fun>
# let display_rect c ()   =
    let gc = get_gc c in
      Graphics.set_color (get_gc_bcol gc);
      Graphics.fill_rect c.x c.y c.w c.h ;;
val display_rect : component -> unit -> unit = <fun>
# let direct_layout c c1 lopt =
    let px = theInt lopt "PosX" 0
    and py = theInt lopt "PosY" 0 in
      c1.x <- c.x + px; c1.y <- c.y + py ;;
val direct_layout : component -> component -> (string * opt_val) list -> unit =
  <fun>
```

これで、パラメータとして幅と高さをとる create_component 関数や前述の関数を使ってコンポーネントを生成することができるようになりました。

```
# let create_component iw ih  =
    let dc =
       {info="Anonymous";
        x=0; y=0; w=iw; h=ih;
        gc = make_default_context() ;
        container = false;
        parent = [] ; children = [] ;
        layout_options = [] ;
        layout = (fun a b → ());
        display = (fun () → ());
        mem = (fun s → false);
        listener = (fun s  → false);}
    in
      dc.layout <- direct_layout dc;
      dc.mem <- in_rect dc;
      dc.display <- display_rect dc;
      dc ;;
val create_component : int -> int -> component = <fun>
```

それでは次のように空のコンポーネントを宣言してみます。

```
# let empty_component = create_component 0 0 ;;
```

この値は、少なくとも 1 つのコンポーネントを含まなければならない値を生成する際のデフォルト値として使われます。（例えば *rich_status*）

子コンポーネントの追加

コンテナにコンポーネントを追加する上で厄介なのは、コンポーネントをコンテナのどの位置に配置するかという事です。layout フィールドはこの配置関数を保持します。この関数は子コンポーネントとオプションリストを受け取り、コンテナ内での子コンポーネントの新しい座標を計算します。配置関数によっては異なるオプションが使われる事もあります。この後 *panel* コンポーネントについて述べる時にいくつかの配置関数を説明します。(368 ページを参照) ここでは、コンポーネントツリー内での表示関数の伝搬、座標の変化、イベントの伝搬の仕組みを簡単に説明します。アクションの伝搬には、リストの全ての要素に対して指定した関数を適用する List.iter 関数を使います。

change_coord 関数は座標の相対的な変化をコンポーネントやその子コンポーネントの座標に適用します。

```
# let rec change_coord  c (dx,dy)   =
     c.x <- c.x + dx; c.y <- c.y + dy;
     List.iter (fun s → change_coord s (dx,dy) ) c.children;;
val change_coord : component -> int * int -> unit = <fun>
```

add_component 関数はコンポーネントを追加できるかどうかをチェックした上で親 (c) と子 (c1) を関連づけます。配置オプションのリストは子コンポーネントにより保持され、親コンポーネントの配置関数が変化した時に再利用されます。この関数に渡されたオプションリストは配置関数により使われます。コンポーネントを追加できない場合は 3 通りあり、それらはそのコンポーネントが既に子になっている場合と、親がコンテナでない場合と、子が親より大きい場合です。

```
# let add_component  c c1 lopt   =
     if c1.parent <> [] then failwith "add_component: already a parent"
     else
       if not (is_container c ) then
         failwith "add_component: not a container"
       else
         if (c1.x + c1.w > c.w) || (c1.y + c1.h > c.h)
         then failwith "add_component: bad position"
         else
           c.layout  c1 lopt;
           c1.layout_options <- lopt;
           List.iter (fun s → change_coord s (c1.x,c1.y)) c1.children;
           c.children <- c1 :: c.children;
           c1.parent <-  [c] ;;
val add_component : component -> component -> lopt -> unit = <fun>
```

ツリーのある場所からのコンポーネントの除去は、次の関数により行われます。親子間の関連付け、子やその全ての子の座標が変更されます。

```
# let remove_component c c1 =
      c.children <- List.filter ((!=) c1)  c.children;
      c1.parent <- List.filter ((!=) c) c1.parent;
      List.iter (fun s → change_coord s (- c1.x, - c1.y)) c1.children;
      c1.x <- 0; c1.y <- 0;;
val remove_component : component -> component -> unit = <fun>
```

コンテナの配置関数が変化するときの処理は、そのコンテナが子を持っているかどうかに依存します。何も変化しなければ処理は簡単です。そうでなければまずコンテナの全ての子を除去し、コンテナの配置関数を更新した後再び全ての子を以前と同じオプションで追加します。

```
# let set_layout f c =
    if c.children = [] then c.layout <- f
    else
      let ls = c.children in
        List.iter (remove_component c) ls;
        c.layout <- f;
        List.iter (fun s → add_component c s s.layout_options) ls;;
val set_layout : (component -> lopt -> unit) -> component -> unit = <fun>
```

これが配置オプションのリストを保持しておく理由です。オプションリストが新しい配置関数により認識されない場合は、デフォルト値が使われるでしょう。

コンポーネントが表示されると、子コンポーネントに対して表示イベントが伝搬されなければなりません。子同士は重ならないので、表示の順番は重要ではありません。

```
# let rec display  c   =
    c.display ();
    List.iter (fun cx → display cx ) c.children;;
val display : component -> unit = <fun>
```

イベントハンドリング

物理イベント（マウスクリック、キー押下、マウス移動）を処理するには、ユーザの操作に反応して物理ステータス（Graphics.status 型）を返す Graphics.wait_next_event 関数を使います。（132 ページ参照）この物理ステータスは、イベント（*rich_event* 型）、物理ステータス、キーボードフォーカスと一般フォーカスを処理しているコンポーネントや最後にこれらを正常に処理したコンポーネントを含むリッチステータス（*rich_status* 型）を計算するのに使われます。一般フォーカスは全てのイベントを受け付けるフォーカスです。

次にリッチイベントの操作、ステータス情報のコンポーネントへの伝搬、情報の生成、メインイベント処理ループを行う関数を説明します。

ステータス関連の関数

これらの関数はマウスポインタの位置やフォーカスを持っているかどうかを取得します。フォーカス関連の関数にはもう 1 つ、フォーカスを設定したり解除したりするコンポーネントのパラメータが必要です。

```
# let get_event e = e.re;;
# let get_mouse_x e = e.stat.Graphics.mouse_x;;
# let get_mouse_y e = e.stat.Graphics.mouse_y;;
# let get_key e = e.stat.Graphics.key;;

# let has_key_focus e c = e.key_focus == c;;
# let take_key_focus e c = e.key_focus <- c;;
# let lose_key_focus e c = e.key_focus <- empty_component;;
# let has_gen_focus e c = e.gen_focus == c;;
# let take_gen_focus e c = e.gen_focus <- c;;
# let lose_gen_focus e c = e.gen_focus <- empty_component;;
```

イベントの伝搬

リッチイベントはコンポーネントに送信され処理されます。前に述べた表示の仕組みに似て、子コンポーネントは親よりも高い優先度で単純なマウス移動イベントを処理します。コンポーネントがイベントに関連づけられたステータス情報を受け取ると、それを処理できる子がいるかどうかを調べます。いるなら子は `true` を返し、そうでなければ `false` を返します。もしイベントを処理できる子がいなければ、親は自分の `listener` フィールドに設定された関数を使います。

キーボードの働きに対応するステータス情報の伝搬方法はこれと異なります。親コンポーネントはキーボードフォーカスを持っているならイベントを処理し、そうでなければ子に伝搬させます。

あるイベントを処理した結果として生成されるイベントもあります。例えば、コンポーネントがフォーカスを取得したという事は別のコンポーネントがフォーカスを失ったという事です。そのようなイベントは対象となるコンポーネントによりすぐに処理されます。この仕組みは異なるコンポーネント間のマウス移動に伴う entry/exit イベントでも同じです。

send_event 関数は *rich_status* 型の値とコンポーネントを受け取り、イベントが処理されたかどうかを表す真偽値を返します。

```
# let rec send_event rs c =
    match get_event rs with
```

```
      MouseDown | MouseUp | MouseDrag | MouseMove →
       if c.mem(get_mouse_x rs, get_mouse_y rs) then
         if  List.exists (fun sun → send_event rs sun) c.children then true
         else ( if c.listener rs then (rs.last <-c; true) else false )
       else false
    | KeyPress | KeyRelease →
       if has_key_focus rs c then
         ( if c.listener rs then (rs.last<-c; true)
           else false )
       else List.exists (fun sun → send_event rs sun) c.children
    | _  → c.listener rs;;
val send_event : rich_status -> component -> bool = <fun>
```

コンポーネントの階層構造は木であって、循環的なグラフではありません。この事は、send_event 内の再帰が無限ループにならない事を保証します。

イベント生成

物理的なアクション（マウスクリックなど）によって生成されたイベントと過去の履歴と関連するアクション（コンポーネント外にマウスカーソルが動いたなど）により発生したイベントは区別されます。その結果、リッチイベントを生成する 2 つの関数が定義される事になります。

前者を扱う関数は 2 つの物理ステータス情報からリッチイベントを生成します。

```
# let compute_rich_event s0 s1  =
    if s0.Graphics.button <> s1.Graphics.button then
    begin
      if s0.Graphics.button then MouseDown else MouseUp
    end
    else if s1.Graphics.keypressed then KeyPress
    else if (s0.Graphics.mouse_x <> s1.Graphics.mouse_x  ) ||
            (s0.Graphics.mouse_y <> s1.Graphics.mouse_y  ) then
    begin
      if s1.Graphics.button then MouseDrag else MouseMove
    end
    else raise Not_found;;
val compute_rich_event : Graphics.status -> Graphics.status -> rich_event =
  <fun>
```

後者の種類のイベントを生成する関数は最後に起きた 2 つのリッチイベントを引数に取ります。

```
# let send_new_events res0 res1 =
      if res0.key_focus <> res1.key_focus then
```

```
      begin
        ignore(send_event  {res1 with re = LostFocus} res0.key_focus);
        ignore(send_event  {res1 with re = GotFocus} res1.key_focus)
      end;
      if (res0.last <> res1.last) &&
         (( res1.re = MouseMove) || (res1.re = MouseDrag)) then
      begin
        ignore(send_event  {res1 with re = MouseExit} res0.last);
        ignore(send_event  {res1 with re = MouseEnter} res1.last )
      end;;
val send_new_events : rich_status -> rich_status -> unit = <fun>
```

rich_event 型の初期値を定義します。これはイベントループの履歴を初期化するのに使われます。

```
# let initial_re =
    { re = Exposure;
      stat = { Graphics.mouse_x=0; Graphics.mouse_y=0;
               Graphics.key = ' ';
               Graphics.button = false;
               Graphics.keypressed = false };
      key_focus = empty_component;
      gen_focus = empty_component;
      last = empty_component } ;;
```

イベントループ

イベントループはあるコンポーネントとの一連のやりとりを管理します。そのコンポーネントは通常、インターフェースを構成する全てのコンポーネントの先祖です。インターフェースが各物理イベントが処理されるごとに再表示されるべきかどうか（b_disp）、マウス移動を処理するかどうか（b_motion）を表す真偽値が渡されます。最後の引数（c）はコンポーネントツリーの根っこです。

```
# let loop b_disp b_motion  c =
    let res0 = ref initial_re  in
    try
      display c;
      while true do
        let lev = [Graphics.Button_down; Graphics.Button_up;
                   Graphics.Key_pressed] in
        let flev = if b_motion then (Graphics.Mouse_motion) :: lev
                   else lev in
        let s = Graphics.wait_next_event flev
        in
```

```
          let res1 = {!res0 with stat = s} in
           try
              let res2 = {res1 with
                          re = compute_rich_event !res0.stat res1.stat} in
                ignore(send_event res2 c);
                send_new_events !res0 res2;
                res0 := res2;
               if b_disp then display c
           with Not_found → ()
     done
    with e → raise e;;
val loop : bool -> bool -> component -> unit = <fun>
```

このループから抜けるのはイベントハンドラのどれかが例外を投げた時だけです。

テスト関数

マウス/キーボードイベントに対応するステータス情報を手動で生成するための 2 つの関数を定義します。

```
# let make_click e x y =
    {re = e;
     stat = {Graphics.mouse_x=x; Graphics.mouse_y=y;
             Graphics.key = ' '; Graphics.button = false;
             Graphics.keypressed = false};
     key_focus = empty_component;
     gen_focus = empty_component;
     last = empty_component}

  let make_key e ch c =
    {re = e;
     stat = {Graphics.mouse_x=0; Graphics.mouse_y=0;
             Graphics.key = c; Graphics.button = false;
             Graphics.keypressed = true};
     key_focus = empty_component;
     gen_focus = empty_component;
     last = empty_component};;
val make_click : rich_event -> int -> int -> rich_status = <fun>
val make_key : rich_event -> 'a -> char -> rich_status = <fun>
```

これでマウスイベントをコンポーネントに送るシミュレートができるようになりました。

コンポーネントの定義

座標変化、イベント伝搬などいろいろな表示の仕組みが整いました。残ったのは、便利で使いやすいコンポーネントを定義する事です。コンポーネントを次の 3 つのカテゴリに分ける事にします。

- テキストを表示するだけのような、イベントを処理しない単純なコンポーネント
- テキスト入力のような、イベントを処理する単純なコンポーネント
- コンテナとその様々なレイアウト方法

値はコンポーネント間やアプリケーション/コンポーネント間で共有データを変更する事により受け渡しされます。共有は、変更されるべきデータを環境に持つクロージャにより実現されます。さらにコンポーネントの振る舞いはイベント処理の結果によって変えられるので、コンポーネントは内部状態をイベントハンドラ内のクロージャに含める必要があります。例えば、テキスト入力フィールドは入力されているテキストにアクセスします。このため、コンポーネントは次の様式に従って書くことにします。

- コンポーネントの内部状態を表す型を定義する。
- 内部状態を操作する関数を定義する。
- 表示するための関数を定義する。与えられた座標がコンポーネント内にあるか判定してイベントを処理する。
- コンポーネントを生成する関数を定義する。これによりクロージャと内部状態が関連づけられる。
- イベントの到着をシミュレートすることでコンポーネントのテストを行う。

これらのコンポーネントの実装を説明します。

- 単純なテキスト (`label`);
- 単純なコンテナ (`panel`);
- 単純なボタン (`button`);
- いくつかのテキストから選択する (`choice`);
- テキスト入力 (`textfield`);
- リッチコンポーネント (`border`).

ラベルコンポーネント

ラベルと呼ばれる最も単純なコンポーネントです。これは文字列を画面に表示します。イベントは処理しません。表示関数と生成関数を書くことから始めましょう。

表示には前景色と背景色とフォントを考慮しなければなりません。コンポーネントが占める画面領域を消し、前景色とカーソル位置を設定するのはすのは `display_init` 関数の仕事です。`display_label` 関数は `display_init` を呼んだ後すぐに引数として渡された文字列を表示します。

```
# let display_init c =
```

```
    Graphics.set_color (get_gc_bcol (get_gc c)); display_rect  c ();
    let gc= get_gc c in
      use_gc gc;
      let (a,b) = get_gc_cur gc in
        Graphics.moveto (c.x+a) (c.y+b)
   let display_label s c () =
      display_init c; Graphics.draw_string s;;
val display_init : component -> unit = <fun>
val display_label : string -> component -> unit -> unit = <fun>
```

このコンポーネントはとても単純なので、内部状態を作る必要はありません。生成関数から渡されて表示される文字列は、display_label 関数だけが知っています。

```
# let create_label s lopt =
    let gc = make_default_context () in   set_gc gc lopt; use_gc gc;
      let (w,h) = Graphics.text_size s in
        let u = create_component w h  in
          u.mem <- (fun x → false);  u.display <- display_label s  u;
          u.info <- "Label"; u.gc <- gc;
          u;;
val create_label : string -> (string * opt_val) list -> component = <fun>
```

このコンポーネントの色を変えるには、グラフィックコンテキストを直接操作する必要があります。*label* l1 を表示させた図が 13.1 です。

```
# let courier_bold_24 = Sopt "*courier-bold-r-normal-*24*"
  and courier_bold_18  = Sopt "*courier-bold-r-normal-*18*";;
# let l1 = create_label "Login: " ["Font", courier_bold_24;
                                  "Background", Copt gray1];;
```

図 13.1: *label* の表示

panel コンポーネント、コンテナ、レイアウト

panel はコンテナになる事ができるグラフィカルな領域です。パネルを生成する関数はとても単純で、一般的なコンポーネント生成関数にコンテナかどうかを表す真偽値引数を加えたものです。*panel* 内かどうかを判別する関数と表示関数は create_component によりデフォルト関数に初期化されます。

```
# let create_panel b  w h  lopt =
    let u = create_component w h   in
      u.container <- b;
      u.info <- if b then "Panel container" else "Panel";
      let gc = make_default_context () in set_gc gc lopt; u.gc <- gc;
      u;;
val create_panel : bool -> int -> int -> (string * opt_val) list -> component =
  <fun>
```

子コンポーネントの配置はコンテナの巧妙な部分です。2つのレイアウト関数 center_layout、grid_layout を定義します。最初の center_layout はコンポーネントをコンテナの中心に配置します。

```
# let center_layout c c1 lopt =
      c1.x <- c.x + ((c.w -c1.w) /2); c1.y <- c.y + ((c.h -c1.h) /2);;
val center_layout : component -> component -> 'a -> unit = <fun>
```

次の grid_layout はコンテナを同じサイズの格子いくつかに分割します。レイアウトオプションは列数"Col"と行数"Row"です。

```
# let grid_layout (a, b)  c c1 lopt =
    let px = theInt lopt "Col" 0
    and py = theInt lopt "Row" 0 in
     if (px >= 0) && (px < a) && ( py >=0) && (py < b) then
       let lw = c.w /a
       and lh = c.h /b in
            if (c1.w > lw) || (c1.h > lh) then
              failwith "grid_placement: too big component"
            else
              c1.x <- c.x + px * lw + (lw - c1.w)/2;
              c1.y <- c.y + py * lh + (lh - c1.h)/2;
       else  failwith "grid_placement: bad position";;
val grid_layout :
  int * int -> component -> component -> (string * opt_val) list -> unit =
  <fun>
```

もちろんもっとたくさん定義することもできます。コンテナの set_layout 関数によりレイアウト関数をカスタマイズすることができます。図 13.2 はコンテナとして定義された *panel* で、2 つのラベルが追加されています。対応するするプログラムを次に示します。

```
# let l2 = create_label "Passwd: " ["Font", courier_bold_24;
                                    "Background", Copt gray1] ;;
```

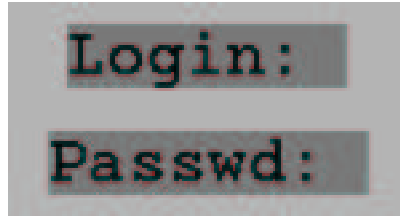


図 13.2: A *panel* component.

```
# let p1 = create_panel true 150 80 ["Background", Copt gray2] ;;
# set_layout  (grid_layout (1,2) p1) p1;;
# add_component p1 l1 ["Row", Iopt 1];;
# add_component p1 l2 ["Row", Iopt 0];;
```

コンポーネントは、インターフェースに統合できるように少なくとも 1 つの親を持っていないといけません。また、Graphics ライブラリは 1 つのウィンドウしか扱えないのでコンテナである本源的なウィンドウを定義しなければなりません。

```
# let open_main_window w h =
     Graphics.close_graph();
     Graphics.open_graph (" "^(string_of_int w)^"x"^(string_of_int h));
     let u = create_component  w h in
        u.container <- true;
        u.info <- "Main Window";
        u;;
val open_main_window : int -> int -> component = <fun>
```

ボタンコンポーネント

ボタンは文字列を表示すると共にそこで起きたマウスクリックに反応できるコンポーネントです。この振る舞いを実現するために、ボタンコンポーネントは文字列とマウスのイベントハンドラを持つ *button_state* 型の状態を保持します。

```
# type button_state =
   { txt : string; mutable action :  button_state → unit } ;;
```

create_bs 関数はこの状態を生成します。set_bs_action 関数はボタンが押された時に実行されるアクション関数を設定し、get_bs_text 関数はボタンの文字列を取得します。

```
# let create_bs s = {txt = s; action = fun x → ()}
  let set_bs_action bs f = bs.action <- f
```

```
  let get_bs_text bs = bs.txt;;
val create_bs : string -> button_state = <fun>
val set_bs_action : button_state -> (button_state -> unit) -> unit = <fun>
val get_bs_text : button_state -> string = <fun>
```

表示関数はラベルのものと似ていますが、表示文字列をボタンが保持する状態から取得する点が違います。イベント処理関数はマウスボタンが押された時、デフォルトでは set_bs_action により設定されたアクション関数を呼び出します。

```
# let display_button  c bs  () =
     display_init c;  Graphics.draw_string (get_bs_text bs)
  let listener_button c bs  e = match get_event e with
    MouseDown → bs.action bs; c.display (); true
  | _ → false;;
val display_button : component -> button_state -> unit -> unit = <fun>
val listener_button : component -> button_state -> rich_status -> bool =
  <fun>
```

これでボタン生成関数に必要な関数は全て定義されました。

```
# let create_button s lopt     =
    let bs = create_bs s in
      let gc = make_default_context () in
        set_gc gc lopt; use_gc gc;
        let w,h = Graphics.text_size (get_bs_text bs) in
          let u = create_component w h   in
            u.display <- display_button u bs;
            u.listener <- listener_button u bs;
            u.info <- "Button";
            u.gc <- gc;
            u,bs;;
val create_button :
  string -> (string * opt_val) list -> component * button_state = <fun>
```

この関数が返す値は、前半の要素が生成されたボタンコンポーネントで、後半の要素がボタンの内部状態であるようなタプルです。

図 13.3 はボタンが追加されたパネルです。

ボタンに表示されている文字列を標準出力に出力するアクション関数がボタンに関連づけられています。

```
# let b,bs = create_button "Validation" ["Font", courier_bold_24;
                                          "Background", Copt gray1];;
```

図 13.3: ボタンの表示とマウスクリックへの反応

```
# let p2 = create_panel true 150 60 ["Background", Copt gray2];;
# set_bs_action bs (fun bs → print_string ( (get_bs_text bs)^ "...");
                              print_newline());;
# set_layout  (center_layout p2) p2;;
# add_component p2 b  [];;
```

ラベルと違って、ボタンコンポーネントはマウスクリックにどう反応すべきか知っています。これをテストするには "マウスクリック" イベントをパネル p2 の中心にあるボタンに送ります。これによりボタンに関連づけられたアクション関数が実行されます。

```
# send_event (make_click MouseDown 75 30) p2;;
- : bool = false
```

そしてイベントが実際に処理された事を示す `true` が返ってきます。

choice コンポーネント

choice コンポーネントはマウスクリックにより選択肢の中から 1 つのだけ選ぶことができるコンポーネントです。必ず選択中の選択肢があります。ほかの選択肢に対してマウスクリックすると、現在の選択は変更され、アクション関数が実行されます。ボタンで使ったのと同じテクニックがここでも使えます。選択肢リストで必要な内部状態を定義する事から始めます。

```
# type choice_state =
    { mutable ind : int; values : string array; mutable sep : int;
      mutable height : int; mutable action : choice_state → unit } ;;
```

インデックス `ind` は選択肢 `values` の中で強調表示されるべき文字列を表します。`sep` は選択肢間の間隔、`height` は選択肢の高さをピクセル単位で指定します。アクション関数は *choice_state* 型の値を受け取り、index を使った処理をすることができます。

ここで、状態を生成するための関数とアクション関数を設定する関数を定義します。

```
# let create_cs sa = {ind = 0; values = sa; sep = 2;
```

```
                     height = 1; action = fun x → ()}
  let set_cs_action cs f = cs.action <- f
  let get_cs_text cs = cs.values.(cs.ind);;
val create_cs : string array -> choice_state = <fun>
val set_cs_action : choice_state -> (choice_state -> unit) -> unit = <fun>
val get_cs_text : choice_state -> string = <fun>
```

表示関数は全ての選択肢を表示し、現在選択中のものを反転表示して強調します。イベント処理関数はマウスボタンが離された時にアクション関数を実行します。

```
# let display_choice c cs  () =
     display_init c;
     let (x,y) = Graphics.current_point()
     and nb = Array.length cs.values in
       for i = 0 to nb-1 do
         Graphics.moveto x (y + i*(cs.height+ cs.sep));
         Graphics.draw_string cs.values.(i)
       done;
       Graphics.set_color (get_gc_fcol (get_gc c));
       Graphics.fill_rect x (y+ cs.ind*(cs.height+ cs.sep)) c.w cs.height;
       Graphics.set_color (get_gc_bcol (get_gc c));
       Graphics.moveto x  (y + cs.ind*(cs.height + cs.sep));
       Graphics.draw_string cs.values.(cs.ind) ;;
val display_choice : component -> choice_state -> unit -> unit = <fun>

# let listener_choice c cs e = match e.re with
    MouseUp →
      let x = e.stat.Graphics.mouse_x
      and y = e.stat.Graphics.mouse_y in
        let cy = c.y in
          let i = (y - cy) / ( cs.height + cs.sep) in
            cs.ind <- i; c.display ();
            cs.action cs; true
  |   _  → false ;;
val listener_choice : component -> choice_state -> rich_status -> bool =
  <fun>
```

choice コンポーネントの生成関数は選択肢の文字列リストとオプションリストを受け取りコンポーネント自身と内部状態を返します。

```
# let create_choice lc lopt  =
    let sa =  (Array.of_list (List.rev lc)) in
    let cs = create_cs sa in
    let gc = make_default_context () in
```

```
      set_gc gc lopt;  use_gc gc;
        let awh = Array.map (Graphics.text_size) cs.values in
        let w = Array.fold_right (fun (x,y) → max x)  awh 0
        and h = Array.fold_right (fun (x,y) → max y)  awh 0 in
        let h1 = (h+cs.sep) * (Array.length sa) + cs.sep  in
        cs.height <- h;
        let u = create_component w h1   in
          u.display <- display_choice u cs;
          u.listener <- listener_choice u cs ;
          u.info <- "Choice "^ (string_of_int (Array.length cs.values));
          u.gc <- gc;
          u,cs;;
val create_choice :
  string list -> (string * opt_val) list -> component * choice_state = <fun>
```

図 13.4 に示した 3 つの一連の図は choice コンポーネントが追加されたパネルです。選択された文字列を標準出力に出力するアクション関数が関連づけられています。これらの図はマウスクリックによるもので、次のプログラムによってシミュレートされています。

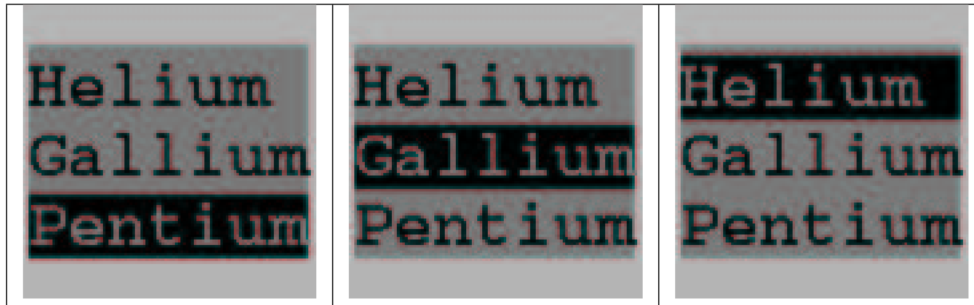


図 13.4: 選択肢リストの表示と選択

```
# let c,cs = create_choice ["Helium"; "Gallium"; "Pentium"]
                           ["Font", courier_bold_24;
                            "Background", Copt gray1];;
# let p3 = create_panel true 110 110  ["Background", Copt gray2];;
# set_cs_action cs (fun cs → print_string ( (get_cs_text cs)^"...");
                             print_newline());;
# set_layout  (center_layout p3) p3;;
# add_component p3 c  [];;
```

さまざまなイベントを送ることでコンポーネントを直接テストする事もできます。次の例は、図 13.4 中央の図ように選択を変更します。

```
# send_event (make_click MouseUp 60 55 ) p3;;
```

```
- : bool = false
```

次のイベントを送ると最初の選択肢が選択されます。

```
# send_event (make_click MouseUp 60 90 ) p3;;
- : bool = false
```

textfield コンポーネント

テキスト入力フィールドまたは *textfield* は、文字列を入力することができる領域です。

文字列は左揃えや（電卓のように）右揃えにできます。そして、カーソルは次に文字が挿入されることになる場所を表します。この動作にはもっと複雑な内部状態、すなわち入力された文字列、入力の方向、カーソルの座標、文字がどう表示されるか、アクション関数が必要になります。

```
# type textfield_state =
    { txt : string;
      dir : bool; mutable ind1 : int; mutable ind2 : int; len : int;
      mutable visible_cursor : bool; mutable cursor : char;
      mutable visible_echo : bool; mutable echo : char;
      mutable action : textfield_state → unit } ;;
```

この内部状態を生成するには初期文字列、入力できる文字数、テキストの入力方向を指定します。

```
# let create_tfs txt size dir  =
    let l = String.length txt in
      (if size < l then failwith "create_tfs");
      let ind1 = if dir then 0 else size-1-l
      and ind2 = if dir then l else size-1 in
      let n_txt = (if dir then (txt^(String.make (size-l) ' '))
                            else ((String.make (size-l) ' ')^txt )) in
    {txt = n_txt; dir=dir; ind1 = ind1; ind2 = ind2; len=size;
     visible_cursor  = false;  cursor = ' '; visible_echo =  true; echo = ' ';
     action= fun x →  ()};;
val create_tfs : string -> int -> bool -> textfield_state = <fun>
```

次の関数を使うと表示される文字列などいろいろなフィールドにアクセスする事ができます。

```
# let set_tfs_action tfs f = tfs.action <- f
  let set_tfs_cursor b c tfs =  tfs.visible_cursor <- b; tfs.cursor <- c
```

```
  let set_tfs_echo b c tfs =  tfs.visible_echo <- b; tfs.echo <- c
  let get_tfs_text tfs =
    if tfs.dir then String.sub tfs.txt tfs.ind1 (tfs.ind2 - tfs.ind1)
    else String.sub tfs.txt (tfs.ind1+1) (tfs.ind2 - tfs.ind1);;
```

set_tfs_text 関数は tf コンポーネントの内部状態 tfs が持つテキストを txt に変更します。

```
# let set_tfs_text tf tfs txt =
      let l = String.length txt in
        if l > tfs.len then failwith "set_tfs_text";
        String.blit  (String.make tfs.len ' ') 0 tfs.txt 0 tfs.len;
        if tfs.dir then (String.blit txt 0 tfs.txt 0 l;
                     tfs.ind2 <- l )
        else   ( String.blit txt 0 tfs.txt (tfs.len -l) l;
                 tfs.ind1 <- tfs.len-l-1 );
        tf.display ();;
val set_tfs_text : component -> textfield_state -> string -> unit = <fun>
```

表示操作に際しては、文字表示のされかた、カーソルがの可視/不可視を考慮に入れなければなりません。display_textfield 関数はカーソルがどこにあるのかを表示する display_cursor 関数を呼び出します。

```
# let display_cursor c tfs =
    if tfs.visible_cursor then
      ( use_gc (get_gc c);
        let (x,y) = Graphics.current_point() in
        let (a,b) = Graphics.text_size " " in
        let shift =  a *  (if tfs.dir then max (min (tfs.len-1) tfs.ind2)  0
                           else tfs.ind2) in
          Graphics.moveto (c.x+x + shift) (c.y+y);
          Graphics.draw_char tfs.cursor);;
val display_cursor : component -> textfield_state -> unit = <fun>
# let display_textfield c tfs  () =
      display_init c;
      let s = String.make tfs.len ' '
      and txt = get_tfs_text tfs in
        let nl = String.length txt in
        if (tfs.ind1 >= 0) && (not tfs.dir) then
           Graphics.draw_string (String.sub s 0 (tfs.ind1+1) );
        if tfs.visible_echo  then (Graphics.draw_string (get_tfs_text tfs))
        else Graphics.draw_string (String.make (String.length txt) tfs.echo);
        if (nl > tfs.ind2) && (tfs.dir)
          then Graphics.draw_string (String.sub s tfs.ind2 (nl-tfs.ind2));
```

```
      display_cursor c tfs;;
val display_textfield : component -> textfield_state -> unit -> unit = <fun>
```

この種のコンポーネントに対するイベントリスナは複雑です。入力方向（左か右）によって、すでに入力された文字列を動かす必要がでてくるかもしれません。フォーカスは入力領域をマウスでクリックすることで取得されます。

```
# let listener_text_field u tfs e =
      match e.re with
        MouseDown → take_key_focus e u ; true
      | KeyPress →
          ( if Char.code (get_key e)  >= 32 then
            begin
              ( if tfs.dir then
                ( ( if tfs.ind2 >= tfs.len then (
                      String.blit tfs.txt 1 tfs.txt 0 (tfs.ind2-1);
                      tfs.ind2 <- tfs.ind2-1) );
                    tfs.txt.[tfs.ind2] <- get_key e;
                    tfs.ind2 <- tfs.ind2 +1 )
                else
                ( String.blit tfs.txt 1 tfs.txt 0 (tfs.ind2);
                  tfs.txt.[tfs.ind2] <- get_key e;
                  if tfs.ind1 >= 0 then tfs.ind1 <- tfs.ind1 -1
                );
              )
            end
            else (
            ( match Char.code (get_key e) with
                13 → tfs.action tfs
              |  9 → lose_key_focus e u
              |  8 → if (tfs.dir && (tfs.ind2 > 0))
                        then tfs.ind2 <- tfs.ind2 -1
                        else if (not tfs.dir) && (tfs.ind1 < tfs.len -1)
                             then tfs.ind1 <- tfs.ind1+1
              | _ → ()
            ))); u.display(); true
      | _ → false;;
val listener_text_field : component -> textfield_state -> rich_status -> bool =
  <fun>
```

テキストフィールドを生成する関数はこれまでと似たようなものです。

```
# let create_text_field  txt size dir lopt  =
    let tfs = create_tfs txt size dir
```

```
    and l = String.length txt in
     let gc = make_default_context () in
      set_gc gc lopt; use_gc gc;
      let (w,h) = Graphics.text_size (tfs.txt) in
        let u = create_component w h   in
          u.display <- display_textfield u tfs;
          u.listener <-  listener_text_field u tfs ;
          u.info <- "TextField"; u.gc <- gc;
          u,tfs;;
val create_text_field :
  string ->
  int -> bool -> (string * opt_val) list -> component * textfield_state =
  <fun>
```

この関数はコンポーネント自身とその内部状態の組を返します。次のコードにより図 13.5 のようなコンポーネントを生成することができます。

```
# let tf1,tfs1 = create_text_field "jack"  8 true ["Font", courier_bold_24];;
# let tf2,tfs2 = create_text_field "koala" 8  false ["Font", courier_bold_24];;
# set_tfs_cursor true '_' tfs1;;
# set_tfs_cursor true '_' tfs2;;
# set_tfs_echo false '*' tfs2;;
# let p4 = create_panel true 140 80 ["Background", Copt gray2];;
# set_layout  (grid_layout (1,2) p4) p4;;
# add_component p4 tf1 ["Row", Iopt 1];;
# add_component p4 tf2 ["Row", Iopt 0];;
```

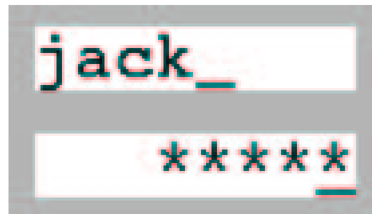


図 13.5: テキスト入力コンポーネント

拡張コンポーネント

これまでに述べたコンポーネントを越えた新しいコンポーネントを構築することもできます。例えば 136 ページにある計算機のように斜めの縁がついたコンポーネントです。あるコンポーネントより大きいパネルを作って、それを何らかの方法で塗り潰した後でコンポーネントを中央に追加すればこの効果を実現することができます。

```
# type border_state =
    {mutable relief : string; mutable line : bool;
     mutable bg2 : Graphics.color; mutable size : int};;
```

生成関数はオプションリストを受け取り内部状態を作ります。

```
# let create_border_state lopt =
    {relief = theString lopt "Relief" "Flat";
     line = theBool lopt "Outlined" false;
     bg2 = theColor lopt "Background2" Graphics.black;
     size = theInt lopt "Border_size" 2};;
val create_border_state : (string * opt_val) list -> border_state = <fun>
```

"Top"、"Bot"、"Flat"オプションを追加することで、図 5.6（130 ページ）のボックスで使われた縁の情報を宣言できます。

```
# let display_border bs c1 c () =
    let x1 = c.x  and y1 = c.y in
    let x2 = x1+c.w-1 and y2 = y1+c.h-1 in
    let ix1 = c1.x and iy1 =  c1.y in
    let ix2 = ix1+c1.w-1 and iy2 = iy1+c1.h-1 in
    let border1 g = Graphics.set_color g;
      Graphics.fill_poly [| (x1,y1);(ix1,iy1);(ix2,iy1);(x2,y1) |] ;
      Graphics.fill_poly [| (x2,y1);(ix2,iy1);(ix2,iy2);(x2,y2) |]
    in
      let border2 g =  Graphics.set_color g;
        Graphics.fill_poly [| (x1,y2);(ix1,iy2);(ix2,iy2);(x2,y2) |] ;
        Graphics.fill_poly [| (x1,y1);(ix1,iy1);(ix1,iy2);(x1,y2) |]
      in
        display_rect c ();
        if bs.line then (Graphics.set_color (get_gc_fcol (get_gc c));
                         draw_rect x1 y1 c.w c.h);
       let b1_col = get_gc_bcol ( get_gc c)
       and b2_col = bs.bg2 in
        match bs.relief with
          "Top" ->  (border1 b1_col; border2 b2_col)
        |  "Bot" -> (border1 b2_col; border2 b1_col)
        |  "Flat" ->  (border1 b1_col; border2 b1_col)
        |  s -> failwith ("display_border: unknown relief: "^s)
  ;;
val display_border : border_state -> component -> component -> unit -> unit =
  <fun>
```

縁を作る関数はコンポーネントとオプションリストを引数に取り、そのコンポーネントを含むパネルを構築します。

```
# let create_border c lopt =
     let bs = create_border_state lopt in
     let p = create_panel true (c.w + 2 * bs.size)
                               (c.h + 2 * bs.size) lopt in
       set_layout (center_layout p) p;
       p.display <- display_border bs c p;
       add_component p c [] ; p;;
val create_border : component -> (string * opt_val) list -> component = <fun>
```

これで、前のテストで宣言したラベルコンポーネントとテキスト入力コンポーネント tf1 にボーダーを持たせたコンポーネントを生成することができるようになりました。図 13.6 のような結果になります。

```
# remove_component   p1 l1;;
# remove_component   p4 tf1;;
# let b1 = create_border l1  [];;
# let b2 = create_border tf1 ["Relief", Sopt "Top";
                              "Background", Copt Graphics.red;
                              "Border_size", Iopt 4];;
# let p5 = create_panel true 140 80 ["Background", Copt gray2];;
# set_layout  (grid_layout (1,2) p5) p5;;
# add_component p5 b1 ["Row", Iopt 1];;
# add_component p5 b2 ["Row", Iopt 0];;
```

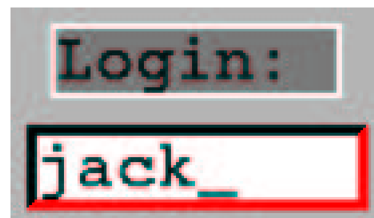


図 13.6: 拡張コンポーネント

Awi ライブラリのセットアップ

ここまでで、このライブラリの重要な部分は完成しました。この章で宣言した全ての型や値は 1 つのファイルにまとめる事ができます。[2] このライブラリは 1 つのモジュールを

2. ただしテストするための宣言は除きます。

構成します。このファイルを awi.ml と呼ぶなら awi モジュールになります。ファイル名とモジュール名の対応については 14 章を参照してください。

このファイルをコンパイルすると、コンパイル済みインターフェースファイル awi.cmi とコンパイラによってバイトコード awi.cmo かネイティブコード awi.cmx が生成されます。バイトコンパイラを使うには次のコマンドを入力します。

```
ocamlc -c awi.ml
```

インタラクティブトップレベルでこのライブラリを使うには、#load "awi.cmo";; コマンドでバイトコードをロードする必要があります。これで、このモジュールで宣言された関数を呼び出してコンポーネントを作ったりすることができるようになります。

この関数の返り値は Awi.component です。これについては 14 章に詳しく書いてあります。

Example: A フラン-ユーロ変換器

この新しいライブラリを使ってフラン-ユーロ間の通貨変換器を作ることにしましょう。変換自体は取るに足らないものですが、インターフェースの構築部分を見るとコンポーネント間でどのように通信が行われているかが分かります。また新しい通貨に慣れるまでの間は両方向に変換できるようにしたいです。使うコンポーネントは次の通りです。

- 変換の方向を表す 2 つの選択肢
- 金額を入力したり結果を表示したりするための 2 つのテキストフィールド
- 変換を行うためのボタン
- テキストフィールドの意味を表示するための 2 つのラベル

図 13.7 にこれらのコンポーネントが示されています。

コンポーネント間の通信は共有状態によって実現されています。そのためにフラン（a)、ユーロ（b)、どちらの方向に変換されるか（dir)、変換するための係数（fa、fb）を持つ *state_conv* を定義します。

```
# type state_conv =
    { mutable a:float; mutable b:float; mutable dir : bool;
      fa : float; fb : float } ;;
```

初期状態を次のように定義します。

```
# let e =  6.55957074
  let fe = { a =0.0; b=0.0; dir = true; fa = e; fb = 1./. e};;
```

変換関数は変換方向によって浮動小数点数の結果を返します。

```
# let calculate fe =
    if fe.dir then fe.b <- fe.a /. fe.fa else fe.a <- fe.b /. fe.fb;;
```

```
val calculate : state_conv -> unit = <fun>
```

2 つの選択肢に対するマウスクリックは変換の方向を変えます。選択肢の文字列は"->" と "<-"です。

```
# let action_dir fe cs = match get_cs_text cs with
    "->" → fe.dir <- true
  | "<-" → fe.dir <- false
  | _ → failwith "action_dir";;
val action_dir : state_conv -> choice_state -> unit = <fun>
```

ボタンに関連づけられたアクションは変換を行い、2 つのテキストフィールドのうちどちらかに結果を表示します。このため、アクション関数の引数としてこれらの 2 つのテキストフィールドも渡します。

```
# let action_go fe tf_fr tf_eu tfs_fr tfs_eu  x =
    if fe.dir then
      let r = float_of_string (get_tfs_text tfs_fr) in
        fe.a <- r; calculate fe;
        let sr = Printf.sprintf "%.2f" fe.b in
            set_tfs_text tf_eu tfs_eu  sr
    else
      let r = float_of_string (get_tfs_text tfs_eu) in
        fe.b <- r; calculate fe;
        let sr = Printf.sprintf "%.2f" fe.a in
            set_tfs_text tf_fr tfs_fr sr;;
val action_go :
  state_conv ->
  component -> component -> textfield_state -> textfield_state -> 'a -> unit =
  <fun>
```

あとはインターフェースを構築するだけです。次の関数は幅、高さ、変換の状態を受け取って 3 つのアクティブコンポーネントを含んだメインコンテナを返します。

```
# let create_conv w h fe =
  let gray1 = (Graphics.rgb 120 120 120) in
  let m = open_main_window w h
  and  p = create_panel true (w-4) (h-4) []
  and  l1 = create_label "Francs" ["Font", courier_bold_24;
                                   "Background", Copt gray1]
  and l2 = create_label "Euros" ["Font", courier_bold_24;
                                 "Background", Copt gray1]
  and c,cs = create_choice ["->"; "<-"] ["Font", courier_bold_18]
```

```
  and tf1,tfs1 = create_text_field  "0" 10 false ["Font", courier_bold_18]
  and tf2,tfs2 = create_text_field "0" 10 false ["Font", courier_bold_18]
  and b,bs = create_button " Go " ["Font", courier_bold_24]
  in
    let gc = get_gc m in
      set_gc_bcol gc gray1;
      set_layout (grid_layout (3,2) m ) m;
      let tb1 = create_border tf1  []
      and tb2 = create_border tf2  []
      and bc = create_border c  []
      and bb =
        create_border  b
          ["Border_size", Iopt 4; "Relief", Sopt "Bot";
           "Background", Copt gray2; "Background2", Copt Graphics.black]
      in
        set_cs_action cs (action_dir fe);
        set_bs_action bs (action_go fe tf1 tf2 tfs1 tfs2);
        add_component m l1 ["Col",Iopt 0;"Row",Iopt 1];
        add_component m l2 ["Col",Iopt 2;"Row",Iopt 1];
        add_component m bc ["Col",Iopt 1;"Row",Iopt 1];
        add_component m tb1 ["Col",Iopt 0;"Row",Iopt 0];
        add_component m tb2 ["Col",Iopt 2;"Row",Iopt 0];
        add_component m bb  ["Col",Iopt 1;"Row",Iopt 0];
        m,bs,tf1,tf2;;
val create_conv :
  int ->
  int -> state_conv -> component * button_state * component * component =
  <fun>
```

次のコードによりコンテナ m の中でイベント処理ループが始まります。その結果図 13.7 のような画面になります。

```
# let (m,c,t1,t2) = create_conv 420 150 fe ;;
# display m ;;
```

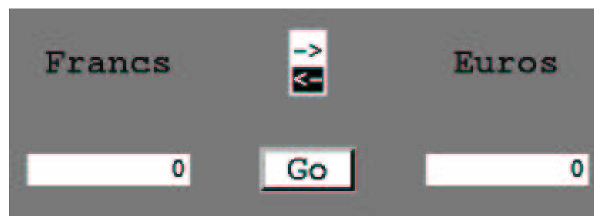


図 13.7: 変換器のウィンドウ

全てのイベント処理クロージャは同じ状態を共有しているので、選択肢がクリックされると表示されている文字列と変換の方向の両方が変わります。

もっと知りたい方は

クロージャによりコンポーネントに対するイベントアクション関数の登録ができます。ですが、既にあるハンドラを “再利用” して追加の振る舞いをさせるようにすることはできません。まったく新しいハンドラを定義する必要があります。16 章では、関数とオブジェクト指向パラダイムの比較をして、ハンドラ拡張の可能性について議論します。

ここまでのプログラムでは多くの構造体が同一の名前を持つフィールドを持っています。（例えば txt）最後に宣言した内容はそれ以前の宣言を隠してしまうので、プログラム内でフィールドに直接アクセスするのは難しいです。型を定義するたびにその型のフィールドへのアクセス関数も定義したのはそのためです。もう一つの解決方法はライブラリをいくつかのモジュールに分けることです。こうするとモジュール名を指定することによりフィールド名を区別することができます。いずれにせよアクセス関数のおかげでこのライブラリの全ての機能を使うことができます。14 章では型の重ね合わせの話題に戻り、抽象データ型の紹介をします。重ね合わせを使うと、環状になってはならないコンポーネントの親子関係のような敏感なフィールドを修正させないことで厳密さを増すことができます。

このライブラリを拡張する方法はいくつもあります。

コンポーネントのデザインに関する基準の 1 つは、新しいものが書けなければならないということです。新しい mem、display 関数を定義して任意の形のコンポーネントを作るのはとても簡単です。この方法で楕円型、しずく型のボタンを作る事もできます。

これまでに紹介したいくつかのレイアウト関数はどれも貧弱なものです。各領域の高さと幅が可変であるようなグリッドレイアウトを追加する事もできます。また、十分な空きがある限りコンポーネントを並べて配置したいかもしれません。コンポーネントのサイズ変更が子コンポーネントに伝わるようにする必要もあるでしょう。

最小コスト経路を見つける

多くのアプリケーションは重みつき有向グラフ上の最小コスト経路を見つける必要があります。問題はグラフの辺に関連づけられた非負の重みを用いて経路を見つけることです。ここではダイクストラ法を用いて経路を求めてみましょう。

これまでに紹介したライブラリが使えます。出てくる順に並べると、次のモジュールが使われます。入出力のために Genlex と Printf、キャッシュの実装のために Weak、キャッシュによって節約できた時間を計るために Sys、グラフィカルユーザーインターフェースを作るために Awi。Sys モジュールはコマンドライン引数としてグラフが記述されたファイルを受け取るためにも使われます。

グラフの表現

重みつき有向グラフはノードの集合、辺の集合、辺の集合から重み値への写像により定義されます。**重みつき有向グラフ**のデータの表現方法はたくさんあります。

- 隣接行列:
  行列の各要素 $(m(i,j))$ はノード i からノード j への辺を表し、その値は辺の重みを表します。
- 隣接リスト:
  各ノード i はリスト $[(j_1,w_1);..;(j_n,w_n)]$ に関連づけられ、それぞれの組 (i,j_k,w_k) は重みが w_k である辺を表します。
- 3 種 (?):
  ノードリスト、辺リスト、それから辺の重みを求める関数です。

各表現を使ったときの性質はグラフの規模と辺の数によります。このアプリケーションの目標はいかにしてメモリを全部使わずに最後に行われた計算をキャッシュするかということなので、重みつき有向グラフを表すのに隣接行列を使うことにします。この方法だとリスト操作によってメモリ使用量が増えることはありません。

```
# type  cost  = Nan | Cost of float;;
# type  adj_mat =  cost array array;;
# type 'a graph = { mutable ind : int;
                    size : int;
                    nodes : 'a array;
                    m : adj_mat};;
```

size フィールドはノード数の最大値、ind フィールドは実際のノード数です。

ここでグラフを生成する関数を定義しましょう。

グラフを作る関数は引数としてノードとノード番号の最大値をとります。

```
# let create_graph n s =
    { ind = 0; size = s; nodes = Array.create s n;
      m = Array.create_matrix s s Nan } ;;
val create_graph : 'a -> int -> 'a graph = <fun>
```

belongs_to 関数はノード n がグラフ g に含まれるかどうかを調べて返します。

```
# let belongs_to n g =
    let rec aux i =
      (i < g.size) & ((g.nodes.(i) = n) or (aux (i+1)))
    in aux 0;;
val belongs_to : 'a -> 'a graph -> bool = <fun>
```

index 関数はグラフ g におけるノード n のインデックスを返します。もしそのノードが存在しなかったら例外 Not_found が投げられます。

```
# let index n g =
    let rec aux i =
      if i >= g.size then raise Not_found
      else if g.nodes.(i) = n then i
           else aux (i+1)
    in aux 0 ;;
val index : 'a -> 'a graph -> int = <fun>
```

次の 2 つの関数はノードとコスト c を持つ辺をグラフに追加します。

```
# let add_node n g =
    if g.ind = g.size then failwith "the graph is full"
    else if belongs_to n g then failwith "the node already exists"
    else (g.nodes.(g.ind) <- n; g.ind <- g.ind + 1) ;;
val add_node : 'a -> 'a graph -> unit = <fun>
# let add_edge e1 e2 c g =
    try
      let x = index e1 g and y = index e2 g in
        g.m.(x).(y) <- Cost c
    with Not_found → failwith "node does not exist" ;;
val add_edge : 'a -> 'a -> float -> 'a graph -> unit = <fun>
```

これで、ノードと辺のリストから簡単に重みつき有向グラフを作ることができます。test_aho 関数は図 13.8 のグラフを生成します。

```
# let test_aho () =
    let g = create_graph "nothing" 5 in
      List.iter (fun x → add_node x g) ["A"; "B"; "C"; "D"; "E"];
      List.iter (fun (a,b,c) → add_edge a b c g)
        ["A","B",10.;
         "A","D",30.;
         "A","E",100.0;
         "B","C",50.;
         "C","E",10.;
         "D","C",20.;
         "D","E",60.];
      for i=0 to g.ind -1 do g.m.(i).(i) <- Cost 0.0 done;
       g;;
val test_aho : unit -> string graph = <fun>
# let a = test_aho();;
val a : string graph =
```

```
{ind = 5; size = 5; nodes = [|"A"; "B"; "C"; "D"; "E"|];
 m = [|[|Cost 0; Cost 10; Nan; Cost 30; Cost ...|]; ...|]}
```

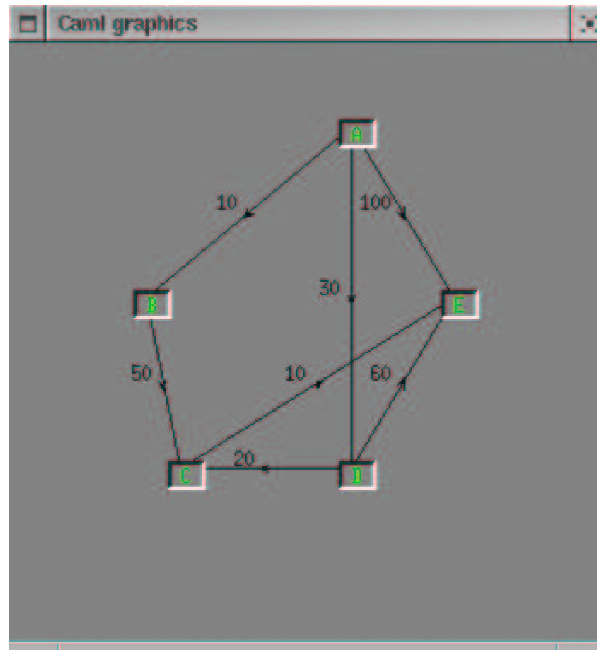


図 13.8: テストグラフ

いろいろなグラフの構築

プログラム中で直接グラフを作るのでは退屈です。これを避けるために、グラフの簡単なテキスト表現を定義しましょう。こうすると、グラフを記述したテキストファイルをアプリケーションが読み込むことで、そのグラフを生成することができます。

グラフのテキスト表現は次のような行の集まりです。

- ノード数: `SIZE number`;
- ノード名: `NODE name`;
- 辺の重み: `EDGE name1 name2 cost`;
- コメント: `# comment`.

例えば、次の `aho.dat` ファイルは図 13.8 のグラフを表します。

```
SIZE 5
NODE A
NODE B
```

```
NODE C
NODE D
NODE E
EDGE A B 10.0
EDGE A D 30.0
EDGE A E 100.0
EDGE B C 50.
EDGE C E 10.
EDGE D C 20.
EDGE D E 60.
```

グラフファイルを読むために字句解析モジュール Genlex を使います。字句解析器はキーワードのリスト keywords から構築されます。

parse_line 関数はキーワードに関連づけられたアクションを実行します。

```
# let keywords = [ "SIZE"; "NODE"; "EDGE"; "#"];;
val keywords : string list = ["SIZE"; "NODE"; "EDGE"; "#"]
# let lex_line l = Genlex.make_lexer keywords (Stream.of_string l);;
val lex_line : string -> Genlex.token Stream.t = <fun>
# let parse_line g s = match  s with parser
    [< '(Genlex.Kwd  "SIZE"); '(Genlex.Int n) >] →
        g := create_graph "" n
  | [< '(Genlex.Kwd  "NODE"); '(Genlex.Ident name) >] →
        add_node name !g
  | [< '(Genlex.Kwd  "EDGE"); '(Genlex.Ident e1);
       '(Genlex.Ident  e2); '(Genlex.Float c) >] →
        add_edge e1 e2 c !g
  | [< '(Genlex.Kwd  "#") >] → ()
  | [<>] → () ;;
Characters 44-46:
    [< '(Genlex.Kwd  "SIZE"); '(Genlex.Int n) >] ->
    ^^
Syntax error
```

グラフファイルからグラフを生成する関数で解析器が使われます。

```
# let create_graph name =
    let g = ref {ind=0; size=0; nodes =[||]; m = [||]} in
    let ic = open_in name in
     try
       print_string ("Loading "^name^": ");
       while true do
        print_string ".";
         let l = input_line ic in parse_line g (lex_line l)
       done;
       !g
```

```
    with End_of_file → print_newline(); close_in ic; !g ;;
Characters 238-248:
        let l = input_line ic in parse_line g (lex_line l)
                                 ^^^^^^^^^^
Unbound value parse_line
```

次のコマンドはファイル aho.dat からグラフを生成します。

```
# let b = create_graph "PROGRAMMES/aho.dat" ;;
val b : int -> string graph = <fun>
```

ダイクストラ法

ダイクストラ法は 2 点間の最小コスト経路を求めるアルゴリズムです。ノード $n1$、$n2$ 間のコストはそれらのノード間の経路のコストを合計したものです。このアルゴリズムを適用するにはコストは非負であることが必要です。このため同じノードを 2 回以上通ってもコストが小さくなることはありません。

ダイクストラ法を使うと、ある起点ノード $n1$ からたどれる全てのノードについてその 2 点間の最小コスト経路を効率よく計算することができます。ダイクストラ法では $n1$ からの最小コスト経路がすでに分かっているようなノードの集合を考えます。集合に含まれるノードから 1 つの辺をたどることで到達できるノードを考えることでこの集合は次々と大きくなっていきます。集合に追加されるのは、そのようなノードの候補の中で最小のコストを持つものです。

計算の状態を記録しておくための *comp_state* 型と状態を生成する関数を定義します。

```
# type comp_state = { paths : int array;
                      already_treated : bool array;
                      distances : cost array;
                      source : int;
                      nn : int};;
# let create_state () =  { paths = [||]; already_treated = [||]; distances = [||];
                           nn = 0; source = 0};;
```

source フィールドは起点ノードを表します。already_treated フィールドは起点ノードからそのノードまでの最小コスト経路がすでに分かっていることを表します。nn フィールドはノードの総数を表します。distances フィールドは起点ノードからの最小コストを表します。path には最小コスト経路をたどるための先行ノードが入ります。起点ノードへの経路は path をたどることで構築することができます。

コスト関数

コストに関する4つの関数を定義します。a_cost は辺が存在するかどうか、float_of_cost は浮動小数点数で表されたコスト値、add_cost は2つのコストの合計、less_cost は片方のコストが他方のコストより小さいかどうかを返します。

```
# let a_cost c = match c with Nan → false | _-> true;;
val a_cost : cost -> bool = <fun>
# let float_of_cost c = match c with
     Nan → failwith "float_of_cost"
   | Cost x → x;;
val float_of_cost : cost -> float = <fun>
# let add_cost  c1 c2 = match (c1,c2) with
     Cost x, Cost y → Cost (x+.y)
   | Nan, Cost y → c2
   | Cost x, Nan → c1
   | Nan, Nan →  c1;;
val add_cost : cost -> cost -> cost = <fun>
# let less_cost  c1 c2 = match (c1,c2) with
     Cost x, Cost y → x < y
   | Cost x, Nan → true
   | _, _ →  false;;
val less_cost : cost -> cost -> bool = <fun>
```

Nan は計算や比較される際に特別な役割を持ちます。メイン関数を紹介する所（391）でこの事について再び触れます。

アルゴリズムの実装

既知の最小コスト経路から次のノードを見つけるための手順は2つに分けることができます。first_not_treated は最小コスト経路が既知のノード集合に含まれない最初のノードを返します。このノードは次の関数、least_not_treated の初期値として使われます。この関数は、集合に含まれないノードのうち起点からのコストが最小のノードを返します。そのノードが集合に追加されることになります。

```
# exception Found of int;;
exception Found of int
# let first_not_treated cs =
     try
       for i=0 to cs.nn-1 do
         if not cs.already_treated.(i) then raise (Found i)
       done;
       raise Not_found;
       0
     with Found i → i ;;
val first_not_treated : comp_state -> int = <fun>
# let least_not_treated p cs =
```

```
    let ni = ref p
    and nd = ref cs.distances.(p) in
      for i=p+1 to cs.nn-1 do
        if not cs.already_treated.(i) then
         if less_cost cs.distances.(i)  !nd then
           ( nd := cs.distances.(i);
             ni := i )
      done;
      !ni,!nd;;
val least_not_treated : int -> comp_state -> int * cost = <fun>
```

one_round 関数は新しいノードを既知の集合に追加し、残りのノードについて必要ならば起点からのコストを更新します。

```
# exception No_way;;
exception No_way
# let one_round cs  g =
    let p = first_not_treated cs  in
    let np,nc = least_not_treated p cs in
      if not(a_cost nc  ) then raise No_way
      else
      begin
        cs.already_treated.(np) <- true;
        for i = 0 to cs.nn -1 do
          if not cs.already_treated.(i) then
          if a_cost g.m.(np).(i) then
            let ic = add_cost cs.distances.(np)  g.m.(np).(i) in
              if less_cost ic cs.distances.(i)   then (
                cs.paths.(i) <- np;
                cs.distances.(i) <- ic
               )
        done;
        cs
      end;;
val one_round : comp_state -> 'a graph -> comp_state = <fun>
```

さあ、残るのはこれまで定義した関数を繰り返し呼ぶことです。dij 関数は起点ノードとグラフを引数にとり起点から全てのノードまでの最小コスト経路情報を含む comp_state 型の値を返します。

```
# let dij s g =
    if belongs_to s g then
    begin
      let i = index s g in
```

```
      let cs = { paths = Array.create g.ind (-1) ;
                 already_treated = Array.create g.ind false;
                 distances = Array.create g.ind Nan;
                 nn = g.ind;
                 source = i}  in
        cs.already_treated.(i) <- true;
        for j=0 to g.ind-1 do
          let c = g.m.(i).(j) in
            cs.distances.(j) <- c;
            if a_cost c then cs.paths.(j) <- i
        done;
        try
          for k = 0 to cs.nn-2 do
            ignore(one_round cs g)
          done;
          cs
        with No_way → cs
    end
    else failwith "dij: node unknown";;
val dij : 'a -> 'a graph -> comp_state = <fun>
```

Nan は距離の初期値で、比較関数 less_cost の定義からは無限大を表します。それに対してコストの加算関数 add_cost ではゼロ扱いされています。これにより距離テーブルの実装が単純になっています。

ここでダイクストラ法の実行を試すことができます。

```
# let g = test_aho ();;
# let r = dij "A" g;;
```

返り値は

```
# r.paths;;
- : int array = [|0; 0; 3; 0; 2|]
# r.distances;;
- : cost array = [|Cost 0; Cost 10; Cost 50; Cost 30; Cost 60|]
```

結果の表示

結果を読みやすくするために、表示関数を作ってみましょう。

dij の返り値に含まれる paths テーブルは計算された経路の最後の 1 辺しか含んでいません。すべての経路を得るためには再帰的に起点まで戻る必要があります。

```
# let display_state f (g,st)  dest =
```

```
    if belongs_to dest g then
       let d = index dest g in
         let rec aux is =
            if is = st.source then Printf.printf "%a"  f g.nodes.(is)
            else (
              let old = st.paths.(is) in
              aux old;
              Printf.printf " -> (%4.1f) %a" (float_of_cost g.m.(old).(is))
                                            f g.nodes.(is)
            )
         in
           if not(a_cost st.distances.(d)) then Printf.printf "no way\n"
           else (
             aux d;
             Printf.printf " = %4.1f\n" (float_of_cost st.distances.(d)));;
val display_state :
  (out_channel -> 'a -> unit) -> 'a graph * comp_state -> 'a -> unit = <fun>
```

この再帰関数はノードを正しい順番で表示するためにスタックを使っています。グラフ表示の多相性を保てるように、"a"フォーマットは引数に関数パラメータ f をとります。

"A"（index 0）から "E"（index 4）までの最小コスト経路は次のように表示されます。

```
# display_state (fun x y → Printf.printf "%s!" y) (a,r) "E";;
A! -> (30.0) D! -> (20.0) C! -> (10.0) E! = 60.0
- : unit = ()
```

経路上の各ノードと各辺のコストが表示されます。

キャッシュの紹介

ダイクストラ法は起点ノードから全てのノードについて最小コスト経路を計算します。これらの最小コスト経路を、起点ノードが同じであるような次回の問い合わせに備えて保存しておくというアイデアが浮かびます。しかしこれにはかなりの量のメモリーが必要です。ここでは "ウィークポインタ" を紹介します。ある起点ノードに対する計算の結果はウィークポインタのテーブルに保存されます。次の計算の時は、すでに計算された結果があるかどうかを判定することができます。この種類のポインタは弱いので、その領域に入っていた状態データは必要ならばガーベッジコレクタによって解放されます。この仕組みにより、大量のメモリーの割り当てによって残りのプログラムが中断されることが避けられます。最悪の場合でも未来の問い合わせによって計算が繰り返されるだけです。

キャッシュの実装

新しい型 *'a comp_graph* を定義します。

```
# type 'a comp_graph =
    { g : 'a graph; w : comp_state Weak.t } ;;
```
g はグラフ、w フィールドには各起点ごとの計算状態を指すウィークポインタのテーブルを表します。

これらの値は create_comp_graph 関数により生成されます。

```
# let create_comp_graph g =
    { g = g;
      w = Weak.create g.ind } ;;
val create_comp_graph : 'a graph -> 'a comp_graph = <fun>
```

dij_quick 関数は計算が既に行われたかどうかを判定します。既に行われていたら、保存されている結果を返します。そうでなければ計算が行われ、結果がウィークポインタのテーブルに保存されます。

```
# let dij_quick s cg =
    let i = index s cg.g in
      match Weak.get cg.w i with
        None → let cs = dij s cg.g in
                  Weak.set cg.w i (Some cs);
                  cs
      | Some cs → cs;;
val dij_quick : 'a -> 'a comp_graph -> comp_state = <fun>
```

表示関数は以前と同じものが使えます。
```
# let cg_a = create_comp_graph a in
  let r = dij_quick "A" cg_a in
   display_state (fun x y → Printf.printf "%s!" y) (a,r) "E" ;;
A! -> (30.0) D! -> (20.0) C! -> (10.0) E! = 60.0
- : unit = ()
```

性能評価

ランダムな起点ノードリストのそれぞれを処理させて dij、dij_quick 関数の性能を計ってみましょう。この方法で頻繁に最小コスト経路を計算するアプリケーション、例えば路線検索システムのようなものをシミュレートすることができます。

時間を計測する関数を定義します。

```
# let exe_time f g ss =
    let t0 = Sys.time() in
```

```
      Printf.printf "Start (%5.2f)\n" t0;
      List.iter (fun s → ignore(f s g)) ss;
      let t1 = Sys.time() in
       Printf.printf "End (%5.2f)\n" t1;
       Printf.printf "Duration = (%5.2f)\n" (t1 -. t0) ;;
val exe_time : ('a -> 'b -> 'c) -> 'b -> 'a list -> unit = <fun>
```

そして 20000 個のランダムなノードリストを作り、グラフ a に対して性能を測ります。

```
# let ss =
    let ss0 = ref [] in
    let i0 = int_of_char 'A' in
    let new_s i = Char.escaped (char_of_int (i0+i)) in
    for i=0 to 20000 do ss0 := (new_s (Random.int a.size))::!ss0 done;
    !ss0 ;;
val ss : string list =
  ["A"; "B"; "D"; "A"; "E"; "C"; "B"; "B"; "D"; "E"; "B"; "E"; "C"; "E"; "E";
   "D"; "D"; "A"; "E"; ...]
# Printf.printf"Function dij :\n";
  exe_time dij a ss ;;
Function dij :
Start ( 1.51)
End ( 1.93)
Duration = ( 0.42)
- : unit = ()
# Printf.printf"Function dij_quick :\n";
  exe_time dij_quick (create_comp_graph a) ss ;;
Function dij_quick :
Start ( 1.93)
End ( 1.99)
Duration = ( 0.06)
- : unit = ()
```

予想通りの結果になりました。保持された結果に直接アクセスするのは 2 回目の計算よりかなり速いです。

グラフィカルインターフェース

Awi ライブラリを使ってグラフを表示するグラフィカルインターフェースを作りましょう。起点ノードと終点ノードを選択すると、その間の最小コスト経路が計算されて表示されます。そのために、グラフィカルインターフェース、グラフ、計算状態をフィールドに持つ *'a gg* 型を定義します。

```
#  #load "PROGRAMMES/awi.cmo";;
```

```
# type 'a gg = { mutable src : 'a * Awi.component;
                 mutable dest : 'a * Awi.component;
                 pos : (int * int) array;
                 cg : 'a comp_graph;
                 mutable state : comp_state;
                 mutable main : Awi.component;
                 to_string : 'a -> string;
                 from_string : string -> 'a } ;;
```

src、dest フィールドはノードとコンポーネントの組です。pos フィールドは各コンポーネントの座標を持ちます。main はコンポーネント群のメインコンテナです。to_string 関数、from_string 関数は *'a* 型と文字列型の間の変換関数です。

これらの値を生成するために必要なのはグラフ情報、位置テーブル、変換関数です。

```
# let create_gg cg vpos ts fs =
    {src = cg.g.nodes.(0),Awi.empty_component;
     dest = cg.g.nodes.(0),Awi.empty_component;
     pos = vpos;
     cg = cg;
     state = create_state () ;
     main = Awi.empty_component;
     to_string = ts;
     from_string = fs};;
val create_gg :
  'a comp_graph ->
  (int * int) array -> ('a -> string) -> (string -> 'a) -> 'a gg = <fun>
```

可視化

グラフを表示するためにはノードと辺を描画する必要があります。ノードは Awi のボタンコンポーネントで表し、辺は直接メインウィンドウに描画することにします。display_edge 関数は辺を描画します。display_shortest_path は見つかった経路を異なる色で描画します。

Drawing Edges 辺は 2 つのノードを結び、重みに関連づけられます。2 つのノードが接続されている様子は線分で表すことができます。問題は線分の向きをどう表すかですが、これは矢印で表すことにします。矢印は、正しい方向を向くように線分の角度だけ回転させます。最後に辺の重みが辺のとなりに描画されます。

辺の矢印を描画するために、座標を回転させる rotate 関数と移動させる translate 関数を定義します。display_arrow 関数により矢印を描画します。

```
# let rotate l a =
    let ca = cos a  and sa = sin a in
      List.map (function (x,y) → ( x*.ca +. -.y*.sa, x*.sa +. y*.ca)) l;;
val rotate : (float * float) list -> float -> (float * float) list = <fun>
# let translate l (tx,ty) =
    List.map (function (x,y) → (x +. tx, y +. ty)) l;;
val translate : (float * float) list -> float * float -> (float * float) list =
  <fun>
# let display_arrow (mx,my) a =
    let triangle = [(5.,0.); (-3.,3.); (1.,0.); (-3.,-3.); (5.,0.)] in
    let tr =  rotate triangle a in
    let ttr = translate tr (mx,my) in
    let tt = List.map (function (x,y) → (int_of_float x, int_of_float y)) ttr
    in
      Graphics.fill_poly (Array.of_list tt);;
val display_arrow : float * float -> float -> unit = <fun>
```

辺の重みを表す文字列の位置は辺の角度に依存します。

```
# let display_label (mx,my) a lab =
    let (sx,sy) = Graphics.text_size lab in
    let pos = [ float(-sx/2),float(-sy) ] in
    let pr = rotate pos a in
    let pt = translate pr (mx,my) in
      let px,py = List.hd pt in
        let ox,oy = Graphics.current_point () in
          Graphics.moveto ((int_of_float mx)-sx-6)
                          ((int_of_float my) );
          Graphics.draw_string lab;
          Graphics.moveto ox oy;;
val display_label : float * float -> float -> string -> unit = <fun>
```

それではこれまでに定義した関数を display_edge 関数で使いましょう。

```
# let display_edge gg col i j =
    let g = gg.cg.g in
    let x,y = gg.main.Awi.x,gg.main.Awi.y in
      if a_cost g.m.(i).(j) then (
        let (a1,b1) = gg.pos.(i)
        and (a2,b2) = gg.pos.(j) in
          let x0,y0 = x+a1,y+b1   and x1,y1 = x+a2,y+b2 in
          let rxm =  (float(x1-x0)) /. 2.  and rym =  (float(y1-y0)) /. 2. in
          let xm = (float x0) +. rxm and ym = (float y0) +. rym in
            Graphics.set_color col;
            Graphics.moveto x0 y0;
```

```
            Graphics.lineto x1 y1;
            let a =  atan2 rym rxm in
            display_arrow (xm,ym) a;
            display_label (xm,ym) a
              (string_of_float(float_of_cost g.m.(i).(j)));;
val display_edge : 'a gg -> Graphics.color -> int -> int -> unit = <fun>
```

経路の表示　経路を表示するには、経路に含まれる全ての辺を描画すればいいです。したがって、経路の描画はこれまで辺を描いてきたやり方でできます。

```
# let rec display_shortest_path gg col dest =
     let g = gg.cg.g in
       if belongs_to dest g then
         let d = index dest g in
          let rec aux is =
             if is = gg.state.source then ()
             else (
               let old = gg.state.paths.(is) in
                 display_edge gg col old is;
                 aux old )
          in
            if not(a_cost gg.state.distances.(d)) then Printf.printf "no way\n"
            else aux d;;
val display_shortest_path : 'a gg -> Graphics.color -> 'a -> unit = <fun>
```

グラフの表示　display_gg 関数はグラフを描画します。もし終点ノードが空でなければ、起点ノードと終点ノード間の経路が描画されます。

```
#  let display_gg gg () =
     Awi.display_rect gg.main ();
     for i=0 to gg.cg.g.ind -1 do
       for j=0 to gg.cg.g.ind -1 do
         if i<> j then display_edge gg (Graphics.black) i j
       done
     done;
     if snd gg.dest != Awi.empty_component then
        display_shortest_path gg Graphics.red (fst gg.dest);;
val display_gg : 'a gg -> unit -> unit = <fun>
```

ノードコンポーネント

ノードも描画される必要があります。ユーザーは起点ノードと終点ノードを指定することができるのでノード用のコンポーネントを定義します。

ユーザーの主なアクションは終点ノードを指定することです。したがってノードはマウスクリックに反応できるコンポーネントである必要があります。そのようなコンポーネントとして、マウスクリックに反応するボタンコンポーネントを選びました。

ノードのアクション　ノードが選択されているかどうかを表す必要があります。このために、inverse 関数によりノードの背景色を変えられるようにします。

```
# let inverse b =
    let gc = Awi.get_gc b in
      let fcol = Awi.get_gc_fcol gc
      and bcol = Awi.get_gc_bcol gc in
        Awi.set_gc_bcol gc fcol;
        Awi.set_gc_fcol gc bcol;;
val inverse : Awi.component -> unit = <fun>
```

ノード選択は action_click 関数により行われます。この関数はノードがクリックされた時に呼ばれます。引数としてボタンに関連づけられたノードとグラフが渡されます。起点/終点ノードが両方選択されたら、dij_quick 関数により最小コスト経路が計算されます。

```
# let action_click node gg b bs =
    let (s1,s) = gg.src
    and (s2,d) = gg.dest in
     if  s == Awi.empty_component then (
       gg.src  <- (node,b); inverse b )
     else
      if d == Awi.empty_component then (
        inverse b;
        gg.dest <- (node,b);
        gg.state <- dij_quick s1 gg.cg;
       display_shortest_path gg (Graphics.red) node
      )
      else (inverse s; inverse d;
            gg.dest <- (s2,Awi.empty_component);
            gg.src <- node,b; inverse b);;
val action_click : 'a -> 'a gg -> Awi.component -> 'b -> unit = <fun>
```

インターフェースの作成 インターフェースを作るメイン関数はグラフとオプションリストを引数にとり、いろいろなコンポーネントを作ってグラフの各要素に関連づけます。引数はグラフ（gg）、幅と高さ（gw、gh）、グラフ/ノードオプションリスト（lopt）、ノードの境界線オプションリスト（lopt2）です。

```
# let main_gg gg gw gh lopt lopt2 =
    let gc = Awi.make_default_context () in
      Awi.set_gc gc lopt;
  (* compute the maximal button size *)
      let vs = Array.map gg.to_string gg.cg.g.nodes in
      let vsize = Array.map Graphics.text_size  vs in
        let w = Array.fold_right (fun (x,y) → max x)  vsize 0
        and h = Array.fold_right (fun (x,y) → max y)  vsize 0 in
  (* create the main panel *)
        gg.main <- Awi.create_panel true gw gh lopt;
        gg.main.Awi.display <- display_gg gg;
  (* create the buttons *)
          let vb_bs =
            Array.map (fun x → x,Awi.create_button (" "^(gg.to_string x)^" ")
                         lopt)
                       gg.cg.g.nodes in
            let f_act_b = Array.map (fun (x,(b,bs)) →
                                      let ac = action_click x  gg b
                                      in Awi.set_bs_action bs ac)  vb_bs in
            let bb =
              Array.map (function (_,(b,_)) → Awi.create_border b lopt2) vb_bs
            in
              Array.iteri
                (fun i (b) → let x,y = gg.pos.(i) in
                                  Awi.add_component gg.main  b
                                    ["PosX",Awi.Iopt (x-w/2);
                                     "PosY", Awi.Iopt (y-h/2)]) bb;
            ();;
val main_gg :
  'a gg ->
  int ->
  int -> (string * Awi.opt_val) list -> (string * Awi.opt_val) list -> unit =
  <fun>
```

ボタンは自動的に作成され、メインウィンドウに配置されます。

インターフェースのテスト インターフェースを作成する全ての準備が整いました。変換関数を単純にするためにノードが文字列であるようなグラフを使います。グラフ gg を次のようにして作ります。

```
# let id x = x;;
# let pos =  [| 200, 300; 80, 200 ; 100, 100; 200, 100;  260, 200 |];;
# let gg = create_gg  (create_comp_graph (test_aho())) pos id id;;
# main_gg  gg 400 400 ["Background", Awi.Copt  (Graphics.rgb 130 130 130);
              "Foreground",Awi.Copt  Graphics.green]
  [ "Relief", Awi.Sopt "Top";"Border_size", Awi.Iopt 2];;
```

Awi.loop true false gg.main;; を実行すると Awi ライブラリのイベントループが始まります。

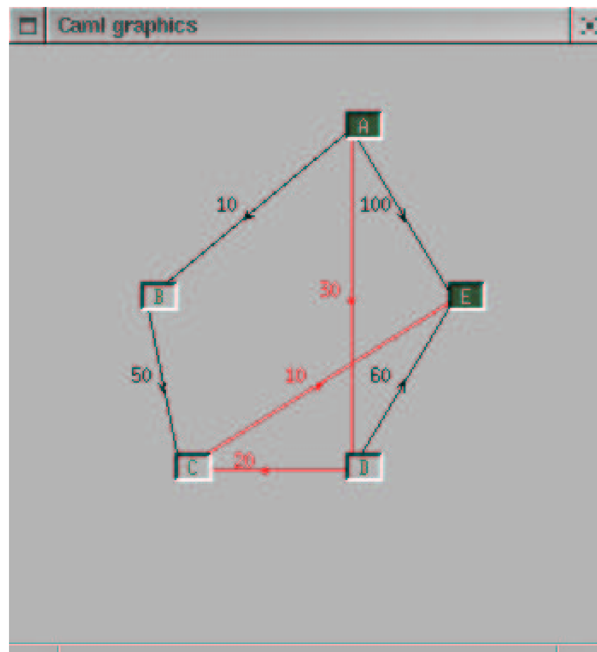


図 13.9: ノードの選択

図 13.9 はノード"A"、"E"間の最小コスト経路が計算された結果です。経路の各辺は色が変わっています。

スタンドアロンアプリケーションの作成

ここでスタンドアロンアプリケーションを作るのに必要な手順を紹介しましょう。アプリケーションはグラフが保存されたファイルの名前を引数にとります。スタンドアロンアプリケーションを実行する環境に Objective Caml がインストールされている必要はありません。

グラフ記述ファイル

グラフ記述ファイルにはグラフそのものに関する情報に加えてグラフィカルインターフェースに関する情報も記述することにします。そこで後者の情報のために新たなフォーマットを定義します。これらの記述から *g_info* 型の値を生成します。

```
# type g_info = {npos : (int * int) array;
                 mutable opt : Awi.lopt;
                 mutable g_w : int;
                 mutable g_h : int};;
```

グラフィカルインターフェースの情報のフォーマットはリスト key2 の 4 つのキーワードにより記述されます。

```
# let key2 = ["HEIGHT"; "LENGTH"; "POSITION"; "COLOR"];;
val key2 : string list = ["HEIGHT"; "LENGTH"; "POSITION"; "COLOR"]
# let lex2 l = Genlex.make_lexer key2 (Stream.of_string l);;
val lex2 : string -> Genlex.token Stream.t = <fun>

# let pars2 g gi s = match s with parser
      [< '(Genlex.Kwd "HEIGHT"); '(Genlex.Int i) >] → gi.g_h <- i
   |  [< '(Genlex.Kwd "LENGTH"); '(Genlex.Int i) >] → gi.g_w <- i
   |  [< '(Genlex.Kwd "POSITION"); '(Genlex.Ident s);
         '(Genlex.Int i); '(Genlex.Int j) >] → gi.npos.(index s g) <- (i,j)
   |  [< '(Genlex.Kwd "COLOR"); '(Genlex.Ident s);
         '(Genlex.Int r); '(Genlex.Int g); '(Genlex.Int b) >] →
              gi.opt <- (s, Awi.Copt (Graphics.rgb r g b)) :: gi.opt
   | [<>] → ();;
Characters 44-46:
     [< '(Genlex.Kwd "HEIGHT"); '(Genlex.Int i) >] -> gi.g_h <- i
     ^^
Syntax error
```

アプリケーションの作成

create_graph 関数は入力ファイルの名前を受け取り、グラフとグラフィカルインターフェースに関する情報の組を返します。

```
# let create_gg_graph name =
    let g = create_graph name in
      let gi = {npos = Array.create g.size (0,0); opt=[]; g_w =0; g_h = 0;} in
    let ic = open_in name in
    try
      print_string ("Loading (pass 2) " ^name ^" : ");
```

```
      while true do
        print_string ".";
        let l = input_line ic in pars2 g gi (lex2 l)
      done ;
      g,gi
    with End_of_file → print_newline(); close_in ic; g,gi;;
Characters 95-96:
      let gi = {npos = Array.create g.size (0,0); opt=[]; g_w =0; g_h = 0;} in
                                    ^
This expression has type int -> 'a graph but is here used with type 'b graph
```

create_app 関数はグラフィカルインターフェースを構築します。

```
# let create_app name =
    let g,gi = create_gg_graph name in
    let size = (string_of_int gi.g_w) ^ "x" ^ (string_of_int gi.g_h) in
    Graphics.open_graph (" "^size);
    let gg = create_gg (create_comp_graph g) gi.npos id id in
    main_gg gg gi.g_w gi.g_h
      [ "Background", Awi.Copt (Graphics.rgb 130 130 130) ;
        "Foreground", Awi.Copt Graphics.green ]
      [ "Relief", Awi.Sopt "Top" ; "Border_size", Awi.Iopt 2 ] ;
    gg;;
Characters 37-52:
    let g,gi = create_gg_graph name in
               ^^^^^^^^^^^^^^^
Unbound value create_gg_graph
```

最後に、main 関数は入力ファイルをコマンドライン引数として受け取り、インターフェースつきのグラフを構築し、メインコンポーネントのイベント処理ループを開始します。

```
# let main () =
    if (Array.length Sys.argv ) <> 2
    then Printf.printf "Usage: dij.exe filename\n"
    else
      let gg = create_app Sys.argv.(1) in
        Awi.loop true false gg.main;;
Characters 122-132:
      let gg = create_app Sys.argv.(1) in
               ^^^^^^^^^^
Unbound value create_app
```

このプログラムの最後の式は main をスタートさせます。

実行可能ファイル

スタンドアロンアプリケーションを作る動機は配布が容易にしたいということです。このセクションで定義した型や関数を dij.ml にまとめました。このファイルをコンパイルして必要なライブラリをリンクします。Linux 環境でコンパイルするためのコマンドラインは次のようになります。

```
ocamlc -custom -o dij.exe graphics.cma awi.cmo graphs.ml \
    -cclib -lgraphics -cclib -L/usr/X11/lib -cclib -lX11
```

Graphics ライブラリを使ったスタンドアロンアプリケーションのコンパイル方法は 5 章と 7 章に書いてあります。

最後に

このアプリケーションの骨格は十分に一般的なものです。重みつきグラフで表現できる種類の問題はたくさんあります。たとえば迷路中の経路は、各交差点がノードであるようなグラフとして表せます。迷路を解くことは、スタート/ゴール間の最小コスト経路を計算することに対応します。

C 言語、Objective Caml で性能の比較を行うために私達はダイクストラ法のアルゴリズムを C 言語でも書いてみました。C のプログラムでは Objective Caml のと同じデータ構造を使っています。

グラフィカルインターフェースをより良くするには、ファイル名を入力するテキストフィールドとグラフの読み込み/保存を行う 2 つのボタンを追加します。見た目を良くするためにノードの位置をマウスで修正できるといいかもしれません。

2 つめの改善点はノードの外見を変えられる機能です。ボタンを表示するときには矩形を描画する関数が呼ばれます。ノードの描画にポリゴンを使うこともできます。

Part III

Application Structure

The third part of this work is dedicated to application development and describes two ways of organizing applications: modules and objects. The goal is to easily structure an application for incremental and rapid development, maintenance facilitated by the ability to change gracefully, and the possibility of reusing large parts for future development.

We have already presented the language's predefined modules (第 8 章参照) viewed as compilation units. Objective Caml's module language supports on the one hand the definition of new simple modules in order to build one's own libraries, perhaps including abstract types, and on the other hand the definition of modules parameterized by other modules, called **functors**. The advantage of this parameterization lies in being able to "apply" a module to different argument modules in order to create specialized modules. Communication between modules is thus explicit, via the parameter module signature, which contains the types of its global declarations. However, nothing stops you from applying a functor to a module with a more extended signature, as long as it remains compatible with the specified parameter signature.

Besides, the Objective Caml language has an object-oriented extension. First of all object-oriented programming permits structured communication between objects. Rather than applying a function to some arguments, one sends a message (a request) to an object which knows how to deal with it. The object, an instance of a class (a structure gathering together data and methods), then executes the corresponding code. The main relation between classes is inheritance, which lets one describe subclasses which retain all the declarations of the ancestor class. Late binding between the name of a message and the corresponding code within the object takes place during program execution. Nevertheless Objective Caml typing guarantees that the receiving object will always have a method of this name, otherwise type inference would have raised a compile-time error. The second important relation is subtyping, where an object of a certain class can always be used in place of an object of another class. In this way a new type of polymorphism is introduced: inclusion polymorphism.

Finally the construction of a graphical interface, begun in chapter 5, uses different event management models. One puts together in an interface several components with respect to which the user or the system can produce events. The association of a component with a handler for one or more events taking place on it allows one to easily add to and modify such interfaces. The component-event-handler association can be cloaked in several forms: definition of a function (called a callback), inheritance with redefinition of handler methods, or finally registration of a handling object (delegation model).

Chapter 14 is a presentation of modular programming. The different prevailing terminologies of abstract data types and module languages are explained and illustrated by simple modules. Then the module language is detailed. The correspondence between modules (simple or not) and compilation units is made clear.

Chapter 15 contains an introduction to object-oriented programming. It brings a new way of structuring Objective Caml programs, an alternative to modules. This chapters shows how the notions of object-oriented programming (simple and multiple inheritance, abstract classes, parameterized classes, late binding) are articulated with

respect to the language's type system, and extend it by the subtyping relation to inclusion polymorphism.

Chapter 16 compares the two preceding software models and explains what factors to consider in deciding between the two, while also demonstrating how to simulate one by the other. It treats various cases of mixed models. Mixing leads to the enrichment of each of these two models, in particular with parameterized classes using the abstract type of a module.

Chapter 17 presents two classes of applications: two-player games, and the construction of a world of virtual robots. The first example is organized via various parameterized modules. In particular, a parameterized module is used to represent games for application of the minimax $\alpha\beta$ algorithm. It is then applied to two specific games: Connect 4 and Stone Henge. The second example uses an object model of a world and of abstract robots, from which, by inheritence, various simulations are derived. This example is presented in chapter 21.

14

モジュールを使ったプログラミング

Modular design and modular programming support the decomposition of a program into several *software units*, also called *modules*, which can be developed largely independently. ひとつひとつのモジュールは，プログラム全体を構成する他のモジュールとは別にコンパイルすることができます．その結果，あるモジュールを使ってプログラムを開発する人は，そのモジュールのソースコードを得る必要がなくなります — コンパイルされたモジュールのコードがあれば実行できるプログラムをつくるのに十分なのです． しかし，プログラマは使われているモジュールの**インターフェース**，すなわち，そのモジュールが，どんな値，関数，例外，果てはサブモジュールを，どんな名前や型で提供しているかということを知らなくてはなりません．

モジュールのインターフェースを明示的に書き下すことによって，そのモジュールを使うプログラムから，モジュールの実装の詳細を隠蔽することができます．モジュールについて知ることができるのは，モジュール外に「輸出」された定義の名前と型で，その正確な実装はわかりません．ですから，モジュールの保守の際に，高い柔軟性をもってモジュールの実装を変えていくことができます．つまり，インターフェースが変わらず，また挙動が保存されている限り，モジュールを使う側は実装が変わったことに気が付かないでしょう．このことは，大きなプログラムの保守や変更を非常に容易にすることができます．局所定義のように，モジュールインターフェースは，モジュール設計者が公開したくない実装の一部を隠す支援をします．この隠蔽機構の重要な応用として，**抽象データ型**の実装があります．

最後に，Objective Caml のモジュールシステムのように先進的なものは，**汎用体**とも呼ばれる，**パラメータつきモジュール**の定義を支援しています．パラメータつきモジュールは他のモジュールをパラメータとしてとるモジュールで，コード再利用の機会を増やしてくれます．

この章のあらまし

最初の章では標準ライブラリの Stack モジュールの例を使って Objective Caml モジュールを説明し，そして，このモジュールを別の実装・同じインターフェースで作成してみます．第 2 節は，簡単なモジュールに関する Objective Caml の**モジュール言語**を導入し，その使い方のいくつかを示します．特に，モジュール間の型共有について議論します． 第 3 節で，Objective Caml では**ファンクタ**と呼ばれる，パラメータつきモジュールを扱います．最後に第 4 節で，モジュールプログラミングの大きな例として，銀行・顧客からの複数の視点や，いくつかのパラメータがあるような銀行口座管理プログラムを作成します.

コンパイル単位としてのモジュール

Objective Caml は，多数の定義済のモジュールとともに配布されています.

8 章ではこれらのモジュールをプログラム中でどのように使うかを見ました．ここでは，ユーザがどのようにして似たようなモジュールを定義できるかを示します.

インターフェースと実装

Stack という定義済モジュールはスタック—「後入れ先だし」の規則にしたがうような待ち行列—の主要な機能を提供します.

```
# let queue = Stack.create () ;;
val queue : '_a Stack.t = <abstr>
# Stack.push 1 queue ; Stack.push 2 queue ; Stack.push 3 queue ;;
- : unit = ()
# Stack.iter (fun n -> Printf.printf "%d " n)  queue ;;
3 2 1 - : unit = ()
```

Objective Caml はすべてのソースコードとともに配布されていますから，スタックの実際の実装を見ることができます.

ocaml-2.04/stdlib/stack.ml

```
type 'a t = { mutable c : 'a list }
exception Empty
let create () = { c = [] }
let clear s = s.c <- []
let push x s = s.c <- x :: s.c
let pop s = match s.c with  hd::tl -> s.c <- tl; hd  |  []  -> raise Empty
let length s = List.length s.c
let iter f s = List.iter f s.c
```

これをみると，スタックの型（Stack モジュールの外側では Stack.t，内側ではただ単に t と書かれます）はリストを含む変更可能フィールドをひとつ持ったレコードであることがわかります．このリストはスタックの内容を保持していて，リストの先頭がスタックの一番上に対応しています．スタック操作は，基本的なリスト操作をレコードのフィールドに適用することで実装されています．

この内部に関する知識を使えば，スタックを表現しているリストに直接アクセスできそうですが，Objective Caml ではそれは許されません．

```
# let list = queue.c ;;
Characters 12-19:
  let list = queue.c ;;
             ^^^^^^^
Unbound record field label c
```

コンパイラは Stack.t が c フィールドを持ったレコード型であることを知らないかのごとく文句を言ってきます．Stack モジュールのインターフェースをみれば，コンパイラは本当に知らないことがわかります．

ocaml-2.04/stdlib/stack.mli

```
(* Module [Stack]: last-in first-out stacks *)
(* This module implements stacks (LIFOs), with in-place modification. *)

type 'a t         (* The type of stacks containing elements of type ['a]. *)

exception Empty   (* Raised when [pop] is applied to an empty stack. *)

val create: unit -> 'a t
        (* Return a new stack, initially empty. *)
val push: 'a -> 'a t -> unit
        (* [push x s] adds the element [x] at the top of stack [s]. *)
val pop: 'a t -> 'a
        (* [pop s] removes and returns the topmost element in stack [s],
           or raises [Empty] if the stack is empty. *)
val clear : 'a t -> unit
        (* Discard all elements from a stack. *)
val length: 'a t -> int
        (* Return the number of elements in a stack. *)
val iter: ('a -> unit) -> 'a t -> unit
        (* [iter f s] applies [f] in turn to all elements of [s],
           from the element at the top of the stack to the element at the
           bottom of the stack. The stack itself is unchanged. *)
```

モジュールの機能について述べたコメントに加えて，このファイルは，stack.ml ファイルで定義された値，型，例外の識別子のうち，Stack モジュールを使う人に見えるべきものを明示的に並べています．より正確には，インターフェースには，これら輸出される定義の名前と型の仕様を宣言します．特に，t という名前の型が輸出されています

が，その型の表現（つまり c フィールドを持つレコード）はこのインターフェースには与えられていません．なので，Stack モジュールを使う人は，Stack.t がどのように表現されているかわからず，この型の値に直接アクセスすることはできません．このことを，*Stack.t* は**抽象的**である，とか**不透明**であるといいます．

インターフェースはスタックを操作する関数も，その名前と型を与えて宣言しています．（関数の型は，型検査器がそれらの関数が正しく使われているかを検査できるように，明示的に与えられなければいけません．）インターフェースでの値や関数の宣言は次のような構文で行います．

構文 : `val nom : type`

インターフェースと実装を関連づける

上に示したように，Stack は，様々な定義をしている実装部分と，エクスポートされる定義の宣言をしているインターフェース部分二つの部分からなります．インターフェースに宣言されているものは，全て，実装部分にマッチする定義がなければなりません．また，実装部分に定義されている値や関数の型は，インターフェースに宣言された型とマッチしていなければなりません．

インターフェースと実装の関係は対称的ではありません．実装はインターフェースが要求するより異状の定義を含むことができます．典型的には，エクスポートされた関数定義が，その名前がインターフェースに現れない補助関数を使うことができます．このような補助関数はモジュールのクライアントからは直接呼ぶことができません．同様に，インターフェースで定義の型を制限することができます．**let** *id x* = *x* で定義される，恒等関数 id を含むモジュールを考えてみましょう． このインターフェースは id を，（より一般的な型である *'a* --> *'a* の代りに）*int* --> *int* 型として宣言することができます．すると，このモジュールのクライアントは id を整数にしか適用することができません．

モジュールのインターフェースは，その実装から，明確に切り離されていますから，ひとつのインターフェースにたいして，いくつかの実装を与えたりすることができます．例えば，異なるアルゴリズムや，操作が同じであるデータ構造をテストしたりすることができます．例として，Stack モジュールの別の実装をみてみましょう．これは，リストの代りに配列を使って実装しています．

```
type 'a t = { mutable sp : int; mutable c : 'a array }
exception Empty
let create () = { sp=0 ; c = [||] }
let clear s = s.sp <- 0; s.c <- [||]
let size = 5
let increase s = s.c <- Array.append s.c (Array.create size s.c.(0))

let push x s =
  if s.sp >= Array.length s.c then increase s ;
  s.c.(s.sp) <- x ;
  s.sp <- succ s.sp
```

```
let pop s =
  if s.sp = 0 then raise Empty
  else let x = s.c.(s.sp) in s.sp <- pred s.sp ; x

let length s = s.sp
let iter f s = for i = pred s.sp downto 0 do f s.sc.(i) done
```

この新しい実装は，インターフェースファイル stack.mli からの要求を満たしています．ゆえに，どんなプログラムでも，定義済の Stack の代りに，この実装を使うことができます．

分割コンパイル

ほとんどの modern プログラミング言語と同様に，Objective Caml は別々にコンパイルされる複数のコンパイル単位へのプログラムの分割を support しています．ひとつのコンパイル単位は，拡張子 .ml をもつ実装ファイルと 拡張子 .mli をもつインターフェースファイルのふたつのファイルから構成されます．各コンパイル単位はモジュールとして見ることができます．name.ml という実装ファイルをコンパイルすると Name というモジュールが定義されます．[1]

モジュールで定義される，値，型，例外は，**ドット記法** (qualified (**限定された？**) **識別子**) と呼ばれる Module.identifier という形，または **open** 構文を使って，参照することができます．

a.ml	b.ml

`type t = { x:int ; y:int } ;;`
`let f c = c.x + c.y ;;`
	`let val = { A.x = 1 ; A.y = 2 } ;;`
	`A.f val ;;`
	`open A ;;`
	`f val ;;`

インターフェースファイル (.mli ファイル) は，このインターフェースに依存するモジュール (つまり，実装ファイルやこのモジュールのクライアント) がコンパイルされる前に ocamlc -c でコンパイルしなければなりません．

もしも，実装ファイルに対しインターフェースファイルがない場合，Objective Caml は，このモジュールは全ての定義をエクスポートしてるとみなします．

実行形式ファイルを生成するためのリンクは 7 章に述べられているように行われます— ocamlc コマンドを，-c オプションなしで，プログラムを構成する全てのコンパイル単位のオブジェクトファイルを引数として並べて起動します．警告: オブジェクトファイル名は依存している順でコマンドラインに並べられなければなりません．すなわち，もし，モジュール B が別のモジュール A を参照しているとしたら，リンカに渡すコマンドライン上で，オブジェクトファイル a.cmo は b.cmo より前に来なければなりません．結果として，ふたつのモジュール間の相互依存は禁止されます．

1. name.ml と Name.ml は，ともに同じモジュール名になります．

例えば，a.ml と b.ml というふたつのソースファイルと，それにマッチするインターフェースファイル a.mli，b.mli から実行形式ファイルを生成するには，次のようにコマンドを起動します.

```
> ocamlc -c a.mli
> ocamlc -c a.ml
> ocamlc -c b.mli
> ocamlc -c b.ml
> ocamlc a.cmo b.cmo
```

インターフェースファイル，実装ファイルひとつずつからなるコンパイル単位は分割コンパイルと情報隠蔽をサポートします．しかし，プログラムを構造化する一般的なツールとしての能力は高くありません．特に，モジュールとファイルの間に一対一の関係があるので，与えられたインターフェースにたいして複数の実装を同時に使ったり，逆に，同じ実装にたいして複数のインターフェースを与えたりするのを妨げています．入れ子のモジュールや，モジュールに関してパラメータ化もサポートされていません．これらの弱点をカバーするために，Objective Caml は特別な文法と構文によるモジュール言語を提供して，言語自身の中でモジュールを操作できるようにしています．本章の残りではこのモジュール言語を紹介します.

モジュール言語

The Objective Caml language features a sub-language for modules, which comes in addition to the core language that we have seen so far. In this module language, the interface of a module is called a **signature** and its implementation is called a **structure**. When there is no ambiguity, we will often use the word “module” to refer to a structure.

The syntax for declaring signatures and structures is as follows:

構文 :

module type *NAME* **=**
  sig
    interface declarations
  end

構文 :

module *Name* **=**
  struct
    implementation definitions
  end

警告 The name of a module *must* start with an uppercase letter. There are no such case restrictions on names of signatures, but by convention we will use names in uppercase for signatures.

Signatures and structures do not need to be bound to names: we can also use anonymous signature and structure expressions, writing simply

構文 : `sig` *declarations* `end`

構文 : `struct` *definitions* `end`

We write *signature* and *structure* to refer to either names of signatures and structures, or anonymous signature and structure expressions.

Every structure possesses a default signature, computed by the type inference system, which reveals all the definitions contained in the structure, with their most general types. When defining a structure, we can also indicate the desired signature by adding a signature constraint (similar to the type constraints from the core language), using one of the following two syntactic forms:

構文 : `module` *Name* `:` *signature* `=` *structure*

構文 : `module` *Name* `= (`*structure* `:` *signature*`)`

When an explicit signature is provided, the system checks that all the components declared in the signature are defined in the structure *structure*, and that the types are consistent. In other terms, the system checks that the explicit signature provided is "included in", or implied by, the default signature. If so, *Name* is viewed in the remainder of the code with the signature "*signature*", and only the components declared in the signature are accessible to the clients of the module. (This is the same behavior we saw previously with interface files.)

Access to the components of a module is via the dot notation:

構文 : $Name_1$`.`$name_2$

We say that the name $name_2$ is **qualified** by the name $Name_1$ of its defining module.

The module name and the dot can be omitted using a directive to **open** the module:

構文 : `open` *Name*

In the scope of this directive, we can use short names $name_2$ to refer to the components of the module *Name*. In case of name conflicts, opening a module hides previously defined entities with the same names, as in the case of identifier redefinitions.

ふたつのスタックモジュール

We continue the example of stacks by recasting it in the module language. The signature for a stack module is obtained by wrapping the declarations from the `stack.mli` file in a signature declaration:

```
# module type STACK =
    sig
      type 'a t
```

```
      exception Empty
      val create: unit → 'a t
      val push: 'a → 'a t → unit
      val pop: 'a t → 'a
      val clear : 'a t → unit
      val length: 'a t → int
      val iter: ('a → unit) → 'a t → unit
    end ;;
module type STACK =
  sig
    type 'a t
    exception Empty
    val create : unit -> 'a t
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
    val clear : 'a t -> unit
    val length : 'a t -> int
    val iter : ('a -> unit) -> 'a t -> unit
  end
```

A first implementation of stacks is obtained by reusing the `Stack` module from the standard library:

```
# module StandardStack = Stack ;;
module StandardStack :
  sig
    type 'a t = 'a Stack.t
    exception Empty
    val create : unit -> 'a t
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
    val top : 'a t -> 'a
    val clear : 'a t -> unit
    val copy : 'a t -> 'a t
    val is_empty : 'a t -> bool
    val length : 'a t -> int
    val iter : ('a -> unit) -> 'a t -> unit
  end
```

We then define an alternate implementation based on arrays:

```
# module MyStack =
    struct
      type 'a t = { mutable sp : int; mutable c : 'a array }
      exception Empty
      let create () = { sp=0 ; c = [||] }
      let clear s = s.sp <- 0; s.c <- [||]
      let increase s x =  s.c <- Array.append s.c (Array.create 5 x)
      let push x s =
        if s.sp >= Array.length s.c then increase s x;
        s.c.(s.sp) <- x;
        s.sp <- succ s.sp
      let pop s =
```

```
        if s.sp =0 then raise Empty
        else (s.sp <- pred s.sp ; s.c.(s.sp))
      let length s = s.sp
      let iter f s = for i = pred s.sp downto 0 do f s.c.(i) done
    end ;;
module MyStack :
  sig
    type 'a t = { mutable sp : int; mutable c : 'a array; }
    exception Empty
    val create : unit -> 'a t
    val clear : 'a t -> unit
    val increase : 'a t -> 'a -> unit
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
    val length : 'a t -> int
    val iter : ('a -> 'b) -> 'a t -> unit
  end
```

These two modules implement the type `t` of stacks by different data types.

```
# StandardStack.create () ;;
- : '_a StandardStack.t = <abstr>
# MyStack.create () ;;
- : '_a MyStack.t = {MyStack.sp = 0; MyStack.c = [||]}
```

To abstract over the type representation in `Mystack`, we add a signature constraint by the `STACK` signature.

```
# module MyStack = (MyStack : STACK) ;;
module MyStack : STACK
# MyStack.create() ;;
- : '_a MyStack.t = <abstr>
```

The two modules `StandardStack` and `MyStack` implement the same interface, that is, provide the same set of operations over stacks, but their `t` types are different. It is therefore impossible to apply operations from one module to values from the other module:

```
# let s = StandardStack.create() ;;
val s : '_a StandardStack.t = <abstr>
# MyStack.push 0 s ;;
Characters 15-16:
  MyStack.push 0 s ;;
                 ^
This expression has type 'a StandardStack.t = 'a Stack.t
but is here used with type int MyStack.t
```

Even if both modules implemented the *t* type by the same concrete type, constraining `MyStack` by the signature `STACK` suffices to abstract over the *t* type, rendering it incompatible with any other type in the system and preventing sharing of values and operations between the various stack modules.

```
# module S1 = ( MyStack : STACK ) ;;
module S1 : STACK
# module S2 = ( MyStack : STACK ) ;;
module S2 : STACK
# let s = S1.create () ;;
val s : '_a S1.t = <abstr>
# S2.push 0 s ;;
Characters 10-11:
  S2.push 0 s ;;
            ^
This expression has type 'a S1.t but is here used with type int S2.t
```

The Objective Caml system compares abstract types by names. Here, the two types `S1.t` and `S2.t` are both abstract, and have different names, hence they are considered as incompatible. It is precisely this restriction that makes type abstraction effective, by preventing any access to the definition of the type being abstracted.

モジュールと情報隠蔽

This section shows additional examples of signature constraints hiding or abstracting definitions of structure components.

型の実装を隠蔽する

Abstracting over a type ensures that the only way to construct values of this type is via the functions exported from its definition module. This can be used to restrict the values that can belong to this type. In the following example, we implement an abstract type of integers which, by construction, can never take the value 0.

```
# module Int_Star =
    ( struct
        type t = int
        exception Isnul
        let of_int = function 0 → raise Isnul | n → n
        let mult = ( * )
      end
    :
      sig
        type t
        exception Isnul
        val of_int : int → t
        val mult : t → t → t
      end
    ) ;;
module Int_Star :
  sig type t exception Isnul val of_int : int -> t val mult : t -> t -> t end
```

値を隠蔽する

We now define a symbol generator, similar to that of page 101, using a signature constraint to hide the state of the generator.

We first define the signature GENSYM exporting only two functions for generating symbols.

```
# module type GENSYM =
    sig
      val reset : unit → unit
      val next : string → string
    end ;;
```

We then implement this signature as follows:

```
# module Gensym : GENSYM =
    struct
      let c = ref 0
      let reset () = c:=0
      let next s = incr c ; s ^ (string_of_int !c)
   end;;
module Gensym : GENSYM
```

The reference `c` holding the state of the generator `Gensym` is not accessible outside the two exported functions.

```
# Gensym.reset();;
- : unit = ()
# Gensym.next "T";;
- : string = "T1"
# Gensym.next "X";;
- : string = "X2"
# Gensym.reset();;
- : unit = ()
# Gensym.next "U";;
- : string = "U1"
# Gensym.c;;
Characters 0-8:
  Gensym.c;;
  ^^^^^^^^
Unbound value Gensym.c
```

The definition of `c` is essentially local to the structure `Gensym`, since it is hidden by the associated signature. The signature constraint achieves more simply the same goal as the local definition of a reference in the definition of the two functions `reset_s` and `new_s` on page 101.

モジュールの複数の「視点」

The module language and its signature constraints support taking several views of a given structure. For instance, we can have a “super-user interface” for the module `Gensym`, allowing the symbol counter to be reset, and a “normal user interface” that permits only the generation of new symbols, but no other intervention on the counter. To implement the latter interface, it suffices to declare the signature:

```
# module type USER_GENSYM =
    sig
      val next : string → string
    end;;
module type USER_GENSYM = sig val next : string -> string end
```

We then implement it by a mere signature constraint.

```
# module UserGensym = (Gensym : USER_GENSYM) ;;
module UserGensym : USER_GENSYM
# UserGensym.next "U" ;;
- : string = "U2"
# UserGensym.reset() ;;
Characters 0-16:
  UserGensym.reset() ;;
  ^^^^^^^^^^^^^^^^
Unbound value UserGensym.reset
```

The `UserGensym` module fully reuses the code of the `Gensym` module. In addition, both modules share the same counter:

```
# Gensym.next "U" ;;
- : string = "U3"
# Gensym.reset() ;;
- : unit = ()
# UserGensym.next "V" ;;
- : string = "V1"
```

モジュール間の型共有

As we saw on page 415, abstract types with different names are incompatible. This can be problematic when we wish to share an abstract type between several modules. There are two ways to achieve this sharing: one is via a special sharing construct in the module language; the other one uses the lexical scoping of modules.

制約による共有

The following example illustrates the sharing issue. We define a module `M` providing an abstract type *M.t*. We then restrict `M` on two different signatures exporting different subsets of operations.

```
# module M =
   (
```

```
    struct
      type t = int ref
      let create() = ref 0
      let add x = incr x
      let get x = if !x>0 then (decr x; 1) else failwith "Empty"
    end
    :
    sig
      type t
      val create : unit → t
      val add : t → unit
      val get : t → int
    end
  ) ;;

# module type S1 =
    sig
      type t
      val create : unit → t
      val add : t → unit
    end ;;

# module type S2 =
    sig
      type t
      val get : t → int
    end ;;
# module M1 = (M:S1) ;;
module M1 : S1
# module M2 = (M:S2) ;;
module M2 : S2
```

As written above, the types *M1.t* and *M2.t* are incompatible. However, we would like to say that both types are abstract but identical. To do this, Objective Caml offers special syntax to declare a type equality over an abstract type in a signature.

構文 : *NAME* **with type** t_1 **=** t_2 **and** . . .

This type constraint forces the type t_1 declared in the signature *NAME* to be equal to the type t_2.

Type constraints over all types exported by a sub-module can be declared in one operation with the syntax

構文 : *NAME* **with module** $Name_1$ **=** $Name_2$

Using these type sharing constraints, we can declare that the two modules M1 and M2 define identical abstract types.

```
# module M1 = (M:S1 with type t = M.t) ;;
module M1 : sig type t = M.t val create : unit -> t val add : t -> unit end
# module M2 = (M:S2 with type t = M.t) ;;
```

```
module M2 : sig type t = M.t val get : t -> int end
# let x = M1.create() in M1.add x ;  M2.get x ;;
- : int = 1
```

共有と入れ子モジュール

Another possibility for ensuring type sharing is to use nested modules. We define two sub-modules (M1 et M2) sharing an abstract type defined in the enclosing module M.

```
# module M =
    ( struct
        type t = int ref
        module M_hide =
          struct
            let create() = ref 0
            let add x = incr x
            let get x = if !x>0 then (decr x; 1) else failwith "Empty"
          end
        module M1 = M_hide
        module M2 = M_hide
      end
     :
      sig
        type t
        module M1 : sig  val create : unit → t  val add : t → unit end
        module M2 : sig  val get : t → int  end
      end ) ;;
module M :
  sig
    type t
    module M1 : sig val create : unit -> t val add : t -> unit end
    module M2 : sig val get : t -> int end
  end
```

As desired, values created by M1 can be operated upon by M2, while hiding the representation of these values.

```
# let x = M.M1.create() ;;
val x : M.t = <abstr>
# M.M1.add x ; M.M2.get x ;;
- : int = 1
```

This solution is heavier than the previous solution based on type sharing constraints: the functions from M1 and M2 can only be accessed via the enclosing module M.

単純なモジュールを拡張する

Modules are closed entities, defined once and for all. In particular, once an abstract type is defined using the module language, it is impossible to add further operations on the abstract type that depend on the type representation without modifying the

module definition itself. (Operations derived from existing operations can of course be added later, outside the module.) As an extreme example, if the module exports no creation function, clients of the module will never be able to create values of the abstract type!

Therefore, adding new operations that depend on the type representation requires editing the sources of the module and adding the desired operations in its signature and structure. Of course, we then get a different module, and clients need to be recompiled. However, if the modifications performed on the module signature did not affect the components of the original signature, the remainder of the program remains correct and does not need to be modified, just recompiled.

パラメータつきモジュール

Parameterized modules are to modules what functions are to base values. Just like a function returns a new value from the values of its parameters, a parameterized module builds a new module from the modules given as parameters. Parameterized modules are also called **functors**.

The addition of functors to the module language increases the opportunities for code reuse in structures.

Functors are defined using a function-like syntax:

構文 : **functor (** *Name* **:** *signature* **)** -> *structure*

```
# module Couple = functor ( Q : sig type t end ) →
    struct type couple = Q.t * Q.t end ;;
module Couple :
  functor (Q : sig type t end) -> sig type couple = Q.t * Q.t end
```

As for functions, syntactic sugar is provided for defining and naming a functor:

構文 : **module** $Name_1$ **(** $Name_2$ **:** *signature* **) =** *structure*

```
# module Couple ( Q : sig type t end ) =  struct type couple = Q.t * Q.t end ;;
module Couple :
  functor (Q : sig type t end) -> sig type couple = Q.t * Q.t end
```

A functor can take several parameters:

構文 :

```
functor ( Name1 : signature1 ) ->
  ⋮
functor ( Namen : signaturen ) ->
  structure
```

The syntactic sugar for defining and naming a functor extends to multiple-argument functors:

構文 :

```
module Name (Name1 : signature1 ) ...( Namen : signaturen ) =
  structure
```

The application of a functor to its arguments is written thus:

構文 :

```
module Name = functor ( structure1 ) ...( structuren )
```

Note that each parameter is written between parentheses. The result of the application can be either a simple module or a partially applied functor, depending on the number of parameters of the functor.

警告 There is no equivalent to functors at the level of signature: it is not possible to build a signature by application of a "functorial signature" to other signatures.

A closed functor is a functor that does not reference any module except its parameters. Such a closed functor makes its communications with other modules entirely explicit. This provides maximal reusability, since the modules it references are determined at application time only. There is a strong parallel between a closed function (without free variables) and a closed functor.

ファンクタとコード再利用

The Objective Caml standard library provides three modules defining functors. Two of them take as argument a module implementing a totally ordered data type, that is, a module with the following signature:

```
# module type OrderedType =
    sig
      type t
      val compare: t → t → int
    end ;;
module type OrderedType = sig type t val compare : t -> t -> int end
```

Function `compare` takes two arguments of type `t` and returns a negative integer if the first is less than the second, zero if both are equal, and a positive integer if the first is greater than the second. Here is an example of totally ordered type: pairs of integers equipped with lexicographic ordering.

```
# module OrderedIntPair =
    struct
      type t = int * int
      let compare (x1,x2) (y1,y2) =
        if x1 < y1 then -1
        else if x1 > y1 then 1
        else if x2 < y2 then -1
        else if x2 > y2 then 1
        else 0
    end ;;
module OrderedIntPair :
  sig type t = int * int val compare : 'a * 'b -> 'a * 'b -> int end
```

The functor `Make` from module `Map` returns a module that implements association tables whose keys are values of the ordered type passed as argument. This module provides operations similar to the operations on association lists from module `List`, but using a more efficient and more complex data structure (balanced binary trees).

```
# module AssocIntPair = Map.Make (OrderedIntPair) ;;
module AssocIntPair :
  sig
    type key = OrderedIntPair.t
    and 'a t = 'a Map.Make(OrderedIntPair).t
    val empty : 'a t
    val add : key -> 'a -> 'a t -> 'a t
    val find : key -> 'a t -> 'a
    val remove : key -> 'a t -> 'a t
    val mem : key -> 'a t -> bool
    val iter : (key -> 'a -> unit) -> 'a t -> unit
    val map : ('a -> 'b) -> 'a t -> 'b t
    val mapi : (key -> 'a -> 'b) -> 'a t -> 'b t
    val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
  end
```

The `Make` functor allows to construct association tables over any key type for which we can write a `compare` function.

The standard library module `Set` also provides a functor named `Make` taking an ordered type as argument and returning a module implementing sets of sets of values of this type.

```
# module SetIntPair = Set.Make (OrderedIntPair) ;;
module SetIntPair :
  sig
    type elt = OrderedIntPair.t
    and t = Set.Make(OrderedIntPair).t
    val empty : t
    val is_empty : t -> bool
    val mem : elt -> t -> bool
    val add : elt -> t -> t
```

```
    val singleton : elt -> t
    val remove : elt -> t -> t
    val union : t -> t -> t
    val inter : t -> t -> t
    val diff : t -> t -> t
    val compare : t -> t -> int
    val equal : t -> t -> bool
    val subset : t -> t -> bool
    val iter : (elt -> unit) -> t -> unit
    val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a
    val for_all : (elt -> bool) -> t -> bool
    val exists : (elt -> bool) -> t -> bool
    val filter : (elt -> bool) -> t -> t
    val partition : (elt -> bool) -> t -> t * t
    val cardinal : t -> int
    val elements : t -> elt list
    val min_elt : t -> elt
    val max_elt : t -> elt
    val choose : t -> elt
  end
```

The type `SetIntPair.t` is the type of sets of integer pairs, with all the usual set operations provided in `SetIntPair`, including a set comparison function `SetIntPair.compare`. To illustrate the code reuse made possible by functors, we now build sets of sets of integer pairs.

```
# module SetofSet = Set.Make (SetIntPair) ;;

# let x = SetIntPair.singleton (1,2) ;;           (* x = { (1,2) }        *)
val x : SetIntPair.t = <abstr>
# let y = SetofSet.singleton SetIntPair.empty ;;  (* y = { {} }           *)
val y : SetofSet.t = <abstr>
# let z = SetofSet.add x y ;;                     (* z = { {(1,2)} ; {} } *)
val z : SetofSet.t = <abstr>
```

The `Make` functor from module `Hashtbl` is similar to that from the `Map` module, but implements (imperative) hash tables instead of (purely functional) balanced trees. The argument to `Hashtbl.Make` is slightly different: in addition to the type of the keys for the hash table, it must provide an equality function testing the equality of two keys (instead of a full-fledged comparison function), plus a hash function, that is, a function associating integers to keys.

```
# module type HashedType =
    sig
      type t
      val equal: t → t → bool
      val hash: t → int
    end ;;
module type HashedType =
  sig type t val equal : t -> t -> bool val hash : t -> int end
```

```
# module IntMod13 =
    struct
      type t = int
      let equal = (=)
      let hash x = x mod 13
    end ;;
module IntMod13 :
  sig type t = int val equal : 'a -> 'a -> bool val hash : int -> int end
# module TblInt = Hashtbl.Make (IntMod13) ;;
module TblInt :
  sig
    type key = IntMod13.t
    and 'a t = 'a Hashtbl.Make(IntMod13).t
    val create : int -> 'a t
    val clear : 'a t -> unit
    val copy : 'a t -> 'a t
    val add : 'a t -> key -> 'a -> unit
    val remove : 'a t -> key -> unit
    val find : 'a t -> key -> 'a
    val find_all : 'a t -> key -> 'a list
    val replace : 'a t -> key -> 'a -> unit
    val mem : 'a t -> key -> bool
    val iter : (key -> 'a -> unit) -> 'a t -> unit
    val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
  end
```

モジュールの局所定義

The Objective Caml core language allows a module to be defined locally to an expression.

構文 :

```
let module Name = structure
  in expr
```

For instance, we can use the `Set` module locally to write a sort function over integer lists, by inserting each list element into a set and finally converting the set to the sorted list of its elements.

```
# let sort l =
    let module M =
       struct
         type t = int
         let compare x y =
           if x < y then -1 else if x > y then 1 else 0
       end
   in
     let module MSet = Set.Make(M)
     in  MSet.elements (List.fold_right MSet.add l MSet.empty) ;;
val sort : int list -> int list = <fun>
```

```
# sort [ 5 ; 3 ; 8 ; 7 ; 2 ; 6 ; 1 ; 4 ] ;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8]
```

Objective Caml does not allow a value to escape a `let module` expression if the type of the value is not known outside the scope of the expression.

```
# let test =
    let module Foo =
      struct
        type t
        let id x = (x:t)
      end
    in Foo.id ;;
Characters 15-101:
  ..let module Foo =
      struct
        type t
        let id x = (x:t)
      end
    in Foo.id...
This ‘let module’ expression has type Foo.t -> Foo.t
In this type, the locally bound module name Foo escapes its scope
```

より大きな例: 銀行口座の管理

We conclude this chapter by an example illustrating the main aspects of modular programming: type abstraction, multiple views of a module, and functor-based code reuse.

The goal of this example is to provide two modules for managing a bank account. One is intended to be used by the bank, and the other by the customer. The approach is to implement a general-purpose parameterized functor providing all the needed operations, then apply it twice to the correct parameters, constraining it by the signature corresponding to its final user: the bank or the customer.

プログラムの構成

The two end modules `BManager` and `CManager` are obtained by constraining the module `Manager`. The latter is obtained by applying the functor `FManager` to the modules `Account`, `Date` and two additional modules built by application of the functors `FLog` and `FStatement`. Figure 14.1 illustrates these dependencies.

モジュールパラメータのシグネチャ

The module for account management is parameterized by four other modules, whose signatures we now detail.

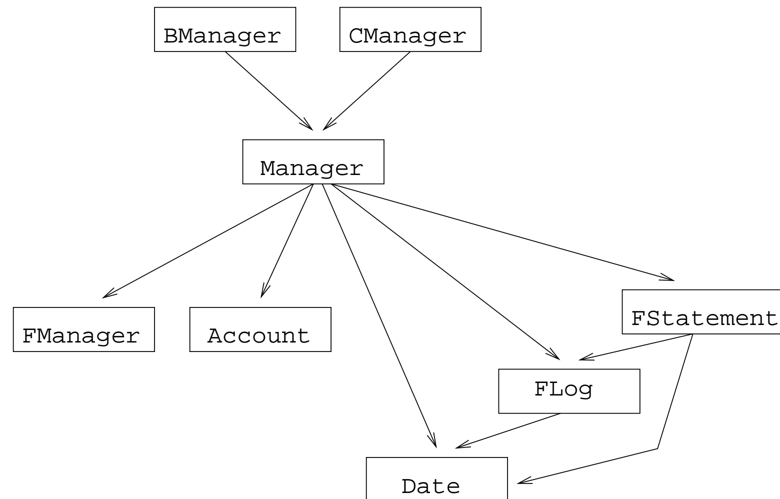


図 14.1: Modules dependency graph.

銀行口座　This module provides the basic operations on the contents of the account.

```
# module type ACCOUNT = sig
    type t
    exception BadOperation
    val create : float → float → t
    val deposit : float → t → unit
    val withdraw : float → t → unit
    val balance : t → float
  end ;;
```

This set of functions provide the minimal operations on an account. The creation operation takes as arguments the initial balance and the maximal overdraft allowed. Excessive withdrawals may raise the `BadOperation` exception.

順序つきの鍵　Operations are recorded in an operation log described in the next paragraph. Each log entry is identified by a key. Key management functions are described by the following signature:

```
# module type OKEY =
    sig
      type t
      val create : unit → t
      val of_string : string → t
      val to_string : t → string
      val eq : t → t → bool
      val lt : t → t → bool
      val gt : t → t → bool
    end ;;
```

The `create` function returns a new, unique key. The functions `of_string` and `to_string` convert between keys and character strings. The three remaining functions are key comparison functions.

ヒストリ Logs of operations performed on an account are represented by the following abstract types and functions:

```
# module type LOG =
    sig
      type tkey
      type tinfo
      type t
      val create : unit → t
      val add : tkey → tinfo → t → unit
      val nth : int → t → tkey*tinfo
      val get : (tkey → bool) → t → (tkey*tinfo) list
    end ;;
```

We keep unspecified for now the types of the log keys (type *tkey*) and of the associated data (type *tinfo*), as well as the data structure for storing logs (type *t*). We assume that new informations added with the `add` function are kept in sequence. Two access functions are provided: access by position in the log (function `nth`) and access following a search predicate on keys (function `get`).

口座明細 The last parameter of the manager module provides two functions for editing a statement for an account:

```
# module type STATEMENT =
     sig
       type tdata
       type tinfo
       val editB : tdata → tinfo
       val editC : tdata → tinfo
     end ;;
```

We leave abstract the type of data to process (*tdata*) as well as the type of informations extracted from the data (*tinfo*).

口座管理用パラメータつきモジュール

Using only the information provided by the signatures above, we now define the general-purpose functor for managing accounts.

```
# module FManager =
   functor (C:ACCOUNT) →
   functor (K:OKEY) →
   functor (L:LOG with type tkey=K.t and type tinfo=float) →
   functor (S:STATEMENT with type tdata=L.t and type tinfo
           = (L.tkey*L.tinfo) list) →
     struct
       type t = { accnt : C.t; log : L.t }
```

```
      let create s d = { accnt = C.create s d; log = L.create() }
      let deposit s g =
        C.deposit s g.accnt ; L.add (K.create()) s g.log
      let withdraw s g =
        C.withdraw s g.accnt ; L.add (K.create()) (-.s) g.log
      let balance g = C.balance g.accnt
      let statement edit g =
        let f (d,i) = (K.to_string d) ^ ":" ^ (string_of_float i)
        in List.map f (edit g.log)
      let statementB = statement S.editB
      let statementC = statement S.editC
    end ;;
module FManager :
  functor (C : ACCOUNT) ->
    functor (K : OKEY) ->
      functor
        (L : sig
               type tkey = K.t
               and tinfo = float
               and t
               val create : unit -> t
               val add : tkey -> tinfo -> t -> unit
               val nth : int -> t -> tkey * tinfo
               val get : (tkey -> bool) -> t -> (tkey * tinfo) list
             end) ->
        functor
          (S : sig
                 type tdata = L.t
                 and tinfo = (L.tkey * L.tinfo) list
                 val editB : tdata -> tinfo
                 val editC : tdata -> tinfo
               end) ->
          sig
            type t = { accnt : C.t; log : L.t; }
            val create : float -> float -> t
            val deposit : L.tinfo -> t -> unit
            val withdraw : float -> t -> unit
            val balance : t -> float
            val statement : (L.t -> (K.t * float) list) -> t -> string list
            val statementB : t -> string list
            val statementC : t -> string list
          end
```

型共有 The type constraint over the parameter `L` of the `FManager` functor indicates that the keys of the log are those provided by the `K` parameter, and that the informations stored in the log are floating-point numbers (the transaction amounts). The type constraint over the `S` parameter indicates that the informations contained in the statement come from the log (the `L` parameter). The signature inferred for the `FManager`

functor reflects the type sharing constraints in the inferred signatures for the functor parameters.

The type `t` in the result of `FManager` is a pair of an account (`C.t`) and its transaction log.

操作 All operations defined in this functor are defined in terms of lower-level functions provided by the module parameters. The creation, deposit and withdrawal operations affect the contents of the account and add an entry in its transaction log. The other functions return the account balance and edit statements.

パラメータの実装

Before building the end modules, we must first implement the parameters to the `FManager` module.

口座 The data structure for an account is composed of a float representing the current balance, plus the maximum overdraft allowed. The latter is used to check withdrawals.

```
# module Account:ACCOUNT =
    struct
     type t = { mutable balance:float; overdraft:float }
     exception BadOperation
     let create b o = { balance=b; overdraft=(-. o) }
     let deposit s c =  c.balance <- c.balance +. s
     let balance c = c.balance
     let withdraw s c =
      let ss = c.balance -. s in
       if ss < c.overdraft then raise BadOperation
       else c.balance <- ss
    end ;;
module Account : ACCOUNT
```

記録用鍵を選ぶ We decide that keys for transaction logs should be the date of the transaction, expressed as a floating-point number as returned by the `time` function from module `Unix`.

```
# module Date:OKEY =
    struct
     type t = float
     let create() = Unix.time()
     let of_string = float_of_string
     let to_string = string_of_float
     let eq = (=)
     let lt = (<)
     let gt = (>)
    end ;;
```

```
module Date : OKEY
```

記録 The transaction log depends on a particular choice of log keys. Hence we define logs as a functor parameterized by a key structure.

```
# module FLog (K:OKEY) =
   struct
    type tkey = K.t
    type tinfo = float
    type t = { mutable contents : (tkey*tinfo) list }
    let create() = { contents = [] }
    let add c i l = l.contents <- (c,i) :: l.contents
    let nth i l = List.nth l.contents i
    let get f l = List.filter (fun (c,_) → (f c)) l.contents
   end ;;
module FLog :
  functor (K : OKEY) ->
    sig
      type tkey = K.t
      and tinfo = float
      and t = { mutable contents : (tkey * tinfo) list; }
      val create : unit -> t
      val add : tkey -> tinfo -> t -> unit
      val nth : int -> t -> tkey * tinfo
      val get : (tkey -> bool) -> t -> (tkey * tinfo) list
    end
```

Notice that the type of informations stored in log entries must be consistent with the type used in the account manager functor.

明細 We define two functions for editing statements. The first (`editB`) lists the five most recent transactions, and is intended for the bank; the second (`editC`) lists all transactions performed during the last 10 days, and is intended for the customer.

```
# module FStatement (K:OKEY) (L:LOG with type tkey=K.t) =
   struct
    type tdata = L.t
    type tinfo = (L.tkey*L.tinfo) list
    let editB h =
     List.map (fun i → L.nth i h) [0;1;2;3;4]
    let editC h =
     let c0 = K.of_string (string_of_float ((Unix.time()) -. 864000.)) in
     let f = K.lt c0 in
      L.get f h
   end ;;
module FStatement :
  functor (K : OKEY) ->
    functor
```

```
(L : sig
       type tkey = K.t
       and tinfo
       and t
       val create : unit -> t
       val add : tkey -> tinfo -> t -> unit
       val nth : int -> t -> tkey * tinfo
       val get : (tkey -> bool) -> t -> (tkey * tinfo) list
     end) ->
sig
  type tdata = L.t
  and tinfo = (L.tkey * L.tinfo) list
  val editB : L.t -> (L.tkey * L.tinfo) list
  val editC : L.t -> (L.tkey * L.tinfo) list
end
```

In order to define the 10-day statement, we need to know exactly the implementation of keys as floats. This arguably goes against the principles of type abstraction. However, the key corresponding to ten days ago is obtained from its string representation by calling the `K.of_string` function, instead of directly computing the internal representation of this date. (Our example is probably too simple to make this subtle distinction obvious.)

目的のモジュール To build the modules `MBank` and `MCustomer`, for use by the bank and the customer respectively, we proceed as follows:

1. define a common "account manager" structure by application of the `FManager` functor;
2. declare two signatures listing only the functions accessible to the bank or to the customer;
3. constrain the structure obtained in 1 with the signatures declared in 2.

```
# module Manager =
   FManager (Account)
            (Date)
            (FLog(Date))
            (FStatement (Date) (FLog(Date))) ;;
module Manager :
  sig
    type t =
      FManager(Account)(Date)(FLog(Date))(FStatement(Date)(FLog(Date))).t = {
      accnt : Account.t;
      log : FLog(Date).t;
    }
    val create : float -> float -> t
    val deposit : FLog(Date).tinfo -> t -> unit
    val withdraw : float -> t -> unit
    val balance : t -> float
```

```
    val statement :
      (FLog(Date).t -> (Date.t * float) list) -> t -> string list
    val statementB : t -> string list
    val statementC : t -> string list
  end

# module type MANAGER_BANK =
     sig
      type t
      val create : float → float → t
      val deposit : float → t → unit
      val withdraw : float → t → unit
      val balance : t → float
      val statementB : t →  string list
     end ;;

# module MBank = (Manager:MANAGER_BANK with type t=Manager.t) ;;
module MBank :
  sig
    type t = Manager.t
    val create : float -> float -> t
    val deposit : float -> t -> unit
    val withdraw : float -> t -> unit
    val balance : t -> float
    val statementB : t -> string list
  end

# module type MANAGER_CUSTOMER =
     sig
      type t
      val deposit : float → t → unit
      val withdraw : float → t → unit
      val balance : t → float
      val statementC : t →  string list
     end ;;

# module MCustomer = (Manager:MANAGER_CUSTOMER with type t=Manager.t) ;;
module MCustomer :
  sig
    type t = Manager.t
    val deposit : float -> t -> unit
    val withdraw : float -> t -> unit
    val balance : t -> float
    val statementC : t -> string list
  end
```

In order for accounts created by the bank to be usable by clients, we added the type constraint on *Manager.t* in the definition of the `MBank` and `MCustomer` structures, to ensure that their *t* type components are compatible.

練習問題

連想リスト

In this first simple exercise, we will implement a polymorphic abstract type for association lists, and present two different views of the implementation.

1. Define a signature `ALIST` declaring an abstract type with two type parameters (one for the keys, the other for the associated values), a creation function, an add function, a lookup function, a membership test, and a deletion function. The interface should be functional, *i.e.* without in-place modifications of the abstract type.
2. Define a module `Alist` implementing the signature `ALIST`
3. Define a signature `ADM_ALIST` for "administrators" of association lists. Administrators can only create association lists, and add or remove entries from a list.
4. Define a signature `USER_ALIST` for "users" of association lists. Users can only perform lookups and membership tests.
5. Define two modules `AdmAlist` and `UserAlist` for administrators and for users. Keep in mind that users must be able to access lists created by administrators.

パラメータつきベクトル

This exercise illustrates the genericity and code reuse abilities of parameterized modules. We will define a functor for manipulating two-dimensional vectors (pairs of (x, y) coordinates) that can be instantiated with different types for the coordinates.

Numbers have the following signature:

```
# module type NUMBER =
   sig
    type a
    type t
    val create : a → t
    val add : t → t → t
    val string_of : t → string
   end ;;
```

1. Define the functor `FVector`, parameterized by a module of signature `NUMBER`, and defining a type *t* of two-dimensional vectors over these numbers, a creation function, an addition function, and a conversion to strings.
2. Define a signature `VECTOR`, without parameters, where the types of numbers and vectors are abstract.
3. Define three structures `Rational`, `Float` et `Complex` implementing the signature `NUMBER`.

4. Use these structures to define (by functor application) three modules for vectors of rationals, reals and complex.

字句解析木

This exercise follows up on the lexical trees introduced in chapter 2, page 62. The goal is to define a generic module for handling lexical trees, parameterized by an abstract type of words.

1. Define the signature `WORD` defining an abstract type *alpha* for letters of the alphabet, and another abstract type *t* for words on this alphabet. Declare also the empty word, the conversion from an alphabet letter to a one-letter word, the accessor to a letter of a word, the sub-word operation, the length of a word, and word concatenation.
2. Define the functor `LexTree`, parameterized by a module implementing `WORD`, that defines (as a function of the types and operations over words) the type of lexical trees and functions `exists`, `insert` et `select` similar to those from chapter 2, page 62.
3. Define the module `Chars` implementing the `WORD` signature for the types *alpha = char* and *t = string*. Use it to obtain a module `CharDict` implementing dictionaries whose keys are character strings.

まとめ

In this chapter, we introduced all the facilities that the Objective Caml module language offers, in particular parameterized modules.

As all module systems, it reflects the duality between interfaces and implementations, here presented as a duality between signatures and structures. Signatures allow hiding information about type, value or exception definitions.

By hiding type representation, we can make certain types abstract, ensuring that values of these types can only be manipulated through the operations provided in the module signature. We saw how to exploit this mechanism to facilitate sharing of values hidden in closures, and to offer multiple views of a given implementation. In the latter case, explicit type sharing annotations are sometimes necessary to achieve the desired behavior.

Parameterized modules, also called functors, go one step beyond and support code reuse through simple mechanisms similar to function abstraction and function application.

もっと学びたい人のために

Other examples of modules and functors can be found in chapter 4 of the Objective Caml manual.

The underlying theory and the type checking for modules can be found in a number of research articles and course notes by Xavier Leroy, at

リンク: http://cristal.inria.fr/~xleroy

The Objective Caml module system follows the same principles as that of its cousin the SML language. Chapter 22 compares these two languages in more details and provides bibliographical references for the interested reader.

Other languages feature advanced module systems, in particular Modula-3 (2 and 3), and ADA. They support the definition of modules parameterized by types and values.

15

オブジェクト指向プログラミング

言語名にも示されているように Objective Caml はオブジェクト指向プログラミングの機能を持っています。命令の逐次実行によって進んでいく手続き型プログラミングや必要な計算から簡約していく関数型プログラミングとは異なり、オブジェクト指向プログラミングはデータ駆動型と考えることができます。オブジェクトは、関連するデータの集合を系統立てる新しい方法を提供しています。まず、データと操作を一まとめにするクラスがあります。操作は、メソッドとも呼ばれますが、オブジェクトの取り得る振る舞いを定義します。メソッドはオブジェクトにメッセージを送ることによって起動されます。オブジェクトはメッセージを受け取るとそのメッセージによって指定されたメソッドに対応する行動、計算を開始します。この働きは関数に引数を適用することとは違っています。なぜなら実際に実行されることになるコードを決めるのはオブジェクト自身だからです。オブジェクトへ送られたメッセージにはメソッドの名前が含まれています。名前とコードの間のこのような遅延束縛の働きによって、柔軟で再利用性の高いプログラムを記述することができます。

オブジェクト指向プログラミングでは**集約** や **継承** などのクラスの間の関係を定義することができます。クラスはオブジェクト同士がメッセージによって通信する仕方を定義します。クラス同士の関係はソフトウェアをモデル化する新しい手段を提供しています。あるクラスを継承したクラスは、親クラスの定義をすべて含んでいますが、データやメソッドを拡張することで、型による制約で許された範囲で新しい振る舞いを再定義することもできます。この本ではクラス間の関係を絵と記号によって表現します。[1]

Objective Caml のオブジェクト指向の機能は型システムと統合されています。クラス宣言は同じ名前の型を定義することになります。二種類の多相性を同時に使うことができます。一つはパラメータ型多相性ですが、これはすでにパラメータ化された型として紹介しました。型パラメータはクラスに対しても利用できます。もう一つは **包含的多相性** と呼ばれているもので、オブジェクトと遅延束縛の間の部分型関係を利用しています。クラス sc の型がクラス c の部分型である時に、クラス sc のオブジェクトは クラス c のオブジェクトの代わりに使うことができます。部分型に関する制約は明示的に記述する

1. クラス間の関係の表現には UML (*Unified Modeling Language*) など様々な記法が提案されている。

必要があります。包含的多相性を使うと、要素の型が同一でない、ある特定の型の部分型になっている非均一リストを作ることができます。遅延束縛のおかげでそのような非均一リストのすべての要素に同じメッセージを送り、要素の実際のクラスに応じた別々のメソッドを起動させることができます。

一方で Objective Caml にはメソッドのオーバーロードという概念はありません。オーバーロードとは同じ名前のメソッドを複数定義することができる機能です。オーバーロードがあると自動的に型推論するのが困難な場合があり、プログラマから情報を補ってもらう必要があります。

Objective Caml はパラメータ型多相性と包含的多相性の両方を持ち、型推論を行う静的な型システムを持つ唯一の言語であることを強調しておきましょう。

この章のあらまし

この章では Objective Caml のオブジェクト指向のための拡張機能について記述します。この機能はこれまでの章で既に解説した言語機能を制限することはありません。オブジェクト指向のためにいくつかの新しいキーワードが追加されます。

最初の節ではクラス宣言の構文、オブジェクト生成、メッセージ送信について記述します。第二節ではクラスの間に作られる関係について説明します。第三節では**オブジェクト型**の概念を明らかにし、抽象クラス、多重継承、型パラメータを持つクラスの豊かな表現力について解説します。第四節では部分型と包含的多相性の能力を示します。第五節では関数型スタイルのオブジェクト指向プログラミングについて扱います。関数型スタイルではオブジェクトの内部状態は更新されず、更新された状態を持った新しいオブジェクトが返されます。第六節ではインターフェースや、クラス変数を作れる局所宣言などの残りの機能について解説します。

クラス、オブジェクト、メソッド

Objective Caml のオブジェクト指向の拡張機能は、言語の関数型と手続き型の核と型システムも含めて統合されています。この、型システムとの統合という点がこの言語の他の言語にない特徴と言えるでしょう。すなわちこの言語は静的に型付けられた、型推論を持つ、オブジェクト指向言語なのです。この拡張機能はクラスとオブジェクトの定義、(多重継承を含む) クラス継承、パラメータ化されたクラス、抽象クラスを含んでいます。クラスのインターフェースはクラス定義から自動的に生成されますが、モジュールと同様にプログラマが署名を書けばより正確なインターフェースを与えることもできます。

オブジェクト指向についての用語

オブジェクト指向プログラミングに関する基本的な用語を以下の表にまとめます。

クラス： **クラス**には、そのクラスに属するオブジェクトの内容が記述されています。内容とは**インスタンス変数**と呼ばれるデータと**メソッド**と呼ばれる操作の集まりからなります。

オブジェクト： オブジェクトとはクラスの要素（インスタンス）です。オブジェクトは、その属するクラスによって定義された振る舞いを持っています。オブジェクトとはプログラムの実際の構成要素であり、一方クラスはオブジェクトがどのように作られ、振る舞うか仕様を与えます。

メソッド： メソッドとはオブジェクトが行うことができる振る舞いのことです。

メッセージ送信 オブジェクトへの **メッセージ送信** とはメソッドを実行すなわち **起動** させるよう要求することです。

クラス宣言

クラスを定義する最も簡単な構文は以下のようなものです。この章ではこの構文を拡張させながら解説を行います。

構文：

```
class クラス名 p1 ... pn =
  object
     ⋮
    インスタンス変数
     ⋮
    メソッド
     ⋮
  end
```

$p_1, \ldots, p_n$ はこのクラスの構築子へのパラメータです。パラメータがない場合は省略できます。

インスタンス変数は以下のように宣言します。

構文：

```
val 変数名 = 式
または
val mutable 変数名 = 式
```

インスタンス変数を **mutable** と宣言した時はその値を変更することができます。そうでない場合はオブジェクトが生成される時に式を評価した値に固定されます。

メソッドは以下のように宣言します。

構文：

```
method メソッド名 p1 ... pn = 式
```

val と **method** 以外の構文もクラス宣言の中で許されていますが、それらは必要に応じて紹介していきます。

クラスの例　例としてやはり point クラスを使うことにします。

- インスタンス変数 x と y が点の座標を表します。
- メソッド get_x と get_y によって座標を知ることができます。
- 移動のためのメソッド（moveto 絶対座標、rmoveto 相対座標）があります。
- データを *string* として表示するメソッド（to_string）があります。
- 原点からの距離を求めるメソッド（distance）があります。

```
# class point (x_init,y_init) =
    object
      val mutable x = x_init
      val mutable y = y_init
      method get_x = x
      method get_y = y
      method moveto (a,b) =  x <- a ; y <- b
      method rmoveto (dx,dy) =  x <- x + dx ; y <- y + dy
      method to_string () =
          "( " ^ (string_of_int x) ^ ", " ^ (string_of_int y) ^")"
      method distance () = sqrt (float(x*x + y*y))
    end ;;
```

get_x や get_y などのメソッドではパラメータが必要ありません。インスタンス変数にアクセスするためのメソッドはパラメータを取らないのが普通です。

point クラスを宣言すると、システムは次のようなメッセージを表示します。

```
class point :
  int * int ->
  object
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method rmoveto : int * int -> unit
    method to_string : unit -> string
  end
```

このメッセージには二種類の情報が含まれています。まずこのクラスのオブジェクトの型を表しています。（実際には「*point* 型」として表示されます。）オブジェクトの型はクラスに含まれるインスタンス変数とメソッドの型のリストになります。この例では *point* 型とは次のような型を省略したものです。

```
< distance : unit → unit; get_x : int; get_y : int;
  moveto : int * int → unit; rmoveto : int * int → unit;
  to_string : unit → unit >
```

次に point クラスの構築子の情報があります。構築子の型は *int*int* --> *point* となります。この構築子は整数の組（点の初期座標を意味する）を受け取り、point オブジェクトを生成します。構築子を呼び出すためにはキーワード **new** を使います。

クラスの型を次のように定義することもできます。

```
# type simple_point = < get_x : int; get_y : int; to_string : unit → unit > ;;
```

```
type simple_point = < get_x : int; get_y : int; to_string : unit -> unit >
```

注意

point 型はクラス宣言の情報をすべて含んでいません。インスタンス変数は型には示されていません。メソッドだけがインスタンス変数にアクセスすることができます。

警告 クラス宣言は型宣言でもある。したがって、未束縛の型変数を含んでいてはならない。

この点については後で型制約（458 ページ）と型パラメータを持つクラス（464 ページ）を扱う時にまた説明します。

クラスの図表現

Objective Caml の型を表現するために UML 記法を採用します。クラスは三つの部分からなる長方形によって示されます。

- 上部はクラスの名前を示す。
- 中段部は属性（インスタンス変数）のリストを表す。
- 下部はメソッドのリストを表す。

図 15.1 に `caml` クラスの図表現の例を示します。

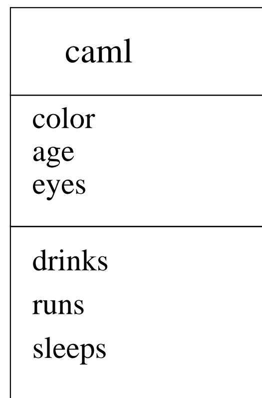


図 15.1: クラスの図表現

インスタンス変数やメソッドの型情報が図中に加えられることもあります。

インスタンス生成

オブジェクトとはクラスの値で、インスタンスと呼ばれます。インスタンスは総称的な演算子である new によって生成されます。new 演算子は、クラス名と初期化のための値を引数として取ります。

構文 : **new** クラス名 式$_1$ ... 式$_n$

以下に *point* クラスのインスタンスを、別々の初期値から生成する例を示します。

```
# let p1 = new point (0,0);;
val p1 : point = <obj>
# let p2 = new point (3,4);;
val p2 : point = <obj>
# let coord = (3,0);;
val coord : int * int = (3, 0)
# let p3 = new point coord;;
val p3 : point = <obj>
```

Objective Caml ではクラスの構築子は一つしか定義できませんが、ユーザ独自の生成関数を定義することができます。

```
# let make_point x = new point (x,x) ;;
val make_point : int -> point = <fun>
# make_point 1 ;;
- : point = <obj>
```

メッセージ送信

オブジェクトにメッセージを送るには # 演算子を使います。[2]

構文 : obj_1**#**メソッド名 p_1 ... p_n

$p_1, \ldots, p_n$ という引数を持ったメッセージがオブジェクトの指定されたメソッドに送られます。オブジェクトの属するクラスでは指定されたメソッドが定義されていなくてはなりません。すなわち、そのメソッドはオブジェクトの型に現われている必要があります。引数の型もメソッドの仮引数の型に適合する必要があります。以下に point クラスのオブジェクトに対するメッセージ送信の例を示します。

```
# p1#get_x;;
- : int = 0
# p2#get_y;;
- : int = 4
# p1#to_string();;
- : string = "( 0, 0)"
# p2#to_string();;
- : string = "( 3, 4)"
```

2. ほとんどのオブジェクト指向言語ではドット記法が使われている。しかしドット記法は既にレコードとモジュールで使われているため別の記号を使う必要があった。

```
# if (p1#distance()) = (p2#distance())
  then print_string ("That's just chance\n")
  else print_string ("We could bet on it\n");;
We could bet on it
- : unit = ()
```

型の観点からいえば *point* 型のオブジェクトも、他の値とまったく同じように Objective Caml の多相型関数で取り扱うことができます。

```
# p1 = p1 ;;
- : bool = true
# p1 = p2;;
- : bool = false
# let l = p1::[];;
val l : point list = [<obj>]
# List.hd l;;
- : point = <obj>
```

警告 オブジェクトの等価性は物理的等価性として定義されています。

この点については部分型関係について解説する時（474 ページ）に詳しく説明します。

クラスの間の関係

クラス同士には二種類の関係を定義することができます。

1. **Has-a** と呼ばれる集約関係
   C_2 クラスが C_1 クラスと *Has-a* 関係を持っているとは、C_2 クラスが、型が C_1 クラスであるインスタンス変数を持っていることです。この関係は、少なくとも一つのインスタンス変数を持っている時に成り立ちます。
2. **Is-a** と呼ばれる継承関係
   C_2 クラスが C_1 クラスのサブクラスであるとは、C_2 クラスが C_1 クラスの振る舞いを拡張したものである、ということを意味しています。オブジェクト指向プログラミングの大きな利点の一つは、ある既存のクラスのコードを再利用しながらその振る舞いを再定義できる能力にあります。クラスを拡張すると、新しいクラスは元のクラスのすべてのインスタンス変数とメソッドを継承します。

集約

C_1 クラスが C_2 クラスを集約するとは、C_1 クラスの少なくとも一つのインスタンス変数の型が C_2 であることを言います。同じ型のインスタンス変数を持っている数が分かる場合にはその数を付け加えることもあります。

集約の例

点の集合としての画像を定義してみましょう。そのために *picture* クラス（図 15.2 参照）を宣言します。*picture* クラスは *point* クラスの配列を持っています。すなわち *picture* クラスは *point* クラスを一般化された **Has-a** 関係を利用して集約しています。

```
# class picture n =
    object
      val mutable ind = 0
      val tab = Array.create n (new point(0,0))
      method add p =
        try  tab.(ind)<-p ; ind <- ind + 1
        with Invalid_argument("Array.set")
                → failwith ("picture.add:ind =" ^ (string_of_int ind))
      method remove () = if (ind > 0) then ind <-ind-1
      method to_string () =
        let s = ref "["
        in for i=0 to ind-1 do s:= !s ^ " " ^ tab.(i)#to_string() done ;
           (!s) ^ "]"
    end ;;
class picture :
  int ->
  object
    val mutable ind : int
    val tab : point array
    method add : point -> unit
    method remove : unit -> unit
    method to_string : unit -> string
  end
```

画像を作り上げるには *picture* クラスのインスタンスを生成し、必要に応じて点を挿入します。

```
# let pic = new picture 8;;
val pic : picture = <obj>
# pic#add p1; pic#add p2; pic#add p3;;
- : unit = ()
# pic#to_string ();;
- : string = "[ ( 0, 0) ( 3, 4) ( 3, 0)]"
```

集約の図表現

picture クラスと *point* クラスの間の関係はグラフを利用して図 15.2 のように表すことができます。根本に菱型が付いた矢印は集約関係を意味しています。この例では *picture* クラスは 0 個以上の点を持っていますので図中の矢印にその数を表記します。

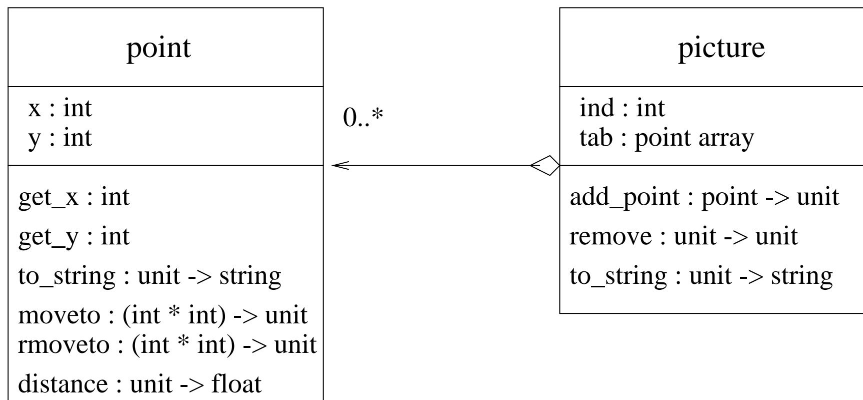


図 15.2: 集約関係

継承関係

これはオブジェクト指向プログラミングで基本となる関係です。c2 クラスが c1 クラスを継承すると、c2 クラスは親クラスのすべてのインスタンス変数とメソッドを継承します。c2 クラスは新しいインスタンス変数やメソッドを定義することもできますし、継承したものを自分の都合のために再定義することもできます。親クラスはまったく無変更のままですから、親クラスを利用するアプリケーションを新しく定義したクラスに合わせて変更する必要はありません。

継承についての構文は次のようになります。

構文 : **inherit** 名前 $_1$ $p_1 \ldots p_n$ [**as** 名前 $_2$]

p_1 から p_n までの引数は 名前 $_1$ クラスの構築子に与える必要があるものです。親クラスのメソッドにアクセスしたい時には **as** キーワードを使って親クラスへの参照を受け取る変数を指定します。この機能は親クラスのメソッドを再定義したときにとりわけ役に立ちます。詳しくは 449 ページを参照してください。

単一継承の例

古典的な例にならって *point* クラスに色を表す新しい属性を付加した新しいクラスを継承によって定義してみましょう。色は *string* 型を持つ c というインスタンス変数によって表現します。点の色情報を問い合わせるメソッド get_color を追加します。最後に文字列に変換する関数を新しい属性を認識できるように再定義します。

注意

メソッド to_string の中の変数 x と y はインスタンス変数であって初期化引数ではありません。

```
# class colored_point (x,y) c =
    object
```

```
      inherit point (x,y)
      val mutable c = c
      method get_color = c
      method set_color nc = c <- nc
      method to_string () = "( " ^ (string_of_int x) ^
                            ", " ^ (string_of_int y) ^ ")" ^
                            " [" ^ c ^ "] "
    end ;;
class colored_point :
  int * int ->
  string ->
  object
    val mutable c : string
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method get_color : string
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method rmoveto : int * int -> unit
    method set_color : string -> unit
    method to_string : unit -> string
  end
```

colored_point クラスの構築子の引数は、*point* クラスの構築子のために必要な座標と新しいクラスのために必要な色属性になります。

継承されたメソッド、新しく定義されたメソッドおよび再定義されたメソッドはこのクラスのインスタンスの振る舞いに期待通りよく適合しています。

```
# let pc = new colored_point (2,3) "white";;
val pc : colored_point = <obj>
# pc#get_color;;
- : string = "white"
# pc#get_x;;
- : int = 2
# pc#to_string();;
- : string = "( 2, 3) [white] "
# pc#distance;;
- : unit -> float = <fun>
```

このとき *point* クラスは *colored_point* クラスの**親クラス**であると言い、逆に *colored_point* クラスは *point* クラスの **子クラス** であると言います。

警告 メソッドを子クラスで再定義するときは、親クラスでのメソッドの型を尊重しなければならない。

継承の図表現

クラスの間の継承関係は子クラスから親クラスへの矢印として表示されます。矢印の先端は閉じた三角形になっています。継承の図表現では子クラスで新しく追加されたインスタ

ンス変数とメソッド、再定義されたメソッドだけを表示します。図 15.3 は *colored_point* クラスとその親である *point* の関係を表しています。

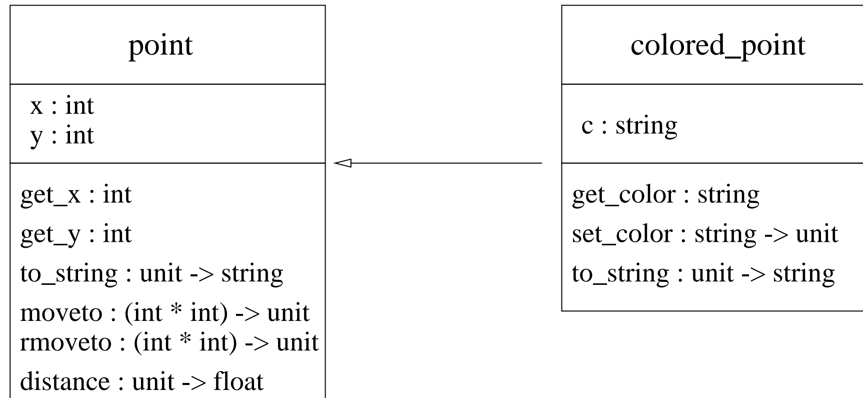


図 15.3: 継承関係

新しいメソッドを追加したため *colored_point* 型は *point* 型とは異なっています。これらのクラスのインスタンスの間での等価性判定はそれぞれのクラスの型の違いを表示するために長いエラーメッセージを表示します。

```
# p1 = pc;;
Characters 6-8:
  p1 = pc;;
       ^^
This expression has type
  colored_point =
    < distance : unit -> float; get_color : string; get_x : int; get_y :
      int; moveto : int * int -> unit; rmoveto : int * int -> unit;
      set_color : string -> unit; to_string : unit -> string >
but is here used with type
  point =
    < distance : unit -> float; get_x : int; get_y : int;
      moveto : int * int -> unit; rmoveto : int * int -> unit;
      to_string : unit -> string >
Only the first object type has a method get_color
```

その他のオブジェクト指向の機能

特別な参照 `self` と `super`

クラスにメソッドを定義する時に親クラスのメソッドを起動することができると便利な場合があります。このために Objective Caml では親クラス（のオブジェクト）への参照に名前が付けることができます。同様にオブジェクト自身への参照にも名前を付けて参照することができます。その場合、オブジェクト自身の参照名は **`object`** というキー

ワードの後に指定します。親クラスへの参照に名前を付けるのは継承関係を宣言する時に行います。

例えば *colored_point* クラスの to_string メソッドを定義するには親クラスの to_string メソッドの振る舞いを利用した方が便利でしょう。

```
# class colored_point (x,y) c  =
    object (self)
      inherit point (x,y) as super
      val c = c
      method get_color = c
      method to_string () = super#to_string() ^  " [" ^ self#get_color ^ "] "
    end ;;
```

親クラスと子クラスのオブジェクトへの参照には好きな名前を付けられますが、慣習として現在のオブジェクトへの参照には self または this が、親クラスへの参照には super という名前が良く使われています。多重継承を行った時は複数の親クラスを区別するために別々の名前を付けた方が分かりやすいかもしれません。(461 ページ参照。)

警告 もし親クラスへの参照と同じ名前の変数を宣言した場合には新しい変数が親クラスへの参照を隠蔽するため親クラスへ参照することができなくなります。

遅延束縛

遅延束縛 の働きによって実際にメッセージが送られるオブジェクトは実行時に決定されます。これはコンパイル時に決定される静的束縛と対立する概念です。Objective Caml ではメソッドの選択に遅延束縛が採用されています。したがって実際に実行されることになるコードはメッセージを受け取るオブジェクトによって実行時に決定されます。

これまでに見て来た *colored_point* クラスでは to_string メソッドを再定義しています。この新しいメソッドの定義では get_color メソッドを呼び出しています。さてここで *colored_point* を継承した別の新しいクラス *colored_point_1* を定義することにしましょう。この新しいクラスでは get_color メソッドを再定義します。(色を意味する文字列が正しいかどうかチェックします。) ただし to_string メソッドは再定義しないことにします。

```
# class colored_point_1 coord c =
    object
      inherit colored_point coord  c
      val true_colors = ["white"; "black"; "red"; "green"; "blue"; "yellow"]
      method get_color = if List.mem c true_colors then c else "UNKNOWN"
    end ;;
```

色付き点を表すどちらのクラスでも to_string メソッドは同じですが、それぞれのクラスから生成されたオブジェクトは異なった振る舞いを示しています。

```
# let p1 = new colored_point (1,1) "blue as an orange" ;;
val p1 : colored_point = <obj>
```

```
# p1#to_string();;
- : string = "( 1, 1) [blue as an orange] "
# let p2 = new colored_point_1 (1,1) "blue as an orange" ;;
val p2 : colored_point_1 = <obj>
# p2#to_string();;
- : string = "( 1, 1) [UNKNOWN] "
```

to_string メソッドの中の get_color の束縛は *colored_point* クラスがコンパイルされた時点では固定されていません。get_color メソッドが呼び出されたときに実際に実行されるコードは *colored_point* クラスと *colored_point_1* クラスのインスタンスに関連付けられているメソッドから決定されます。たとえば *colored_point* クラスのインスタンスに to_string メッセージを送ると、*colored_point* クラスで定義されている get_color が実行されます。一方、同じメッセージを *colored_point_1* のインスタンスに送ると親クラスで定義されている to_string メソッドが実行されますが、その後、子クラスに定義されている get_color メソッドが実行され、色を表現している文字列が適切かどうか判定されることになります。

オブジェクトの内部表現とメッセージの発送

オブジェクトは可変の部分と不変の部分の二つから構成されています。可変部にはレコードとまったく同様にインスタンス変数が含まれています。不変部はメソッド表に対応し、あるクラスのすべてのインスタンスで共有されています。

メソッド表とは関数がところどころに入っているまばらな配列です。一つのアプリケーションの中のすべてのメソッドには重ならない番号が割り振られていて、その番号がメソッド表の索引として使われています。Objective Caml では次のような機械語の存在を仮定しています。それはオブジェクト o と索引 n を受け取り、そのオブジェクトのメソッド表の指定された索引に登録されている関数を返す GETMETHOD(o,n) 命令です。GETMETHOD(o,n) 命令を呼び出した結果を f_n と書くことにします。o#m というメッセージ送信をコンパイルするには、まず m メソッドに該当する索引 n を求め、GETMETHOD(o,n) をオブジェクト o に適用するコードを生成します。これはすなわち f_n 関数をオブジェクト o に適用することに対応しています。遅延束縛は実行時の GETMETHOD の計算の中に実装されています。

メソッドの中でメッセージを self に送るのも同様にメッセージに対応する索引を検索し、メソッド表の中で見つかった関数を呼び出すコードへとコンパイルされます。

継承の場合でも、同じメソッド名は常に同じ索引に対応しているのでメソッドの再定義を気にせずに、再定義が反映されている新しいメソッド表を使って全く同じコードが生成されます。したがって point クラスのインスタンスへ to_string メッセージを送る時は座標を文字列に変換する処理が行われますが、一方 colored_point クラスのインスタンスへ同じメッセージを送る場合は同じ索引を使って色属性を認識するように再定義されたメソッドに対応する関数が選択されるでしょう。

索引の不変性のおかげで部分型（470 ページ参照）も実行に関して矛盾が生じないことが保証されています。もちろん colored_point クラスのインスタンスが明示的に point 型と指定されている場合、to_string メッセージを送ると point クラスのメソッド表の索引が使われます。しかし、その索引は colored_point クラスの索引と同一になるよう定義

されているので、実際に呼び出されるメソッドは、メッセージを受け取ったインスタンスのメソッド表の指定された索引に結びつけられているメソッド、すなわち colored_point クラスの to_string メソッドが起動されます。

もちろん Objective Caml の実際の実装は違いますが、呼び出されるメソッドの動的検索の原則はこれまでに説明したものと変わりません。

初期化

オブジェクト生成の間に実行されるコードを指定するために **initializer** キーワードをクラス定義の中で利用することができます。初期化コードの中ではメソッドの中で許されている計算ならばどんな計算でも行うことができます。

構文 : **initializer** 式

point クラスをまた拡張してみましょう。今度はインスタンスが生成されたことを報告する verbose point を定義してみましょう。

```
# class verbose_point p =
    object(self)
      inherit point p
      initializer
        let xm = string_of_int x and ym = string_of_int y
        in Printf.printf ">> Creation of a point at (%s %s)\n"
                         xm ym ;
           Printf.printf "   , at distance %f from the origin\n"
                         (self#distance()) ;
    end ;;

# new verbose_point (1,1);;
>> Creation of a point at (1 1)
   , at distance 1.414214 from the origin
- : verbose_point = <obj>
```

初期化コードのおもしろく、ためになる利用例は継承されたクラスのインスタンス生成時の動作を追跡することでしょう。例えばこのようになります。

```
# class c1 =
    object
      initializer print_string "Creating an instance of c1\n"
    end ;;

# class c2 =
    object
      inherit c1
      initializer print_string "Creating an instance of c2\n"
    end ;;

# new c1 ;;
Creating an instance of c1
- : c1 = <obj>
```

```
# new c2 ;;
Creating an instance of c1
Creating an instance of c2
- : c2 = <obj>
```

クラス *c2* のインスタンスを生成する前にまず親クラスの生成を行っていることが分かるでしょう。

プライベートメソッド

メソッドに **private** キーワードを付けて宣言すると**プライベートメソッド** になります。プライベートメソッドはそのクラスのインターフェースには存在しますが、そのクラスのインスタンスには存在しません。プライベートメソッドは他のメソッドからのみ呼び出すことができます。そのクラスのインスタンスに対してプライベートメソッドを呼び出すことはできません。しかしプライベートメソッドを継承することは可能です。したがってクラス階層の定義の中で使うことはできます[3]。

構文 : **method private** 名前 = 式

point クラスを拡張してみましょう。最後の移動を取り消す undo メソッドを追加します。そのためには動く前の座標を覚えておく必要があります。このため新しいインスタンス変数 old_x と old_y とこららの変数を更新するメソッドを新しく定義します。ユーザにはこのメソッドを直接呼び出してほしくないのでプライベート宣言をします。moveto メソッドと rmoveto メソッドを再定義し、移動のための古いメソッドを呼び出す前に現在の座標を覚えておくようにします。

```
# class point_m1 (x0,y0) =
    object(self)
      inherit point (x0,y0) as super
      val mutable old_x = x0
      val mutable old_y = y0
      method private mem_pos () = old_x <- x ; old_y <- y
      method undo () = x <- old_x; y <- old_y
      method moveto (x1, y1) = self#mem_pos () ; super#moveto (x1, y1)
      method rmoveto (dx, dy) = self#mem_pos () ; super#rmoveto (dx, dy)
  end ;;
class point_m1 :
  int * int ->
  object
    val mutable old_x : int
    val mutable old_y : int
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method get_x : int
    method get_y : int
    method private mem_pos : unit -> unit
    method moveto : int * int -> unit
    method rmoveto : int * int -> unit
```

3. Objective Caml の **private** は Objective C、C++、Java の **protected** に対応しています。

```
    method to_string : unit -> string
    method undo : unit -> unit
  end
```

point_m1 型の表示の中で mem_pos メソッドの前に **private** キーワードが付けられています。このメソッドは別のメソッドからは呼び出すことができますが、たとえ同じクラスであっても他のインスタンスからは呼び出すことはできません。この条件はインスタンス変数の場合と全く同様です。インスタンス変数 old_x と old_y はコンパイルされた結果の中には表示されていますが、他のインスタンスから直接アクセスすることはできません（441 ページ参照）。

```
# let p = new point_m1 (0, 0) ;;
val p : point_m1 = <obj>
# p#mem_pos() ;;
Characters 0-1:
  p#mem_pos() ;;
  ^
This expression has type point_m1
It has no method mem_pos
# p#moveto(1, 1) ; p#to_string() ;;
- : string = "( 1, 1)"
# p#undo() ; p#to_string() ;;
- : string = "( 0, 0)"
```

警告 型による制約のためにプライベート宣言されたメソッドがパブリックになることがある。

型と総称性

集約関係と継承関係を利用して問題をモデル化する能力に加えて、オブジェクト指向プログラミングには既存のクラスの振る舞いを再利用、変更できる興味深い能力があります。しかし Objective Caml ではクラスを拡張する時には静的な型付けに関する制約を守る必要があります。

抽象クラスを使うとコードを機能分解し、複数のサブクラスが一つの「コミュニケーションプロトコル」を守るように制約を与えることができます。抽象クラスは子クラスのインスタンスが受け取る可能性のあるメッセージの名前と型を固定します。この技法は多重継承と関連させて理解するとより実感できると思います。

開いたオブジェクト型（あるいは単に**開いた型**）の概念は総称的なメソッドを使ったプログラムを正しく機能させることができます。しかし、型に関する制約を明示的に正確に指定する必要が生じるかもしれません。これはとりわけ型パラメータを持つクラスでは必要になるでしょう。型パラメータを持つクラスは、クラスに対してパラメータ型多相性を提供するものです。Objective Caml ではオブジェクト指向拡張部分でもこのような機能を持っているおかげで真に総称的な言語と言えるのです。

抽象クラスと抽象メソッド

抽象クラスの中ではメソッドを本体なしに宣言することができます。そのようなメソッドを抽象メソッドと呼びます。抽象クラスのインスタンスを生成することは違法です。つまり **new** することができません。抽象クラス、抽象メソッドであると指定するには **virtual** キーワードを使います。

構文： **class virtual** 名前 **= object** ... **end**

抽象メソッドを含むクラスは**必ず**抽象メソッドと宣言しなければなりません。抽象メソッドは型のみを指定して宣言します。

構文： **method virtual** 名前：型

抽象クラスのサブクラスが親の抽象メソッドを**すべて**再定義している時そのクラスは通常の具体的なクラスとなります。そうでない時はその子クラスも抽象クラスとして宣言しなければなりません。

例として表示可能なオブジェクトの集合を作ってみましょう。表示可能なオブジェクトとはそのオブジェクトの内容を文字列に変換して表示する print メソッドを持っているオブジェクトです。そのようなオブジェクトには to_string メソッドが必要です。まず printable クラスを定義します。オブジェクトの内容を表す文字列はそのオブジェクトの性質に依存するので printable クラスの to_string メソッドは抽象メソッドになります。結果として printable クラスも抽象クラスになります。

```
# class virtual printable () =
    object(self)
      method virtual to_string : unit → string
      method print () = print_string (self#to_string())
    end ;;
class virtual printable :
  unit ->
  object
    method print : unit -> unit
    method virtual to_string : unit -> string
  end
```

このクラスと to_string メソッドが抽象的なのは型表示にもはっきり示されています。

このクラスから図 15.4 のクラス階層を定義してみましょう。

point クラス、*colored_point* クラス、*picture* クラスを再定義するのは簡単です。これらのクラスの宣言に inherit printable () という行を付け加えれば完了です。継承を通して *print* メソッドが各クラスに提供されます。

```
# let p = new point (1,1) in p#print() ;;
( 1, 1)- : unit = ()
# let pc = new colored_point (2,2) "blue" in pc#print() ;;
( 2, 2) with color blue- : unit = ()
# let t = new picture 3 in t#add (new point (1,1)) ;
                             t#add (new point (3,2)) ;
                             t#add (new point (1,4)) ;
```

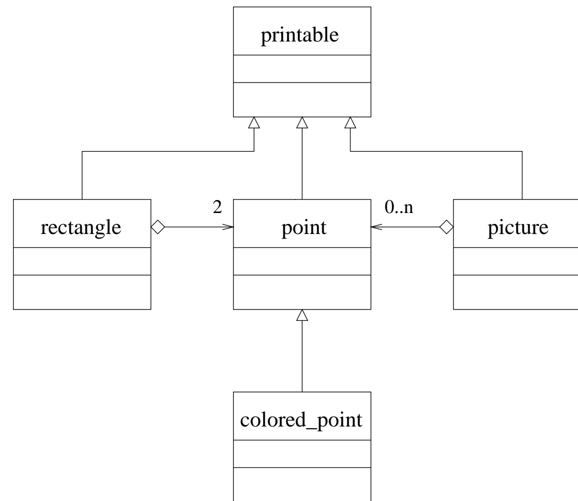


図 15.4: 表示可能オブジェクトを構成するクラスの間の関係

```
                    t#print() ;;
[ ( 1, 1) ( 3, 2) ( 1, 4)]- : unit = ()
```

rectangle クラスは *printable* クラスを継承して to_string メソッドを定義しています。インスタンス変数 llc （同様に urc）は長方形の左下部（同様に右上部）の点の座標を意味しています。

```
# class rectangle (p1,p2) =
    object
      inherit printable ()
      val llc = (p1 : point)
      val urc = (p2 : point)
      method to_string () = "[" ^ llc#to_string() ^ "," ^ urc#to_string() ^ "]"
    end ;;
class rectangle :
  point * point ->
  object
    val llc : point
    val urc : point
    method print : unit -> unit
    method to_string : unit -> string
  end
```

rectangle クラスは抽象クラス *printable* を継承し、print メソッドを引き継いでいます。このクラスは *point* 型のインスタンス変数を二つ（左下部の点と右上部の点）持っています。to_string メソッドは point 型のインスタンス変数 llc と urc に対して to_string メッセージを送っています。

```
# let r = new rectangle (new point (2,3), new point (4,5));;
val r : rectangle = <obj>
# r#print();;
[( 2, 3),( 4, 5)]- : unit = ()
```

クラス、型、オブジェクト

オブジェクトの型はメソッドの型から決定されると以前言いました。たとえば *point* 型は *point* クラスの宣言から推論されますが、実際は以下のような型を省略したものとして定義されます。

```
point =
  < distance : unit -> float; get_x : int; get_y : int;
    moveto : int * int -> unit; rmoveto : int * int -> unit;
    to_string : unit -> string  >
```

これは閉じた型です。つまりすべてのメソッドに関連付けられている型は固定されています。この型に、新しいメソッドや型を追加することはできません。型宣言時に型推論の働きによってクラスに関連付けられている閉じた型が計算されます。

開いた型

オブジェクトにメッセージを送る機能は言語の一部ですから、まだ型が定まっていないオブジェクトに対してもメッセージを送る関数を定義することができます。

```
# let f x = x#get_x ;;
val f : < get_x : 'a; .. > -> 'a = <fun>
```

f の引数に対して推論された型は、x に対してメッセージが送られているためオブジェクト型ですが、このオブジェクト型は開いた型です。関数 f の引数 x は少なくとも get_x というメソッドを持っていなければなりません。このメッセージを送った結果を関数 f の中では使っていないので、結果の型はできるだけ一般的なもの（型変数 *'a*）でなければなりません。このため型推論によって get_x メソッドを持つどんなオブジェクトに対しても関数 f を利用できるようになっています。型 *< get_x : 'a; .. >* の後ろにある連続点「..」は x の型が開いていることを意味しています。

```
# f (new point(2,3)) ;;
- : int = 2
# f (new colored_point(2,3) "emerald") ;;
- : int = 2
# class c () =
    object
      method get_x = "I have a method get_x"
    end ;;
class c : unit -> object method get_x : string end
# f (new c ()) ;;
```

```
- : string = "I have a method get_x"
```

クラスの型を推論すると開いた型を出力することがあります。とりわけクラスのインスタンス生成時の初期値が開いた型になります。次の例では *couple* クラスを定義します。このクラスの初期値 a と b は to_string メソッドを持っています。

```
# class couple (a,b) =
    object
      val p0 = a
      val p1 = b
      method to_string() = p0#to_string() ^ p1#to_string()
      method copy () = new couple (p0,p1)
    end ;;
class couple :
  (< to_string : unit -> string; .. > as 'a) *
  (< to_string : unit -> string; .. > as 'b) ->
  object
    val p0 : 'a
    val p1 : 'b
    method copy : unit -> couple
    method to_string : unit -> string
  end
```

a と b の両方とも型は to_string メソッドを持つ開いた型です。ここで注意が必要なのはこの二つの型は別の型と認識されていることです。これらの型は "*as 'a*" と "*as 'b*" とそれぞれ名付けられています。型変数 *'a* と *'b* は推論された型によって制約されています。

閉じた型から開いた型を生成するためにシャープ記号を使います。

構文 : #オブジェクト型

この型全体でオブジェクト型のすべてのメソッドを持ち、最後に連続点が付いている開いた型を意味しています。

型制約

関数型プログラミングの章(28 ページ)で、ある式が型推論によって生成された型よりも正確な型を持つように制約を与えるにはどうすればいいか学びました。オブジェクト型は、開いた型であれ閉じた型であれこのような制約を与えることができます。すでに定義されているオブジェクト型を、その後別のメソッドに適用するためにあらかじめ開いておきたい場合があるかもしれません。そのような時は開いた型による制約を利用することができます。

構文 : **(***name***:#***type***)**

この構文を使って次のように書くことが出来ます。

```
# let g (x : #point) = x#message;;
val g :
  < distance : unit -> float; get_x : int; get_y : int; message : 'a;
    moveto : int * int -> unit; print : unit -> unit;
    rmoveto : int * int -> unit; to_string : unit -> string; .. > ->
```

```
  'a = <fun>
```

#point という型制約によって x は少なくとも *point* クラスのメソッドをすべて持つことになります。それに加えて「message」メッセージを送っているために引数 x の型には新しいメソッドが追加されています。

オブジェクト指向機能以外の部分と全く同様に Objective Caml ではオブジェクトに対しても推論によって静的な型付けを与えます。この機構が式の型を決定するために充分な情報を得られない時は型変数が割り当てられます。この方式がオブジェクトの型付けにも有効であることを今まで見て来ましたが、それでも時折曖昧な状況が生じます。その場合はプログラマが明確な型情報を与える必要があります。

```
# class a_point p0 =
    object
      val p = p0
      method to_string() = p#to_string()
    end ;;
Characters 6-89:
  ..... a_point p0 =
    object
      val p = p0
      method to_string() = p#to_string()
    end...
Some type variables are unbound in this type:
  class a_point :
    (< to_string : unit -> 'b; .. > as 'a) ->
    object val p : 'a method to_string : unit -> 'b end
The method to_string has type unit -> 'a where 'a is unbound
```

このような曖昧性は引数 p0 が *#point* 型を持つと指定することで解消できます。

```
# class a_point (p0 : #point) =
    object
      val p = p0
      method to_string() = p#to_string()
    end ;;
class a_point :
  (#point as 'a) -> object val p : 'a method to_string : unit -> string end
```

同じ型制約をクラス宣言の中で何度も使いたい時のために、次のような記法を使うことができます。

構文 : **constraint** 型 $_1$ = 型 $_2$

上で挙げた例では引数 p0 が *'a* 型を持つと書くことができます。その場合は型変数 *'a* に型制約を加えます。

```
# class a_point (p0 : 'a) =
    object
      constraint 'a = #point
      val p = p0
      method to_string() = p#to_string()
    end ;;
class a_point :
```

```
  (#point as 'a) -> object val p : 'a method to_string : unit -> string end
```

複数の型制約を一つのクラス宣言に書くことも可能です。

警告 開いた型をメソッドの型に使ってはならない。

開いた型では連続点によって指定された部分に何の制約も受けていない型変数が来る可能性があるため、この強い制約が必要になります。型宣言の時点では自由な型変数を含んではいけないため未束縛の型変数を含むようなメソッドは型推論によってエラーとなります。

```
# class b_point p0 =
    object
      inherit a_point p0
      method get = p
    end ;;
Characters 6-77:
  ..... b_point p0 =
    object
      inherit a_point p0
      method get = p
    end...
Some type variables are unbound in this type:
  class b_point :
    (#point as 'a) ->
    object val p : 'a method get : 'a method to_string : unit -> string end
The method get has type #point where .. is unbound
```

実際に "**constraint** *'a* = *#point*" という型制約によって get の型は *#point* 型という開いた型になります。この型は連続点（..）によって指定された自由な型変数を含んでいるため、このようなメソッドはエラーになります。

継承と変数 self の型

メソッドの型の中の型変数の禁止には例外があります。オブジェクト自身を指す変数 self の型です。二つの点の等価性を調べるメソッドを考えてみましょう。

```
# class point_eq (x,y) =
    object (self : 'a)
      inherit point (x,y)
      method eq (p:'a) = (self#get_x = p#get_x) && (self#get_y = p#get_y)
    end ;;
class point_eq :
  int * int ->
  object ('a)
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method eq : 'a -> bool
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
```

```
    method print : unit -> unit
    method rmoveto : int * int -> unit
    method to_string : unit -> string
  end
```

eq メソッドの型は *'a -> bool* ですが、この型変数はインスタンス生成時の型を意味しています。

point_eq クラスを継承し eq メソッドを再定義することは可能です。再定義されたメソッドの型もやはりインスタンスの型をパラメータとしています。

```
# class colored_point_eq (xc,yc) c =
   object (self : 'a)
     inherit point_eq (xc,yc) as super
     val c = (c:string)
     method get_c = c
     method eq (pc : 'a) =  (self#get_x = pc#get_x) && (self#get_y = pc#get_y)
                      && (self#get_c = pc#get_c)
   end ;;
class colored_point_eq :
  int * int ->
  string ->
  object ('a)
    val c : string
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method eq : 'a -> bool
    method get_c : string
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method print : unit -> unit
    method rmoveto : int * int -> unit
    method to_string : unit -> string
  end
```

colored_point_eq クラスの eq メソッドの型もまた *'a -> bool* ですが、型変数 *'a* は今度は *colored_point_eq* クラスのインスタンスの型を表しています。*colored_point_eq* クラスの eq の定義は親クラスの定義を隠蔽しています。インスタンスの型を含んでいるメソッドはバイナリメソッドと呼ばれています。バイナリメソッドは部分型関係に関する若干の制限があります（470 ページ参照）。

多重継承

多重継承を使うとインスタンス変数やメソッドを複数のクラスから継承できます。もし同じ名前のインスタンス変数やメソッドがあった場合、継承を宣言した順番にしたがって最後の宣言だけが有効となりますが、子クラスでは隠蔽されている親クラスのメソッドに別の名前を割り当てることで、参照することが可能です。しかし、これはインスタンス変数には適用できません。もし継承したクラスによって親クラスのインスタンス変数が隠蔽された場合、その変数を直接アクセスする方法はありません。多くの場合、継

承されたクラスが継承関係を持つことは重要ではありません。多重継承はクラスの再利用性を高めることにポイントがあります。

抽象クラス *geometric_object* を定義してみましょう。このクラスは面積と外周を計算する二つの仮想メソッド compute_area と compute_peri を宣言しています。

```
# class virtual geometric_object () =
   object
     method virtual compute_area : unit → float
     method virtual compute_peri : unit → float
   end;;
```

次に rectangle クラスを次のように再定義します。

```
# class rectangle_2 ((p1,p2) :'a) =
   object
     constraint 'a = point * point
     inherit printable ()
     inherit geometric_object ()
     val llc = p1
     val urc = p2
     method to_string () =
      "["^llc#to_string()^","^urc#to_string()^"]"
     method compute_area() =
      float ( abs(urc#get_x - llc#get_x) * abs(urc#get_y - llc#get_y))
     method compute_peri() =
      float ( (abs(urc#get_x - llc#get_x) + abs(urc#get_y - llc#get_y)) * 2)
   end;;
class rectangle_2 :
  point * point ->
  object
    val llc : point
    val urc : point
    method compute_area : unit -> float
    method compute_peri : unit -> float
    method print : unit -> unit
    method to_string : unit -> string
  end
```

このクラスの実装は図 15.5 の継承グラフにしたがって定義されています。

rectangle クラスのメソッドを二度書きしないですむように図 15.6 にしたがって rectangle クラスを直接継承するようにしましょう。

このような場合は抽象クラス geometric_object の抽象メソッドの定義を与える必要があります。

```
# class rectangle_3 (p2 :'a) =
     object
     constraint 'a = point * point
     inherit rectangle p2
     inherit geometric_object ()
     method compute_area() =
```

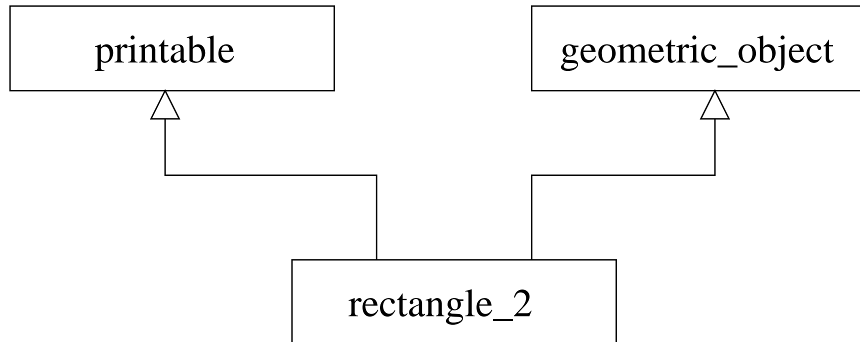


図 15.5: 多重継承

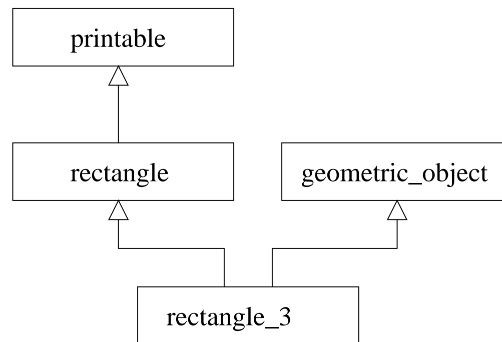


図 15.6: 多重継承（続き）

```
     float ( abs(urc#get_x - llc#get_x) * abs(urc#get_y - llc#get_y))
   method compute_peri() =
     float ( (abs(urc#get_x - llc#get_x) + abs(urc#get_y - llc#get_y)) * 2)
 end;;
```

同じ目的を実現するために printable クラスの階層と geometric_object クラスの階層を別々に定義し、必要になるまで両方のクラスの振る舞いを共有しないでおくやり方もあります。このような考え方に基づいて設計されたクラス階層は図 15.7 のようになります。

printable_rect クラスと geometric_rect クラスが長方形の隅の座標を表すインスタンス変数を持つと定義するとしたら、rectangle_4 は（一つの隅につき二つずつ）四つの点を持つことになります。

```
class rectangle_4 (p1,p2) =
 inherit printable_rect (p1,p2) as super_print
 inherit geometric_rect (p1,p2) as super_geo
end;;
```

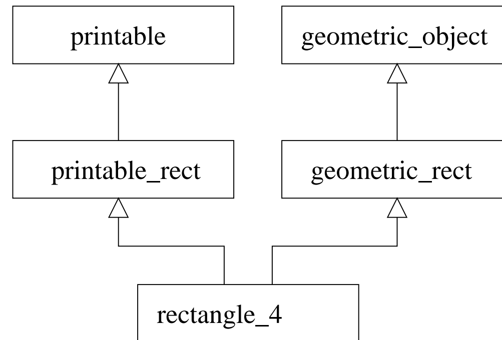


図 15.7: 多重継承（終わり）

`..._rect` という名前の両方のクラスには同じ型のメソッドが存在しますが、この場合、最後に定義されたものだけが直接アクセス可能です。しかし、親クラスに名前（`super_...`）を付けることでそれぞれの親クラスのメソッドを呼び出せるようになります。

多重継承は既存のコードを利用して新しいクラスを作り上げることによってリファクタリングを可能にしますが、コストとして支払わなければならないのは新たに設計されたクラスのサイズです。新しいクラスは、目的とするアプリケーションには不要なインスタンス変数やメソッドを含んでいるためにサイズが必要よりも増大しがちです。そのうえ（最後の例で見たように）情報が複製されている場合はそれらを手動で管理するコストが加わります。最後の `rectangle_4` クラスを使った例では `printable_rect` クラスと `geometric_rect` クラスが座標を表すインスタンス変数を持っていました。もしこれらのクラスのどちらかにその変数を書き換えるメソッド（たとえば一次変換）があれば、その更新をもう一方のクラスに伝搬させる処理が必要になります。このような一貫性保持のための重い処理は、多くの場合、問題をモデル化する手法に根本的な誤りがあることを示唆しています。

型パラメータを持つクラス

型パラメータを持つクラスを使うと Objective Caml のパラメータ型多相性をクラスでも利用できます。Objective Caml の型宣言でのやり方と同じ様にクラス宣言でも型変数をパラメータとして指定できます。この機能はクラスの汎用性とコードの再利用性を高める新しい手段を提供しています。型パラメータを持つクラスは ML 風の型付けと統合されています。

型パラメータを持つクラスの構文はパラメータ化された型の宣言とは若干異なっています。型パラメータを括弧 [] でくくる必要があります。

構文 : **class [** *'a, 'b, ...* **]** クラス名 **= object** ... **end**

ただし型の表示 *('a,'b,...)* クラス名 は今までと変わりません。

仮にペアを実装するクラスを作るとしたら、もっとも単純な方法は次のようになるでしょう。

```
# class pair x0 y0 =
    object
      val x = x0
      val y = y0
      method fst = x
      method snd = y
    end ;;
Characters 6-106:
  ..... pair x0 y0 =
    object
      val x = x0
      val y = y0
      method fst = x
      method snd = y
    end...........
Some type variables are unbound in this type:
  class pair :
    'a ->
    'b -> object val x : 'a val y : 'b method fst : 'a method snd : 'b end
The method fst has type 'a where 'a is unbound
```

しかし、このクラス宣言は以前 *a_point* クラスを定義しようとした（457 ページ）時と同じエラーになります。エラーメッセージは、パラメータ x0 に割り当てられている型変数 *'a* が未束縛（したがって x と fst の型も未束縛）であると述べています。

パラメータ化された型の場合と同様に *pair* クラスに型変数をパラメータとして与える必要があります。そして初期化パラメータ x0 と y0 の型が正しくなるように制約を課する必要があります。

```
# class ['a,'b] pair (x0:'a) (y0:'b) =
    object
      val x = x0
      val y = y0
      method fst = x
      method snd = y
    end ;;
class ['a, 'b] pair :
  'a ->
  'b -> object val x : 'a val y : 'b method fst : 'a method snd : 'b end
```

このようにすれば型推論の結果は型変数 'a と 'b がパラメータとなったクラスの型（インターフェース）となります。

型パラメータを持つクラスのインスタンスが生成されると、型パラメータは生成時のパラメータの型によって具体化されます。

```
# let p = new pair 2 'X';;
val p : (int, char) pair = <obj>
# p#fst;;
- : int = 2
# let q = new pair 3.12 true;;
val q : (float, bool) pair = <obj>
# q#snd;;
- : bool = true
```

注意

クラス宣言では型パラメータを括弧 [] でくくる必要があるが、型としては括弧 () を使って表示される。

型パラメータを持つクラスの継承

型パラメータを持つクラスを継承する時はそのパラメータをはっきりと指定する必要があります。*('a,'b) pair* を継承した *acc_pair* クラスを定義してみましょう。インスタンス変数にアクセスするメソッド get1 と get2 を追加します。

```
# class ['a,'b] acc_pair (x0 : 'a) (y0 : 'b) =
    object
      inherit ['a,'b] pair x0 y0
      method get1 z = if x = z then y else raise Not_found
      method get2 z = if y = z then x else raise Not_found
    end;;
class ['a, 'b] acc_pair :
  'a ->
  'b ->
  object
    val x : 'a
    val y : 'b
    method fst : 'a
    method get1 : 'a -> 'b
    method get2 : 'b -> 'a
    method snd : 'b
  end
# let p = new acc_pair 3 true;;
val p : (int, bool) acc_pair = <obj>
# p#get1 3;;
- : bool = true
```

継承された型パラメータを持つクラスの型パラメータをより正確にすることもできます。点のペアの場合は次のようになります。

```
# class point_pair (p1,p2) =
    object
      inherit [point,point] pair p1 p2
    end;;
class point_pair :
  point * point ->
  object
    val x : point
    val y : point
    method fst : point
    method snd : point
  end
```

point_pair クラスでは型パラメータ *'a* と *'b* が完全に決定されているので、もはや型パラメータは必要ありません。

表示可能なオブジェクト（print メソッドを持つオブジェクト）のペアを作るのに抽象クラス printable （455 ページ）を再利用して、pair クラスから継承した printable_pair クラスを定義します。

```
# class printable_pair x0 y0 =
    object
      inherit [printable, printable] acc_pair x0 y0
      method print () = x#print(); y#print ()
    end;;
```

この実装では printable クラスのペアを持つことができますが、print メソッドを持つ他のクラスに適用できません。

これを解決するため acc_pair の型パラメータを開いた型にすることも考えられます。

```
# class printable_pair (x0 ) (y0 ) =
    object
      inherit [ #printable, #printable ] acc_pair x0 y0
      method print () = x#print(); y#print ()
    end;;
Characters 6-149:
  ..... printable_pair (x0 ) (y0 ) =
    object
      inherit [ #printable, #printable ] acc_pair x0 y0
      method print () = x#print(); y#print ()
    end..
Some type variables are unbound in this type:
  class printable_pair :
    (#printable as 'a) ->
    (#printable as 'b) ->
    object
      val x : 'a
      val y : 'b
      method fst : 'a
      method get1 : 'a -> 'b
      method get2 : 'b -> 'a
      method print : unit -> unit
      method snd : 'b
    end
The method fst has type #printable where .. is unbound
```

しかし、こうすると fst メソッドと snd メソッドが開いた型を持つため失敗します。

そのためクラスの型パラメータを使い、それに開いた型 *#printable* であるという制約を付けることにします。

```
# class ['a,'b] printable_pair (x0 ) (y0 ) =
    object
      constraint 'a = #printable
      constraint 'b = #printable
      inherit ['a,'b] acc_pair x0 y0
      method print () = x#print(); y#print ()
```

```
    end;;
class ['a, 'b] printable_pair :
  'a ->
  'b ->
  object
    constraint 'a = #printable
    constraint 'b = #printable
    val x : 'a
    val y : 'b
    method fst : 'a
    method get1 : 'a -> 'b
    method get2 : 'b -> 'a
    method print : unit -> unit
    method snd : 'b
  end
```

こうすれば点と色付き点の表示可能なペアを作ることができます。

```
# let pp = new printable_pair
             (new point (1,2)) (new colored_point (3,4) "green");;
val pp : (point, colored_point) printable_pair = <obj>
# pp#print();;
( 1, 2)( 3, 4) with color green- : unit = ()
```

型パラメータを持つクラスと型付け

型の観点から見れば、型パラメータを持つクラスとはパラメータ化された型の一種です。そのような型の値は弱い型変数を含むことがあります。

```
# let r = new pair [] [];;
val r : ('_a list, '_b list) pair = <obj>
# r#fst;;
- : '_a list = []
# r#fst = [1;2];;
- : bool = false
# r;;
- : (int list, '_a list) pair = <obj>
```

また、型パラメータを持つクラスは閉じたオブジェクト型と見ることもできます。そのため型パラメータを持つクラスをシャープ記法を使って開いた型として使うことには何の問題もありません。

```
# let compare_nothing ( x : ('a, 'a) #pair) =
    if x#fst = x#fst then x#mess else x#mess2;;
val compare_nothing : < fst : 'a; mess : 'b; mess2 : 'b; snd : 'a; .. > -> 'b =
  <fun>
```

これを利用すると開いた型でもある弱い型変数を含んだ、パラメータ化された型を作ることができます。

```
# let prettytype x ( y : ('a, 'a) #pair) = if x = y#fst then y else y;;
```

```
val prettytype : 'a -> (('a, 'a) #pair as 'b) -> 'b = <fun>
```

もしこの関数に一つだけ引数を適用すると弱い型変数を含む型を持つ関数クロージャに評価されます。*#pair* のような開いた型では連続点（..）によって表される、型が完全に確定していない部分を含んでいます。この観点からは開いた型とは中途半端に確定している型パラメータと認識することもできます。この例のように部分適用の結果、弱められた型は *_#pair* という記法によって表示されます。

```
# let g = prettytype 3;;
val g : ((int, int) _#pair as 'a) -> '_a = <fun>
```

関数 g にペアを適用すると、弱い型が具体化されます。

```
# g (new acc_pair 2 3);;
- : (int, int) acc_pair = <obj>
# g;;
- : (int, int) acc_pair -> (int, int) acc_pair = <fun>
```

その結果、関数 g を普通のペアに対して使えなくなってしまいます。

```
# g (new pair 1 1);;
Characters 4-16:
  g (new pair 1 1);;
     ^^^^^^^^^^^^
This expression has type (int, int) pair = < fst : int; snd : int >
but is here used with type
  (int, int) acc_pair =
    < fst : int; get1 : int -> int; get2 : int -> int; snd : int >
Only the second object type has a method get1
```

最後に、型パラメータを持つクラスの型パラメータも同様に弱い型になるため、同じ結果になります。

```
# let h = prettytype [];;
val h : (('_b list, '_b list) _#pair as 'a) -> '_a = <fun>
# let h2 = h (new pair [] [1;2]);;
val h2 : (int list, int list) pair = <obj>
# h;;
- : (int list, int list) pair -> (int list, int list) pair = <fun>
```

h の引数の型は既に開いた型ではありません。そのため次に示す関数適用は、引数が *pair* 型ではないため型エラーになります。

```
# h (new acc_pair [] [4;5]);;
Characters 4-25:
  h (new acc_pair [] [4;5]);;
     ^^^^^^^^^^^^^^^^^^^^^
This expression has type
  ('a list, int list) acc_pair =
    < fst : 'a list; get1 : 'a list -> int list; get2 : int list -> 'a list;
```

```
      snd : int list >
but is here used with type
  (int list, int list) pair = < fst : int list; snd : int list >
Only the first object type has a method get1
```

注意

型パラメータを持つクラスは、メソッドの型が self 型以外の型変数を持つ時には必ず必要になります。

部分型と包含的多相性

部分型により、ある型のオブジェクトを別のオブジェクトの型であるかのように認識し、使うことができます。*ot2* 型のオブジェクトが *ot1* 型の部分型となるための条件は以下の通りです。

1. *ot1* のすべてのメソッドを含んでいる。
2. *ot1* に含まれている *ot2* のすべてのメソッドは対応する *ot1* のメソッドの部分型になっている。

部分型関係はオブジェクトに対してのみ意味を持ちます。すなわちオブジェクトの型の間でのみ成立します。また部分型関係は常に明示的でなければなりません。オブジェクトの型がある型の部分型であると指定するには以下のような構文を使います。ただしオブジェクトの型はそのスーパータイプの型にしか変換できません。

構文 :

```
(name : sub_type :> super_type)
(name :> super_type)
```

Example

この機能によって colored_point クラスのインスタンスを point クラスのインスタンスとして使うことができます。

```
# let pc = new colored_point (4,5) "white";;
val pc : colored_point = <obj>
# let p1 = (pc : colored_point :> point);;
val p1 : point = <obj>
# let p2 = (pc :> point);;
val p2 : point = <obj>
```

p1 は点であることは分かっていますが、色付き点ではなくなってしまいました。しかし、これは型に関してであり、to_string メッセージは色付き点に関連するメソッドを起動します。

```
# p1#to_string();;
- : string = "( 4, 5) with color white"
```

このようにすれば点と色付き点が混在するリストを作ることができます。

```
# let l = [new point (1,2) ; p1] ;;
val l : point list = [<obj>; <obj>]
# List.iter (fun x → x#print(); print_newline()) l;;
( 1, 2)
( 4, 5) with color white
- : unit = ()
```

もちろんこのようなリストに対して行えるのは point クラスに対して許された操作だけになります。

```
# p1#get_color () ;;
Characters 1-3:
  p1#get_color () ;;
  ^^
This expression has type point
It has no method get_color
```

遅延束縛と部分型を組み合わせることで、**包含的多相性**と呼ばれる新しい形態の多相性を実現できます。これはある特定の型と部分型関係にある限り、複数の型の値を扱う能力を意味しています。静的な型付け情報によって、送られたメッセージが必ず受理されることが保証されていますが、実際にメッセージを受理し、起動されるメソッドは実行時に決定されることになります。

部分型関係は継承ではない

C++、Java、SmallTalk のような主流のオブジェクト指向言語とは違って Objective Caml では部分型と継承は別の概念となっています。この理由は大きく二つあります。

1. クラス c2 が クラス c1 を継承していなかったとしてもクラス c1 の部分型となる可能性があります。実際に colored_point クラスが point クラスと独立に定義されていたとしても point クラスの型を尊重していれば部分型となります。
2. クラス c2 がクラス c1 を継承していたとしてもクラス c1 の部分型とならない可能性があります。この現象は実例を挙げて以下説明をします。この例では、定義中のクラスのメソッドの引数の型が未決定である抽象メソッドを利用します。equal クラスの eq メソッドに注目して下さい。

```
# class virtual equal () =
   object(self:'a)
    method virtual eq : 'a → bool
   end;;
class virtual equal : unit -> object ('a) method virtual eq : 'a -> bool end
# class c1 (x0:int) =
   object(self)
    inherit equal ()
    val x = x0
    method get_x = x
    method eq o = (self#get_x = o#get_x)
   end;;
```

```
class c1 :
  int ->
  object ('a) val x : int method eq : 'a -> bool method get_x : int end
# class c2 (x0:int) (y0:int) =
    object(self)
     inherit equal ()
     inherit c1 x0
     val y = y0
     method get_y = y
     method eq o = (self#get_x = o#get_x) && (self#get_y = o#get_y)
    end;;
class c2 :
  int ->
  int ->
  object ('a)
    val x : int
    val y : int
    method eq : 'a -> bool
    method get_x : int
    method get_y : int
  end
```

クラス c2 のインスタンスの型をクラス c1 のインスタンスの型とみなすことはできません。

```
# let a = ((new c2 0 0) :> c1) ;;
Characters 11-21:
  let a = ((new c2 0 0) :> c1) ;;
            ^^^^^^^^^^
This expression cannot be coerced to type
  c1 = < eq : c1 -> bool; get_x : int >;
it has type c2 = < eq : c2 -> bool; get_x : int; get_y : int >
but is here used with type < eq : c1 -> bool; get_x : int; get_y : int >
Type c2 = < eq : c2 -> bool; get_x : int; get_y : int >
is not compatible with type c1 = < eq : c1 -> bool; get_x : int >
Only the first object type has a method get_y
```

c1 型と *c2* 型は互換性がありません。なぜなら c2 の eq メソッドの型が c1 の eq メソッドの型の部分型ではないからです。なぜこうなるのか見てみましょう。o1 を *c1* のインスタンスとします。仮に *c2* が *c1* の部分型であるとして、そのインスタンスを o21 とします。このとき *c2* の eq メソッドの型は *c1* の eq メソッドの型の部分型になります。さて o21 と o1 は両方とも *c1* 型なので式 o21#eq(o1) は正しく型付けできますが、実行時には o21 は *c2* クラスのインスタンスなので *c2* の eq メソッドが起動されます。しかしこのメソッドは o1 に対して get_y メッセージを送ります。*c1* はこのメソッドを持っていないのでエラーになります。すなわち型システムが破られたことになります。

型システムの役割を議論するためには部分型関係をやや形式的に定義する必要があります。これは次の節で行います。

形式体系

オブジェクトの間の部分型関係　型 t を $< m_1 : \tau_1; \ldots m_n : \tau_n >$、型 t' を $< m_1 : \sigma_1 ; \ldots ; m_n : \sigma_n; m_{n+1} : \sigma_{n+1}; etc \ldots >$ と定義します。型 t' が型 t の部分型であることを $t' \leq t$ と記述することにします。すべての $i \in \{1, \ldots, n\}$ に対して $\sigma_i \leq \tau_i$ が成立する時に $t' \leq t$ と定義します。

関数呼び出し　もし $f : t \to s$ であり $a : t'$ かつ $t' \leq t$ であるならば (fa) は型付け可能であり、その型は s となります。

直観的には型 t を引数に取る関数 f は t の部分型 t' の引数も安全に受け入れられるだろうという事実を意味しています。

関数型の部分型関係　型 $t' \to s'$ が型 $t \to s$ の部分型であることを $t' \to s' \leq t \to s$ と記述し、次の条件が成立することと定義します。

$$s' \leq s \text{ and } t \leq t'$$

$s' \leq s$ という関係を**共変関係**と呼び、$t \leq t'$ という関係を**反変関係**と呼びます。最初は不自然に感じるかもしれませんが、関数型の間の部分型関係は、動的束縛を使ったオブジェクト指向言語のプログラムで簡単に検証することができます。

クラス $c1$ と $c2$ が両方ともメソッド m を持っているとしましょう。$c1$ のメソッド m は $t_1 \to s_1$ という型で、$c2$ のメソッド m は $t_2 \to s_2$ という型であるとします。二つのメソッドを区別するために $c1$ のメソッドを $m_{(1)}$ 、$c2$ のメソッドを $m_{(2)}$ と書くことにします。最後に $c2 \leq c1$ すなわち $t_2 \to s_2 \leq t_1 \to s_1$ と仮定します。それでは共変関係と反変関係の簡単な例を見てみましょう。

$g : s_1 \to \alpha$、$h\ (o : c1)\ (x : t_1)\ =\ g(o\#m(x))$ とおきます。

共変関係： 関数 h の最初の引数は $c1$ 型です。$c2 \leq c1$ なので型 $c2$ のオブジェクトを渡すのは合法です。このとき $o\#m(x)$ によって起動されるメソッドは $m_{(2)}$ になり、s_2 型の値を返します。この値は関数 g に適用され、g の引数の型は s_1 なので $s_2 \leq s_1$ とならなければなりません。

反変関係： 関数 h の二番目の引数の型は t_1 です。上と同じように h の最初の引数に $c2$ 型の値を適用すると $m_{(2)}$ メソッドが起動されます。$m_{(2)}$ は t_2 型の引数でなければならないので $t_1 \leq t_2$ とならなければなりません。

包含的多相性

「多相性」とは、関数の引数にいろいろな「形」（型）の引数を適用できる能力、またはオブジェクトに対して様々な形のメッセージを投げる能力を意味しています。

言語の関数型手続き型カーネルがパラメータ型多相性を持っていることを既に見て来ましたが、それは関数の引数に任意の型の値を適用できる機能でした。関数の多相的引数

は型変数を含んでいます。多相関数は様々な型の引数に対して同じコードを実行します。このため引数の内部構造に影響されずに実行できています。

遅延束縛と結び付いた部分型関係を使うと包含的多相性と呼ばれる、新しい種類の多相性が可能になります。それは同じメッセージを（部分型関係を満たしている）別の型を持つインスタンスに送ることができるようになります。型としては異なる点と色付き点が混在するリストを、色付き点をただの点と見なすことによって点のリストとして作り上げることができます。同じメッセージをリストの要素に送ると、メッセージを受け取るインスタンスの属するクラスによって決まるメソッドが呼び出されることになります。この機構は包含的多相性と呼ばれています。その理由はクラス c に対して送ることができるメッセージの集合はクラス c の部分型であるクラス sc に対して送ることができるメッセージの集合に含まれているからです。パラメータ型多相性とは対照的に実行されるコードはインスタンス毎に別になるかもしれません。

型パラメータを持つクラスを使うと両方の形態の多相性を一緒に利用することができます。

オブジェクトの等価性

445 ページでも触れましたが、オブジェクトの構造的等価性についてのやや違和感を感じるかもしれない性質について説明しましょう。二つのオブジェクトは、それらが物理的に同一である限りにおいて構造的に等価です。

```
# let p1 =  new point (1,2);;
val p1 : point = <obj>
# p1  = new point (1,2);;
- : bool = false
# p1 = p1;;
- : bool = true
```

これは部分型関係に起因しています。クラス *c* の部分型であるクラス *sc* のインスタンス o2 を部分型関係を使ってクラス *c* とみなすことができます。o2 をクラス *c* のインスタンス o1 と比べてみて、もしそれらのインスタンス変数の値がすべて等しかったら o1 と o2 が等価であると考えることもできたでしょう。しかし、これは構造的に等価ではないかもしれません。なぜなら o2 は新しいインスタンス変数が加わっている可能性があるからです。このため Objective Caml では二つのオブジェクトが物理的に違っている時は構造的に違っていると考えます。

```
# let pc1 = new colored_point (1,2) "red";;
val pc1 : colored_point = <obj>
# let q = (pc1 :> point);;
val q : point = <obj>
# p1 = q;;
- : bool = false
```

このように制限された等価性の考え方では二つのオブジェクトが等しいときは正しい情報を与えていますが、二つのオブジェクトが等しいと判定されなかったとしてもそれは実際に違っているかどうか保証していません。

関数型スタイル

オブジェクト指向プログラミングでは通常、手続き型スタイルを採用しています。オブジェクトに送られたメッセージはそのオブジェクトの内部状態（インスタンス変数）を物理的に書き換えることになります。オブジェクト指向プログラミングを関数型スタイルで使うことも可能です。このスタイルではメッセージを送ると、更新された内部状態を持つ新しいオブジェクトを返します。

オブジェクトのコピー

Objective Caml には `self` オブジェクトと全く同一の内容で、いくつかのインスタンス変数の値を変更したコピーを作成するための特別な構文が用意されています。

構文： {< $name_1$=$expr_1$;...; $name_n$=$expr_n$ >}

以下のようにすると、関数型スタイルの点クラスを定義できます。点の座標を相対的に移動した時に、副作用は発生しなくなります。

```
# class f_point p =
   object
    inherit point p
    method f_rmoveto_x (dx) = {< x = x + dx >}
    method f_rmoveto_y (dy) = {< y = y + dy >}
   end ;;
class f_point :
  int * int ->
  object ('a)
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method f_rmoveto_x : int -> 'a
    method f_rmoveto_y : int -> 'a
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method print : unit -> unit
    method rmoveto : int * int -> unit
    method to_string : unit -> string
  end
```

この新しい定義では、点の座標を移動した時に、元のオブジェクトの座標を変更する代わりに変更先の座標を持った新しいオブジェクトを返すようになります。

```
# let p = new f_point (1,1) ;;
val p : f_point = <obj>
# print_string (p#to_string()) ;;
( 1, 1)- : unit = ()
# let q = p#f_rmoveto_x 2 ;;
val q : f_point = <obj>
# print_string (p#to_string()) ;;
( 1, 1)- : unit = ()
```

```
# print_string (q#to_string()) ;;
( 3, 1)- : unit = ()
```

これらのメソッドは新しいオブジェクトを生成するので、f_rmoveto_x メソッドの結果に直接メッセージを送ることが可能です。

```
# print_string ((p#f_rmoveto_x 3)#to_string()) ;;
( 4, 1)- : unit = ()
```

f_rmoveto_x メソッドと f_rmoveto_y メソッドの返り値の型は f_rmoveto_x の型の中の型変数 *'a* が示しているように定義されたクラスのインスタンスの型と同じです。

```
# class f_colored_point (xc, yc) (c:string) =
    object
      inherit f_point(xc, yc)
      val color = c
      method get_c = color
    end ;;
class f_colored_point :
  int * int ->
  string ->
  object ('a)
    val color : string
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method f_rmoveto_x : int -> 'a
    method f_rmoveto_y : int -> 'a
    method get_c : string
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method print : unit -> unit
    method rmoveto : int * int -> unit
    method to_string : unit -> string
  end
```

f_colored_point クラスのインスタンスに対して f_rmoveto_x メッセージを送ると、返り値は f_colored_point クラスの新しいインスタンスになります。

```
# let fpc = new f_colored_point (2,3) "blue" ;;
val fpc : f_colored_point = <obj>
# let fpc2 = fpc#f_rmoveto_x 4 ;;
val fpc2 : f_colored_point = <obj>
# fpc2#get_c;;
- : string = "blue"
```

Oo モジュールで定義されている copy プリミティブを使っても任意のオブジェクトのコピーを行うことができます。

```
# Oo.copy ;;
- : (< .. > as 'a) -> 'a = <fun>
```

```
# let q = Oo.copy p ;;
val q :
  < distance : unit -> float; f_rmoveto_x : int -> f_point;
    f_rmoveto_y : int -> f_point; get_x : int; get_y : int;
    moveto : int * int -> unit; print : unit -> unit;
    rmoveto : int * int -> unit; to_string : unit -> string > =
  <obj>
# print_string (p#to_string()) ;;
( 1, 1)- : unit = ()
# print_string (q#to_string()) ;;
( 1, 1)- : unit = ()
# p#moveto(4,5) ;;
- : unit = ()
# print_string (p#to_string()) ;;
( 4, 5)- : unit = ()
# print_string (q#to_string()) ;;
( 1, 1)- : unit = ()
```

例：リストのためのクラス

関数型スタイルでは返り値を計算するためにオブジェクト自身の値 `self` を使うことがあります。この点について整数のリストを表すクラス階層を定義して具体的に説明しましょう。

最初に、リストの要素の型をパラメータとする抽象クラスを定義します。

```
# class virtual ['a] o_list () =
   object
     method virtual empty : unit → bool
     method virtual cons  : 'a → 'a o_list
     method virtual head :  'a
     method virtual tail : 'a o_list
   end;;
```

次に空でないリストのクラスを定義します。

```
# class  ['a] o_cons (n ,l)  =
   object (self)
     inherit ['a] o_list ()
     val car = n
     val cdr = l
     method empty () = false
     method cons x = new o_cons (x, (self : 'a #o_list :> 'a o_list))
     method head = car
     method tail = cdr
   end;;
class ['a] o_cons :
  'a * 'a o_list ->
  object
    val car : 'a
    val cdr : 'a o_list
```

```
    method cons : 'a -> 'a o_list
    method empty : unit -> bool
    method head : 'a
    method tail : 'a o_list
  end
```

cons メソッドは *'a o_cons* 型の新しいインスタンスを返していることに注意してください。そのために self の型は一度 *'a #o_list* 型に指定し、それから *'a o_list* 型の部分型であるという制約を加えています。もし部分型を使わなければ *'a #o_list* という開いた型がメソッドの型に含まれることになりますが、これは厳格に禁止されています（460 ページ参照）。この付加された制約がなければ self の型が *'a o_list* 型の部分型とはなりません。

このようにすれば cons メソッドの型を意図した通りに定義できます。同じトリックを使って、空リストのクラスを定義します。

```
# exception EmptyList ;;
# class ['a] o_nil () =
    object(self)
     inherit ['a] o_list ()
     method empty () = true
     method cons x = new o_cons (x, (self : 'a #o_list :> 'a o_list))
     method head = raise EmptyList
     method tail = raise EmptyList
    end ;;
```

まず最初に空リストのインスタンスを生成し、それから整数のリストを構築します。

```
# let i = new o_nil ();;
val i : '_a o_nil = <obj>
# let l = new o_cons (3,i);;
val l : int o_list = <obj>
# l#head;;
- : int = 3
# l#tail#empty();;
- : bool = true
```

最後の式では整数の 3 を含むリストに対して tail メッセージを送っています。これは *'a o_cons* クラスの tail メソッドを起動します。その結果に対して empty() メッセージが送られ、最終的に true が返されています。最後に起動されたメソッドが *'a o_nil* クラスの empty メソッドであるのは簡単に分かるでしょう。

オブジェクト拡張機能の他の部分について

この節では「オブジェクト」型とクラス内部での局所宣言について解説します。局所宣言を使って、構築子が局所環境への参照を持つようにしてクラス変数を実現することができます。

インターフェース

クラスのインターフェースは通常は型システムによって推論されますが、型宣言によって定義することも可能です。この型には公開されたメソッドのみ出現します。

構文：

```
class type インターフェース名 =
  object
    ⋮
    val インスタンス変数名_i : 型_i
    ⋮
    method メソッド名_j : 型_j
    ⋮
  end
```

point クラスのインターフェースは次のようになります。

```
# class type interf_point =
  object
    method get_x : int
    method get_y : int
    method moveto : (int * int ) → unit
    method rmoveto : (int * int )  → unit
    method to_string : unit → string
    method distance : unit → float
  end ;;
```

この宣言によって定義された型は型制約として使うことができます。

```
# let seg_length (p1:interf_point) (p2:interf_point) =
   let x = float_of_int (p2#get_x - p1#get_x)
   and y = float_of_int (p2#get_y - p1#get_y) in
    sqrt ((x*.x) +. (y*.y)) ;;
val seg_length : interf_point -> interf_point -> float = <fun>
```

インターフェースは、インスタンス変数とプライベートメソッドのみを隠蔽することができます。抽象メソッドや公開メソッドは隠蔽できません。

インターフェースの用法についてのこの制限を次の例で示します。

```
# let p = ( new point_m1 (2,3) : interf_point);;
Characters 11-29:
  let p = ( new point_m1 (2,3) : interf_point);;
            ^^^^^^^^^^^^^^^^^^
This expression has type
  point_m1 =
    < distance : unit -> float; get_x : int; get_y : int;
      moveto : int * int -> unit; rmoveto : int * int -> unit;
      to_string : unit -> string; undo : unit -> unit >
but is here used with type
```

```
  interf_point =
    < distance : unit -> float; get_x : int; get_y : int;
      moveto : int * int -> unit; rmoveto : int * int -> unit;
      to_string : unit -> string >
Only the first object type has a method undo
```

しかし、インターフェースを継承と同時に使っても構いません。インターフェースはモジュールと組み合わせるととりわけ役に立ちます。オブジェクト型を使って、クラスインターフェースだけが公開されているモジュールの署名を作ることもできます。

クラス内での局所宣言

クラス宣言は新しい型と構築子を定義します。いままでは説明の都合で構築子を環境を持たない関数として扱って来ましたが、実際はインスタンスを生成するために初期値を必要としない構築子を定義可能です。つまり構築子は必ずしも関数的ではありません。それどころかクラス内で局所宣言を行うこともできます。構築子から参照できる局所変数はインスタンスの間で共有され、クラス変数として扱うこともできます。

定数構築子

クラス宣言では構築子へと渡される初期値を使う必要はありません。例えば次のようなクラスは合法です。

```
# class example1 =
    object
      method print () = ()
    end ;;
class example1 : object method print : unit -> unit end
# let p = new example1 ;;
val p : example1 = <obj>
```

このクラスのインスタンス構築子は定数です。インスタンス変数のための初期値を必要としていません。しかし余計な誤解を避けるため () のような何らかの初期値を利用し、構築子を関数であるかのように統一して扱った方がいいでしょう。

構築子のための局所宣言

局所宣言は関数抽象を使って直接書くことができます。

```
# class example2 =
    fun a →
      object
        val mutable r = a
        method get_r = r
        method plus x = r <- r + x
     end;;
class example2 :
  int ->
  object val mutable r : int method get_r : int method plus : int -> unit end
```

これを見ると構築子が持つ関数型の性質が簡単に理解できるでしょう。構築子とはクラス宣言時の環境への参照を持つ関数クロージャです。クラス宣言の構文では局所宣言をこのような関数の中に記述することが許されています。

クラス変数

クラス変数はクラスに対して定義され、そのクラスのすべてのインスタンスで共有されます。通常は、このクラス変数はインスタンス生成時以外で使われます。Objective Camlでは、空でない環境を持つ構築子を関数的に使うことですべてのインスタンス変数で共有されるクラス変数（とりわけ書き換え可能なもの）を実現できます。

以下の例でこの機能を詳しく見て行きましょう。ここではクラスのインスタンスの数を記憶するクラス変数を定義します。このためにまず型パラメータを持つ抽象クラス'a om を定義します。

```
# class virtual  ['a] om =
    object
      method  finalize ()  = ()
      method virtual destroy : unit → unit
      method virtual to_string : unit → string
      method virtual all : 'a list
    end;;
```

次にクラス 'a lo を宣言します。このクラスの構築子は変数 n と l のための局所宣言を含んでいます。変数 n はインスタンス毎に固有の番号を割り当てるために使われます。変数 l は生存しているインスタンスを保持しておくために使われます。

```
# class ['a] lo =
    let l = ref []
    and n = ref 0 in
      fun s →
        object(self:'b )
          inherit ['a] om
          val mutable num = 0
          val name = s
          method to_string () = s
          method print () = print_string s
          method print_all () =
            List.iter (function (a,b) →
                          Printf.printf "(%d,%s) " a (b#to_string())) !l
          method destroy () = self#finalize();
                              l:= List.filter (function (a,b) → a <> num) !l; ()
          method all = List.map snd !l
          initializer incr n; num <- !n;  l:= (num, (self :> 'a  om) ) :: !l ; ()
        end;;
class ['a] lo :
  string ->
  object
    constraint 'a = 'a om
    val name : string
    val mutable num : int
    method all : 'a list
```

```
    method destroy : unit -> unit
    method finalize : unit -> unit
    method print : unit -> unit
    method print_all : unit -> unit
    method to_string : unit -> string
  end
```

lo クラスのインスタンスを生成する毎にイニシャライザは参照 `n` を一つ増やし、固有番号と生成されたインスタンスのペアをリスト `l` に加えます。`print` メソッドはインスタンスの名前を表示し、`print_all` メソッドは変数 `l` に含まれているすべてのインスタンスを表示します。

```
# let m1 = new lo "start";;
val m1 : ('a om as 'a) lo = <obj>
# let m2 = new lo "between";;
val m2 : ('a om as 'a) lo = <obj>
# let m3 = new lo "end";;
val m3 : ('a om as 'a) lo = <obj>
# m2#print_all();;
(3,end) (2,between) (1,start) - : unit = ()
# m2#all;;
- : ('a om as 'a) list = [<obj>; <obj>; <obj>]
```

`destroy` メソッドは、指定されたインスタンスの後処理を行わせるために `finalize` メソッドを呼び出し、すべてのインスタンスのリストから、このインスタンスを取り除きます。`all` メソッドは **new** によって生成されたすべてのインスタンスを返します。

```
# m2#destroy();;
- : unit = ()
# m1#print_all();;
(3,end) (1,start) - : unit = ()
# m3#all;;
- : ('a om as 'a) list = [<obj>; <obj>]
```

このクラスのサブクラスのインスタンスもこのリストに保持されることに注意して下さい。それらのサブクラスを特化しても、このテクニックの妨げにはなりません。しかし、コピーによって（`Oo.copy` または {**< >**}）生成されたインスタンスはこのリストに登録されません。

練習問題

オブジェクトによるスタック

スタックの例をまた考えてみましょう。今回はオブジェクト指向スタイルで作ってみましょう。

1. Objective Caml のリストを使って intstack クラスを定義してください。このクラスは push、pop、 top、size メソッドを持っています。
2. スタックの要素として 3 と 4 を含んでいるスタックのインスタンスを生成してください。
3. スタックの要素がメソッド print : unit -> unit を持つように新しいクラス stack を定義してください。
4. 同じメソッドを持つ、型パラメータを持つクラス ['a] stack を定義してください。
5. それぞれのスタックの実装を比較してください。

遅延束縛

この演習ではサブタイピング以外の目的での遅延束縛の使い方について勉強します。

以下のプログラムについて

1. クラスの関係を記述してください。
2. それぞれのクラスでメッセージの処理が変わる部分を書いてください。
3. エコー無しのキャラクタ端末を使っていたとしたら、このプログラムはどんな結果を表示しますか？

```
exception CrLf;;
class chain_read (m) =
  object (self)
  val msg = m
  val mutable res = ""

  method char_read =
    let c = input_char stdin in
      if (c != '\n') then begin
        output_char stdout c; flush stdout
      end;
      String.make 1 c

  method  private chain_read_aux =
      while true do
        let s = self#char_read in
          if s = "\n" then raise CrLf
          else res <- res ^ s;
      done

  method private chain_read_aux2 =
    let s = self#char_read in
          if s = "\n" then raise CrLf
          else begin res <- res ^ s; self#chain_read_aux2 end

  method chain_read =
    try
```

```
      self#chain_read_aux
    with End_of_file  → ()
      | CrLf → ()

  method input = res <- ""; print_string msg; flush stdout;
                 self#chain_read

  method get = res
end;;

class mdp_read (m) =
  object (self)
  inherit chain_read m
  method char_read = let c  = input_char stdin in
                     if (c != '\n') then begin
                       output_char stdout '*'; flush stdout
                     end;

                     let s = " " in s.[0] <- c; s
end;;

let login = new chain_read("Login : ");;
let passwd = new mdp_read("Passwd : ");;
login#input;;
passwd#input;;
print_string (login#get);;print_newline();;
print_string (passwd#get);;print_newline();;
```

抽象クラスと式の評価器

この演習では抽象クラスを使ったプログラムの機能分解の仕方について勉強します。

この演習で定義される、算術式のためのクラスは抽象クラス *expr_ar* のサブクラスです。

1. 算術式のための抽象クラス *expr_ar* を定義してください。このクラスは *float* 型の eval メソッドと *unit* 型の print メソッドを持っています。両者とも抽象メソッドであり、それぞれ式の評価を行い、算術式を表示します。
2. *expr_ar* クラスの具体サブクラス *constant* を定義してください。
3. *expr_ar* クラスの抽象サブクラス *bin_op* を定義してください。このクラスは eval メソッドを *(float * float) -> float* 型を持つ新しい抽象メソッド oper によって定義しています。同様に、print メソッドを *string* 型を持つ新しい抽象メソッド symbol によって定義しています。
4. *bin_op* クラスの具体サブクラスで、メソッド oper とメソッド symbol を実装した新しいクラス *add* と *mul* を定義してください。
5. 継承木を書いてください。
6. Genlex.token の列を受け取り、*expr_ar* 型のオブジェクトを構築する関数を書いてください。

7. 字句解析器 Genlex を使って標準入力から数式を入力してこのプログラムの動作テストを行ってください。数式の表現として後置記法を使っても構いません。

ライフゲームとオブジェクト

以下の二つのクラスを定義してください。

- セルを表現する cell クラス。メソッド isAlive : unit -> bool を持つ。
- セルの配列を持つ world クラス。以下のメソッドを持っている。

```
display : unit -> unit
nextGen : unit -> unit
setCell : int * int -> cell -> unit
getCell : int * int -> cell
```

1. cell クラスを書いてください。
2. display メソッド、getCell メソッド、setCell メソッドが定義された抽象クラス absWorld を書いてください。nextGen メソッドは抽象メソッドとしてください。
3. absWorld クラスのサブクラス world を定義してください。nextGen メソッドに生存規則を実装します。
4. メインプログラム を書いてください。このプログラムは、世界を生成し、生存しているセルを加えて、世界を表示し、ユーザからの入力を待って、次の状態を計算します（その後表示以降を繰り返す)。

まとめ

この章では Objective Caml 言語のオブジェクト指向拡張部分について解説を行いました。クラスはモジュールの代替物でもあり、継承と遅延束縛の機能によってプログラムをオブジェクトによってモデル化することができます。同時にプログラムの再利用性と適用可能性を高めることもできます。この拡張部分は Objective Caml の型システムと統合されており、またサブタイプの概念を付け加えています。サブタイプを利用するとある型の値を想定している文脈に対してそのサブタイプのインスタンスを適用することが可能になります。サブタイピングと遅延束縛を組み合わせることで、包含的多相性を実現しています。それを使うと例えば、型としては均一であっても振る舞いとしては同一でないリストを構築することができます。

もっと知りたい人へ

オブジェクト指向に関しては膨大な数の文献が存在しています。また個々のオブジェクト指向言語はそれぞれ異なったモデルに基づいています。

一般的な入門書としては、既に第一部で紹介されていますが、オブジェクト指向のアプローチを説明している “Langages à Objets ” [MNC+91] があります。より専門的な本としては “Langages et modèles à objets” [DEMN98] が多くの例を含んでいます。

モデリングの手法を学ぶには “Design patterns” [GHJV95] がデザインパターンの再利用性についての具体的に詳説しています。

UML 記法については Rational のサイトを参照してください。

リンク: http://www.rational.com/uml/resources

オブジェクト指向の機能を持った関数型言語について学ぶには SMALLTALK に由来する “Lisp” objects と CLOS (*Common Lisp Object System*) を挙げておきます。他にも CLOS と同様の機能を実現した Scheme の実装が数多くあります。

他にも静的に型付けされた関数型言語のためのオブジェクト指向言語が提案されています。例えば多重定義のための、パラメータ化されたアドホックな多相性を持つ Haskell があります。

Objective Caml のオブジェクト指向拡張についての理論的側面について文献 [RV98] が解説しています。

インターネット上の文書からも Objective Caml のオブジェクトの静的な型付けについてもっと知ることができます。

María-Virginia Aponte による解説があります。

リンク: http://tulipe.cnam.fr/personne/aponte/ocaml.html

Didier Rémy によるオブジェクトの短い説明があります。

リンク: http://pauillac.inria.fr/˜remy/objectdemo.html

Didier Rémy による解説があります。

リンク: http://pauillac.inria.fr/˜remy/classes/magistere/

Roberto Di Cosmo による講義があります。

リンク: http://www.dmi.ens.fr/users/dicosmo/CourseNotes/OO/

16

アプリケーションの構成モデルの比較

第 14 章と第 15 章では、それぞれ、アプリケーション構成方法に関する二つのモデル、つまり、関数型/モジュール型モデルと、オブジェクトモデルについて見てきました。この二つのモデルは、それぞれの方法で、アプリケーション開発に必要となる以下の点に対処しています。

- プログラムの論理構成: モジュール、もしくは、クラス;
- 分割コンパイル: 単純モジュール;
- 抽象データ型: モジュール（抽象型）、もしくは、オブジェクト;
- コンポーネントの再利用: ファンクター/パラメタ多相による型の共有、もしくは、継承/パラメタ付クラスによるサブタイピング;
- コンポーネントの変更可能性: 遅延束縛（オブジェクト）.

モジュラーなアプリケーション開発は、アプリケーションを論理単位、つまり、モジュールに分割することから始まります。そして、シグネチャの記述によってその仕様を実現し、最後に実装を行います。モジュールを実装する間に、そのモジュールのシグネチャやそのパラメタのシグネチャをを変更する必要があるかもしれません。そして、そのソースを変更することが必要になります。このモジュールが他のアプリケーションにすでに使われている場合、あまり喜ばしいことではありません。それにもかかわらず、このプロセスは、プログラマにとって厳密で安心できるフレームワークを提供します。

オブジェクトモデルでは、問題の解析の結果がクラス間の関係として記述されます。もし、後でクラスが要求される機能を提供していないことがわかった場合には、いつでもサブクラス化によって拡張することができます。このプロセスによって、ソースコードとそのクラスを使用するアプリケーションの動作を変更することなく、クラス階層の大部分の再利用が可能になります。残念ながら、この手法はコード量の爆発につながり、多重継承を用いたコード複製を難しくなります。

多くの問題では、再帰なデータ型とその型のデータへの操作が必要になりますが、実装の過程で、あるいは保守の間にも、型と操作の拡張が必要となるような問題へと進化していきます。二つのモデルは、いずれも、もう一方のやり方で拡張を行うことができま

せん。関数型/モジュール型モデルでは型は拡張できませんが、型への新しい関数（操作）を定義することができます。オブジェクトモデルでは、オブジェクトは拡張できますが、（抽象クラスに対してあらたにメソッドを定義する新しいサブクラスをつくることによって）メソッドは拡張できません。この点で、二つのモデルは双対なのです。

この二つのモデルを一つの言語で統一することの利点は、問題解決に最も適したモデルを選択することができるようになること、そして、それぞれのモデルの限界を克服するために二つのモデルを同時に使用することでできることなのです。

この章の構成

最初の節は、関数型/モジュラーモデルとオブジェクトモデルを比較します。

この比較では、それぞれのモデルの特定の機能を取り上げ、一方のモデルから他方のモデルへ手作業で変換することで、一方のモデルのうちのどれだけが対応するもう一方の機能でシミュレートできるかを示します。

例えば、モジュールで継承をシミュレートすることができますし、単純モジュールを実装するのにクラスを使うこともできます。

その上で、それぞれのモデルの限界を再検討します。第二節では、データ構造とメソッドのための拡張性の問題をとりあげ、二つのモデルを混在使用する方法を提案します。第二節では、オブジェクトに抽象モジュールを使用するという、別の混在使用の方法について述べることにします。

モジュールとオブジェクトの比較

Objective Caml におけるモジュールプログラミングとオブジェクト（指向）プログラミングの違いは、主にその型システムの違いにあります。基本的には、オブジェクトを用いるプログラミングが *ad hoc* 多相（メッセージ送信によって、全くことなるプログラムコードが適用されること）を必要とするのに対して、モジュールによるプログラミングは ML の型システムの範囲内（例えば、パラメタ多相のコードは、異なる型のパラメタに対して実行されうる）にあります。これは、サブタイプの場合に明らかになります。ML の型システムのサブタイプ拡張は、純粋 ML の枠組みの中でシミュレートすることは出来ません。型システムを破壊することなく、異なる型の値を持つリストをつくることはできないでしょう。

モジュールプログラミングとオブジェクト（指向）プログラミングは、プログラムの論理構成に対する、ソフトウエア部品の再利用性と変更可能性を許す、二つの（型付けのおかげで）安全なアプローチです。Objective Caml におけるオブジェクトを用いたプログラミングでは、遅延束縛とサブタイプのおかげで、パラメタ多相（パラメタ付きクラス）と**包含/サブタイプ多相**（メッセージ送信）を利用できます。モジュールプログラミングは、パラメタ多相の使用を制限し、直接束縛を使用することができるので、効率的な実行には便利です。

モジュールプログラミングモデルでは、拡張可能でない再帰的なデータ構造に対して、容易に関数の拡張を行うことができますが、もし、あるバリアント型にケースを一つ追

加したい場合には、ソースコードの大部分を修正しなければいけなくなるだしょう。オブジェクト（指向）プログラミングモデルでは、クラスを用いて再帰的なデータ構造の集まりを定義でき、一つのクラスをデータ型の一つのケースと見なすことができます。

実行効率

遅延束縛は、メソッドテーブルによる間接参照に対応します（451 ページ参照）。クラス外部からのメッセージ送信によるインスタンス変数へのアクセスと同様に、この間接参照も積み重なるとそのコストは高くつきます。

実行効率の低下を示すために、以下のようなクラス階層で見てみましょう。

```
# class virtual test () =
    object
     method virtual sum : unit -> int
     method virtual sum2 : unit -> int
    end;;
# class a x  =
    object(self)
     inherit test ()
     val a = x
     method a = a
     method sum () = a
     method sum2 () = self#a
    end;;
# class b x y  =
    object(self)
    inherit a x as super
     val b = y
     method b = b
     method  sum () = b + a
     method  sum2 () = self#b + super#sum2()
    end;;
```

さて、クラス b のインスタンスへのメッセージ sum と sum2 の送信と、以下の関数 f の呼び出しの実行時間を比べてみましょう。

```
# let f a b = a + b ;;
# let iter g a n = for i = 1 to n do ignore(g a) done ; g a ;;
# let go i j = match i with
                 1 -> iter (fun x -> x#sum()) (new b 1 2) j
               | 2 -> iter (fun x -> x#sum2()) (new b 1 2) j
               | 3 -> iter (fun x -> f 1 x) 2 j ;;

# go (int_of_string (Sys.argv.(1))) (int_of_string (Sys.argv.(2))) ;;
```

1 千万回繰り返した場合の結果は以下の通りです。

	bytecode	native
case 1	07,5 s	0,6 s
case 2	15,0 s	2,3 s
case 3	06,0 s	0,3 s

この例は、標準的な静的束縛に対して遅延束縛のほうがコストがかかることを示すために作られたものです。このコストは、関数の中でのメッセージ送信の回数に対する計算の質に依存します。ネイティブコンパイラを使用することで、このテストの間接参照の部分を変えずに、計算の部分のコストを減らすことができます。ケース 2 の場合には、メッセージ sum2 の送信の際の多重間接参照のコストが「圧縮できない」ことがわかります。

例: グラフィカル・インターフェース

AWI グラフィカルライブラリ（380 参照）は、Objective Caml の関数型/逐次型の言語核を用いて設計されています。モジュールの形にするのも容易です。コンポーネントはそれぞれ独立したモジュールになるため、関数名を一致させることができます。コンポーネントを追加するには、追加するコンポーネントの具体型を知る必要があります。外見と振る舞いを記述するためのフィールドの変更は、新しいモジュールが行われなければなりません。

このライブラリは、オブジェクトを用いて書き直すこともできます。クラス図は、図 16.1 のようになります。

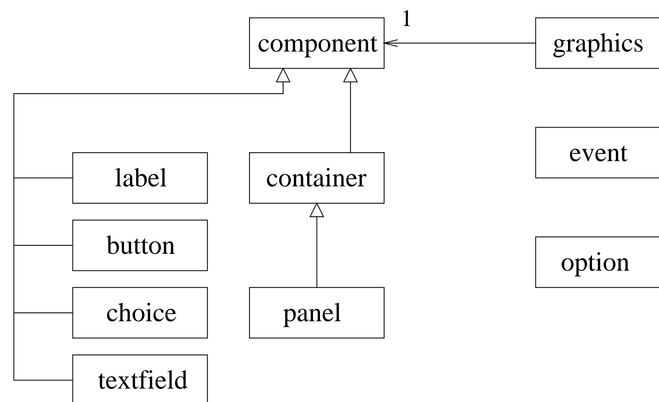


図 16.1: Class hierarchy for AWI.

継承があるので、モジュールを使う場合と比べて、新しい部品を追加するのは簡単です。しかし、オーバーローディングがないため、オプションはメソッドのパラメタに埋め込まれ必要があります。サブタイプ関係を使えば、コンテナに含まれる構成物のリストをつくるのも容易です。遅延束縛により、部品のメソッドが適切に選択されます。オブジェクトモデルの面白い点は、グラフィックコンテキストを拡張したり変更したりできる可能性にあります。おもなグラフィックライブラリがオブジェクトモデルに従って構成されるのはこのためです。

モジュールをクラスに変換する

型をただ一つだけ宣言していて、型独立な多相関数を持たない、単純あモジュールはクラスに変換することができます。ただ一つの型がどのように使われるか（レコード型かバリアント型）によって、モジュールをクラスに変換する方法は異なります。

型宣言

レコード型。 レコード型の場合は、レコード型のフィールドが、インスタンス変数になるようなクラスで記述することができます。

バリアント型。 バリアント型の場合は、「構成要素」の概念モデルを使って、複数のクラスに変換します。抽象クラスがこの型への操作（関数）を記述します。バリアント型における場合別けの各ケースは、抽象クラスの子クラスとなり各ケースのために抽象メソッドを実装します。パターンマッチを行う必要はありませんが、各ケースに応じて適切なメソッドを選択する必要があります。

パラメタ付型 パラメタ付型は、パラメタ付クラスとして実装されます。

抽象型 クラスは抽象型と見なすことができます。クラスの階層の外では、そのクラスの内部状態を消して見ることはできません。それにも関わらず、クラスのインスタンス変数にアクセスするために子クラスを定義することを禁止することは出来ません。

相互再帰型 相互再帰な型の宣言は、相互再帰なクラスの宣言に変換されます。

関数宣言

モジュール型 *t* に依存するパラメタを持つ関数は、メソッドに変換することができます。*t* が現れない関数は、**`private`** として定義できます。残りは、あるパラメタが型 *t* で別のパラメタが *'a* の場合です。このような関数は、標準ライブラリのモジュールでは非常に稀です。非標準的な型付けを行う `Marshal` や `Printf` といった「珍しい」モジュールか、`List` のような線形構造の (…を操作する) モジュールがこの例になっています。最後に、*('a -> 'b -> 'a) -> 'a -> 'b list -> 'a* を持つ型関数 `fold_left` を変換するのは、特にクラス *['b] list* のメソッドの場合には、難しくなります。これは、型変数 *'a* が自由で、メソッドの型に現れないからです。クラス `list` に型パラメタをつけるよりはむしろ、これらの関数を二つの型パラメタとリストのフィールドを持つ新しいクラスのメソッドとして定義した方がよいでしょう。

バイナリーメソッド サブタイプ以外では、バイナリーメソッドは何の問題にもなりません。

その他の宣言　非関数値の宣言は、クラスの外でも受け付けられます。例外についても同様です。

例: イテレータ付きのリスト　以下のシグネチャLIST を持つモジュールをオブジェクトに変換してみましょう。

```
# module type LIST = sig
    type 'a list = C0 | C1 of 'a * 'a list
    val add : 'a list → 'a → 'a list
    val length : 'a list → int
    val hd : 'a list → 'a
    val tl : 'a list → 'a list
    val append : 'a list → 'a list → 'a list
    val fold_left : ('a → 'b → 'a) → 'a → 'b list → 'a
  end ;;
```

まず最初に、この型に対応する抽象クラス`'a list`を定義します。

```
# class virtual ['a] list () =
    object (self : 'b)
      method virtual add : 'a → 'a list
      method virtual empty : unit → bool
      method virtual hd : 'a
      method virtual tl : 'a list
      method virtual length : unit → int
      method virtual append : 'a list  → 'a list
    end ;;
```

次に、c1_list と c0_list という二つのサブクラスを定義します。この二つのクラスがバリアント型に対応します。それぞれのクラスでは、親の抽象クラスのメソッドが定義されます。

```
# class ['a] c1_list (t, q)  =
    object (self )
      inherit ['a] list () as super
      val t = t
      val q = q
      method add x = new c1_list (x, (self : 'a #list :> 'a list))
      method empty () = false
      method length () = 1 + q#length()
      method hd = t
      method tl = q
      method append l  = new c1_list (t,q#append l)
    end ;;
# class ['a] c0_list () =
    object (self)
      inherit ['a] list () as super
      method add x = new c1_list (x, (self : 'a #list :> 'a list))
      method empty () = true
      method length () = 0
```

```
      method hd = failwith "c0_list : hd"
      method tl = failwith "c0_list : tl"
      method append l  = l
    end ;;
# let l = new c1_list (4, new c1_list (7, new c0_list ())) ;;
val l : int list = <obj>
```

関数 LIST.fold_left は、新しい型パラメタを導入する必要があるので list に取り入れることはできません。そこで、このメソッドを実装するためにクラス fold_left を定義します。このために、関数を値とするインスタンス変数（f）を用います。

```
# class virtual ['a,'b] fold_left () =
    object(self)
      method virtual f : 'a -> 'b -> 'a
      method iter r (l : 'b list) =
        if l#empty() then r else self#iter (self#f r (l#hd)) (l#tl)
    end ;;
# class ['a,'b] gen_fl f =
    object
      inherit ['a,'b] fold_left ()
      method f = f
    end ;;
```

Thus we construct an instance of the class gen_fl for addition:

```
# let afl = new gen_fl (+) ;;
val afl : (int, int) gen_fl = <obj>
# afl#iter 0 l ;;
- : int = 11
```

モジュールにおける継承の実現

クラス間の継承関係により、サブクラスにおいて上位クラスの変数宣言とメソッドの集まりを取り出すことができます。モジュールを用いてこれをシミュレートすることができます。継承をしているサブクラスはパラメタ付きモジュールに変換され、そのパラメタが親クラスになるのです。多重継承はモジュールのパラメタの数を増やすことで実現します。15 章で述べた、点と色付きの点の古典的な例題を取り上げ、これをモジュールに変更します。

クラス point は、以下のシグネチャPOINT を持つモジュール Point となります。

```
# module type POINT =
    sig
      type point
      val new_point : (int * int) -> point
      val get_x : point -> int
      val get_y : point -> int
      val moveto : point -> (int * int) -> unit
      val rmoveto : point -> (int * int) -> unit
      val display : point -> unit
```

```
    val distance : point → float
  end ;;
```

クラス colored_point は、シグネチャPOINT のパラメタを持つパラメタ付きモジュール ColoredPoint に変換できます。

```
# module ColoredPoint = functor (P : POINT) →
  struct
    type colored_point = {p:P.point;c:string}
    let new_colored_point p c = {p=P.new_point p;c=c}
    let get_c self = self.c
  (* begin *)
    let get_x self = let super = self.p in P.get_x super
    let get_y self = let super = self.p in P.get_y super
    let moveto self = let super = self.p in P.moveto super
    let rmoveto self = let super = self.p in P.rmoveto super
    let display self = let super = self.p in P.display super
    let distance self = let super = self.p in P.distance super
  (* end *)
    let display self =
      let super = self.p in P.display super; print_string ("has color "^ self.c)
  end ;;
```

「継承された」宣言を行う手間は、自動的な変換処理か言語の拡張によって、軽減することができるでしょう。再帰的なメソッドの宣言は、ただ一つの宣言 `let rec ... and` に書き直されます。多重継承は複数のパラメタを持つファンクタとなります。再定義の手間は遅延束縛の場合ほど大きくはありません。

今回の変換のしかたでは遅延束縛は実装できません。これを実装するためには、それぞれのフィールドが関数/メソッドの型に対応するレコードを定義する必要があります。

それぞれのモデルの限界

関数型/モジュール型モデルは、コードの変更を促進しつつも厳密なフレームを提供しています。Objective Caml のオブジェクトモデルは、クラスの「二つの見方」という問題に苦しめられています。つまり構造と型という二つの見方により、オーバーローディングが存在せず、上位クラスが下位クラスに型制約を課すことが出来ないことです。

モジュール

関数型/モジュール型の基本的な限界は、型の拡張が困難な点にあります。抽象型を用いると型の具体的な実装を切り放すことができますが、パラメタ付きモジュールで抽象型を使う場合には型の等価性を書き下す必要がありシグネチャの記述が複雑になります。

再帰的な依存性。 アプリケーションにおけるモジュールの依存関係のグラフは、有向非循環グラフ（*DAG*）になります。これは、二つのモジュール間で相互に再帰的な型が存在しないことを意味する一方で、相互に最適な値の宣言ができないことを意味します。

シグネチャ記述の難しさ。 型推論の魅力の一つは、関数のパラメタの型を指定する必要がないことです。しかし、シグネチャの仕様はこの便利さを犠牲にしています。「手で」シグネチャの宣言の型を指定することが必要になるのです。コンパイラ `ocamlc` の `-i` オプションを用いることで、`.ml` ファイルの中の大域宣言の全ての型を表示し、この情報を用いてモジュールのシグネチャを作ることができます。この場合には、モジュールの実装を与える前に仕様を定義するという「ソフトウエア工学」の原則を失うことになります。加えて、もしシグネチャとモジュールに大きな変更を加える場合に、シグネチャの編集にまで立ち戻る必要があります。パラメタ付きモジュールには、そのパラメタのためのシグネチャが必要で、これを手で記述する必要があります。最後に、もし関数のシグネチャをパラメタ付きモジュールと関連づけた場合、ファンクタの適用から生じるシグネチャの復帰が不可能になります。このため、非関数型のシグネチャのほとんどを書き下し、それをあとで関数型シグネチャを組み立てるために残しておく必要があります。

モジュールのインポートとエクスポート 単純モジュールの宣言のインポートは、ドット記法（`Module.name`）か、モデルがオープンされている（**`open`** `Module`）場合には、直接、宣言の名前（`name`）を指定することで行われます。インポートされたモジュールのインターフェースの宣言は、定義しているモジュールレベルで直接エクスポートできません。インポートすることでインポートされた宣言にアクセスすることはできますが、インポートされた宣言そのものがその宣言を行ったことにはなりません。継承のシュミレーションの時と同じやり方で、インポートされた値の宣言を行う必要があります。これはパラメタ付きモジュールでも同様です。モジュールパラメタの宣言は、現在のモジュールの宣言とはみなされません。

オブジェクト

Objective Caml のオブジェクトモデルの基本的な限界は型付けにあります。

- メソッドは自由な型のパラメタを持つことはできません。
- あるクラスのメソッドがそのクラスの型から切り放すのが困難です。
- 上位型から下位型に対する型制約がない。
- オーバーロードがありません。

Objective Caml のオブジェクトを拡張を始める際にもっとも気持が悪い点は、型パラメタが自由であるパラメタ型をもつメソッドを定義できない点です。クラスの宣言は新しい型の宣言とみなすことができますから、型の宣言において自由な型を持つ変数の存在を禁止するという一般的な規則が必要になります。このことから、パラメタ付きクラスは Objective Caml のオブジェクトモデルではなくてはならないものになっています。パラメタ付きクラスによりその型変数を結び付けることができるからです。

オーバーロードがない。 Objective Caml のオブジェクトモデルはメソッドオーバーロードを許していません。オブジェクトの型はメソッドの型に対応するので、同じ名前で異なる型を持つ多くのメソッドを処理すると、システムが動的にしか解決できないパラメタ多相のために、多くの曖昧さが生じます。これは、静的な型付けという見方と完全に矛盾しているように見えます。整数のパラメタと浮動小数点のパラメタとを持つ二

つのメソッド message を持つクラス example を取り上げます。e をこのクラスのインスタンス、f を以下のような関数とします。

```
# let f x a = x#message a ;;
```

f e 1 と f e 1.1 の二つの呼び出しを静的に解決することはできません。関数 f のコード中にクラス example に関する情報が一切ないからです。

初期化。 クラスで宣言されるインスタンス変数の初期化は、コンストラクタに渡された値に基づいて計算される必要がある場合に問題になり得ます。

インスタンスの等価性。 二つのオブジェクトに適用できる等価性は、物理的な等価性だけです。物理的に異なる二つのオブジェクトに構造等価が適用された場合、常に false を返します。同じクラスの二つのインスタンスが同じメソッドテーブルを共有していることから、これは驚くべきことかもしれません。メソッドテーブルに対する物理的なテストと、オブジェクトの値（val）に対する構造テストを考えることができます。これは、線形パターンマッチクングの実装上の選択です。

クラス階層。 配布される言語システムにはクラス階層がありません。実際、付属するライブラリ群は、単純モジュールがパラメタ付きモジュールの形式です。これは、言語のオブジェクト拡張が安定化の最中であり、広範囲に使われている事例が少ないことを示しています。

コンポーネントの拡張

データの集合とそのデータに対するメソッドをコンポーネントと呼ぶことにします。関数型/モジュール型モデルでは、コンポーネントは型の定義とその型に対する操作からなります。同様に、オブジェクトモデルでは、コンポーネントはクラス階層からなり、そのクラス階層（ただ）一つのクラスから継承をしているので、そのクラス階層中では、そのクラスの振る舞いの全てを共有します。コンポーネントの拡張性の問題は、その振る舞いを拡張したい一方で、操作されるデータも拡張したいのですが、これをソースコードの変更をせずに行いたいということです。例えば、コンポーネント image は、描画（draw）したり（move）移動したりできる長方形（rectangle）か円形（circle）です。

	rectangle	circle	group
draw	X	X	
move	X	X	
grow			

コンポーネント image にメソッド grow を拡張し、いろいろなイメージの集まりを扱いたいでしょう。二つのモデルの振る舞いは拡張性の方向、つまりデータかメソッドかに依存して異なっています。まず、それぞれのモデルでコンポーネント image の共通部分を定義し、そしてそれを拡張してみましょう。

関数型モデルの場合

二つの場合を持つバリアント型として、型 *image* を定義します。メソッドは型 *image* のパラメタを一つとり、要求されるアクションを実行します。

```
# type image = Rect of float | Circle of float ;;
# let draw = function Rect r → ... | Circle c → ... ;;
# let move = ... ;;
```

その後で、この大域宣言を単純モジュールの中にカプセルすることができるでしょう。

メソッドの拡張

メソッドの拡張はモジュール内の型 *image* の表現に依存します。もしこの型が抽象型ならメソッドの拡張を行うことはできません。具体型の場合には、パターンマッチングによって長方形か円形かを選択し、その縮尺を変更する関数 `grow` を追加するのは簡単です。

データ型の拡張

データ型の拡張は、型 *image* では達成できません。実際、Objective Caml の型は、例外を表す型 *exn* 以外は拡張することができません。同じ型のままでデータを拡張することはできないので、新しい型 *n_image* を以下のように定義する必要があります。

```
type n_image = I of image | G of n_image * n_image;;
```

このように、継承の一種をシミュレートしながら、新しい型に対してメソッドを再定義する必要があります。多くの拡張がある場合、これは複雑になります。

オブジェクトモデルの場合

二つの抽象メソッド `draw` と `move` を持つ抽象クラス `image` のサブクラスである、二つのクラス `rectangle` と `circle` を定義します。

```
# class virtual image () =
    object(self:'a)
      method virtual draw : unit → unit
      method virtual move : float * float → unit
    end;;
# class rectangle x y w h =
    object
      inherit image ()
      val mutable x = x
      val mutable y = y
      val mutable w = w
      val mutable h = h
      method draw () = Printf.printf "R: (%f,%f) [%f,%f]" x y w h
      method move (dx,dy) = x <- x +. dx; y <- y +. dy
    end;;
```

```
# class circle x y r =
    object
      val mutable x = x
      val mutable y = y
      val mutable r = r
      method draw () = Printf.printf "C: (%f,%f) [%f]" x y r
      method move (dx, dy)  = x <- x +. dx; y <- y +. dy
    end;;
```

次のプログラムはイメージのリストを作り、それを表示します。

```
# let r = new rectangle 1. 1. 3. 4.;;
val r : rectangle = <obj>
# let c = new circle 1. 1. 4.;;
val c : circle = <obj>
# let l =  [ (r :> image); (c :> image)];;
val l : image list = [<obj>; <obj>]
# List.iter (fun x → x#draw(); print_newline()) l;;
R: (1.000000,1.000000) [3.000000,4.000000]
C: (1.000000,1.000000) [4.000000]
- : unit = ()
```

データ型の拡張

データは、次のようにイメージを表すクラスのサブクラスを追加することで容易に拡張できます。

```
# class group i1 i2 =
    object
      val i1 = (i1:#image)
      val i2 = (i2:#image)
      method draw () = i1#draw(); print_newline (); i2#draw()
      method move p  = i1#move p; i2#move p
    end;;
```

クラス *group* が継承の外でクラス *image* に依存しているので、「型」*image* が再帰的になったことに注意してください。

```
# let g = new group (r:>image) (c:>image);;
val g : group = <obj>
# g#draw();;
R: (1.000000,1.000000) [3.000000,4.000000]
C: (1.000000,1.000000) [4.000000]- : unit = ()
```

メソッドの拡張

新しいメソッドを持つ *image* の抽象サブクラスを定義します。

```
# class virtual e_image () =
    object
      inherit image ()
      method virtual surface : unit → float
    end;;
```

e_image から継承した二つのクラス *rectangle* と *circle* からそれぞれ継承したクラス *e_rectangle* と *e_circle* を定義できます。そして、この新しいメソッドを使用するために拡張されたイメージを使って作業ができます。残った困難な点は *group* です。これは型 *image* を持つ二つのフィールドからなるので、クラス *e_image* から継承しているときでも、イメージフィールドにメッセージ *grow* を送ることができないのです。このようにして、再帰型に対応するサブクラスの場合を除いてメソッドの拡張を行うことができます。

データとメソッドの拡張

両方の方法で拡張を行うためには、パラメタ付きクラスのために再帰型を定義する必要があります。クラス *group* を再定義します。

```
# class ['a] group i1 i2 =
    object
      val i1 = (i1:'a)
      val i2 = (i2:'a)
      method draw () = i1#draw(); i2#draw()
      method move p = i1#move p; i2#move p
    end;;
```

そして、クラス *e_image* に対しても同じ原則を適用します。

```
# class virtual  ext_image () =
    object
      inherit image ()
      method virtual surface : unit → float
    end;;
# class ext_rectangle x y  w h =
    object
      inherit ext_image ()
      inherit rectangle x y w h
      method surface () = w *. h
    end;;
# class ext_circle x y  r=
    object
      inherit ext_image ()
      inherit circle x y r
      method surface () = 3.14 *. r *.r
    end;;
```

クラス *group* の拡張は以下のようになります。

```
# class ['a] ext_group ei1 ei2 =
```

```
    object
      inherit image()
      inherit ['a] group ei1 ei2
      method surface () = ei1#surface() +. ei2#surface ()
    end;;
```

型 *ext_image* のリスト le を作るプログラムは以下のようになります。

```
# let er = new ext_rectangle 1. 1. 2. 4. ;;
val er : ext_rectangle = <obj>
# let ec = new ext_circle 1. 1. 8.;;
val ec : ext_circle = <obj>
# let eg = new ext_group er ec;;
val eg : ext_rectangle ext_group = <obj>
# let le = [ (er:>ext_image); (ec :> ext_image); (eg :> ext_image)];;
val le : ext_image list = [<obj>; <obj>; <obj>]
# List.map (fun x → x#surface()) le;;
- : float list = [8; 200.96; 208.96]
```

一般化

メソッドの拡張を一般化するためには、メソッド handler の中の関数を統合し、そのメソッドの型を返すパラメタ付きクラスを作るのが望ましいでしょう。このため、次のクラスを定義します。

```
# class virtual ['a]  get_image (f: 'b → unit → 'a) =
    object(self: 'b)
      inherit image ()
      method handler () = f(self) ()
    end;;
```

そして、次のクラスはそのインスタンスを生成するために、関数パラメータが追加されています。

```
# class ['a] get_rectangle f x y  w h =
    object(self: 'b)
      inherit ['a] get_image f
      inherit rectangle x y w h
      method get = (x,y,w,h)
    end;;
# class ['a] get_circle f x y  r=
    object(self: 'b)
      inherit ['a] get_image f
      inherit circle x y r
      method get = (x,y,r)
    end;;
```

クラス *group* の拡張版は、二つの型パラメータを取ります。

```
# class ['a,'c] get_group f eti1 eti2 =
    object
```

```
    inherit ['a] get_image f
    inherit ['c] group eti1 eti2
    method get = (i1,i2)
  end;;
```

get_image のインスタンスを拡張します。

```
# let etr = new get_rectangle
    (fun r () → let (x,y,w,h) = r#get in w *. h) 1. 1. 2. 4. ;;
val etr : float get_rectangle = <obj>
# let etc = new get_circle
    (fun c () → let (x,y,r) = c#get in 3.14 *. r *. r) 1. 1. 8.;;
val etc : float get_circle = <obj>
# let etg = new get_group
    (fun g () → let (i1,i2) = g#get in i1#handler() +. i2#handler())
    (etr :> float get_image) (etc :> float get_image);;
val etg : (float, float get_image) get_group = <obj>
# let gel = [ (etr :> float get_image) ; (etc :> float get_image) ;
              (etg :> float get_image) ];;
val gel : float get_image list = [<obj>; <obj>; <obj>]
# List.map (fun x → x#handler()) gel;;
- : float list = [8; 200.96; 208.96]
```

関数型モデルと組み合わされたとき、オブジェクトモデルでは、データとメソッドの拡張がより簡単になります。

混在した構成

前節の最後の例は、ソフトウエア部品の拡張性の問題に対して二つのモデルを混在して使用する方法の利点を示しています。ここでは、パラメタ付きモジュールと遅延束縛、二つの機能のそれぞれの力を利用するために、これらを混在する方法を提案しました。ファンクタの適用によって、パラメタ付きモジュールの型と関数を用いるクラスを含む新しいモジュールが生成されます。さらに、もしそれによって得られたシグネチャが、パラメタ付きモジュールのシグネチャと一致している場合には、結果のモジュールに対してパラメタ付きモジュールを再適用することができるので、新しいクラスを自動的に作ることができます。その具体例は、並行/分散プログラミングを取り扱う本書の第四部（653ページ）で示します。ファンクタを用いて、データ型から通信プロトコルを生成します。二番目のファンクタによって、このプロトコルから、そのプロトコルで表現されるリクエストを処理する一般的なサーバを実装するクラスを導くことができます。そして、そのサーバを継承を用いて実際に要求されるサービスへと特殊化するのです。

練習問題

データ構造へのクラスとモジュール

古典的なデータ構造のためのファンクタの適用を元にしてクラス階層をつくりたいとします。

以下のストラクチャを定義します。

```
# module type ELEMENT =
    sig
      class element :
         string →
         object
           method to_string : unit → string
           method of_string : string → unit
         end
    end ;;

# module type STRUCTURE =
    sig
      class ['a] structure :
        object
          method add : 'a → unit
          method del : 'a → unit
          method mem : 'a → bool
          method get : unit → 'a
          method all : unit → 'a list
          method iter : ('a → unit) → unit
        end
    end ;;
```

1. 型 ELEMENT と型 STRUCTURE の二つのパラメータ M1 と M2 を持つを記述しなさい。constructing a sub-class of *['a] structure* in which *'a* is constrained to *M1.-element*.
2. シグネチャELEMENT を満たす、単純モジュール Integer を記述しなさい。
3. シグネチャSTRUCTURE を満たす、単純モジュール Stack を記述しなさい。
4. 上記ファンクタに二つのパラメタを適用しなさい。
5. ファンクタを、二つのメソッド to_string と of_string を持つように変更しなさい。
6. 上記ファンクタに二つのパラメータを適用し、その結果に適用しなさい。

抽象型

前問につづいて、シグネチャELEMENT を持ち、モジュールを実装することを考えましょう。

以下のパラメタ付き型を定義します。

```
# type 'a t = {mutable x : 'a t; f : 'a t -> unit};;
```

1. 関数 apply、from_string、to_string を記述しなさい。from_string と to_string は、Marshal モジュールを使用しなさい。
2. 型 t を抽象化することで推論されるシグネチャに対応する、シグネチャ S を記述しなさい。
3. シグネチャS を持つパラメタを取り、シグネチャELEMENT を持つモジュールを返すファンクタを記述しなさい。
4. これを、前問のモジュールのパラメータとして適用しなさい。

まとめ

本章では、関数型/モジュール型モデルとオブジェクトモデル、それぞれによるソフトウエア構造の利点を比較しました。この二つのモデルは、ソフトウエアの再利用性と変更可能性の問題に、それぞれのやり方で対処しています。二つのモデルの大きな違いは、型システム・ファンクタのパラメータの型同一性と、オブジェクトモデルのサブタイプ・遅延束縛によるオブジェクトの評価などから生じています。二つのモデルは、それぞれ単体では、ソフトウエア部品の拡張性の問題の解決に成功してはいません。そこから、この二つのモデルを混在させるというアイデアを得ました。この混在構成は、プログラムの新しい構成をも許容しています。

さらに学びたい人のために

モジュールモデルは、コードの再利用性とインクリメンタルな開発が困難な点が問題になっています。論文 "Modular Programming with overloading and delayed linking" ([AC96]) は、オーバーローディングと同様にモジュールの拡張ができるようなモジュール言語の拡張について述べています。オーバーロードされた関数のためのコードの選択は、CLOS における汎関数 (generic function) のために用いられた手法に由来しています。この拡張されたモジュールに対応するための型システムの修正は、証明されていません。

複数のモデルを混在使用については、論文"Modular Object-Oriented Programming with Units and Mixing"([FF98]) でも、コードの再利用性のしやすさという点から、論じられています。特に、ソフトウエア部品の拡張性の問題は詳細に述べています。

この論文は、以下のアドレスから HTML 形式で入手可能です。

リンク: http://www.cs.rice.edu/CS/PLT/Publications/icfp98-ff/paper.shtml

これらの概念には、型の制約や型の衝突の解消などで動的な型付けを含んでいるとがわかるでしょう。漸次的開発を促進することでコードの再利用性の増加を追求する過程で、静的型付け規則を緩めて「基本的には」静的な型付けされる言語を得ることは、おそらく不合理ではありません。

17

アプリケーション

本章では、二つの例を通してプログラムの構造を説明します。一つ目の例ではモジュールモデルを用い、二つ目の例ではオブジェクトモデルを用います。

最初のアプリケーションは二人ゲームのためのパラメタ化されたモジュールの集まりを提供します。ファンクタを用いて探索木の評価のためのミニマックス-$\alpha\beta$ アルゴリズムを実装します。二つ目のファンクタにより、ゲームのマン・マシン インターフェースの変更が可能になります。そして、これらのパラメタ化されたモジュールは二つのゲームに適用されます。一つ目が三目並べ（tic-tac-toe）で、二つ目は神秘的な ley-line の作成を含んでいます。

二つ目のアプリケーションでは、ロボットが進化する仮想世界を作ります。この仮想世界ではロボットはクラスとして構造化されます。ロボットの異なる振る舞いは、共通の抽象クラスから継承することで得られます。これにより新しい振る舞いを定義するのが簡単で、また、マン・マシン インターフェースの変更も可能になっています。

アプリケーションはそれぞれ、その構造の中に再利用可能なコンポーネントを含んでいます。そのため、同じ基底クラスを用いた異なるルールの新しい二人プレーヤー用ゲームを作るのは簡単です。同様に、仮想世界のロボットの動きのための一般的な機構は、新しいタイプのロボットにも適用できます。

二人ゲーム

本節で紹介するのアプリケーションでは、二つの目的があります。。一つ目の目的は、Objective Caml が記号処理的なアプリケーションを扱うために便利な道具を提供していることと示すのと同様に、状態空間の探索における複雑さに関連する問題を解く方法を探します。二つ目の目的は、探索の一部をパラメタ化し、また、評価関数やゲームの位置の表示関数のようなコンポーネントをカスタマイズしやすくする、二人プレーヤー用ゲームの作成の一般的な枠組みを定義するために、パラメタ化されたモジュールを使う利点を追求します。

まず、二人プレーヤーを含むゲームの問題を示してから、可能な指手の木を効率よく探索することが可能になる**ミニマックス**-$\alpha\beta$ アルゴリズムについて述べます。そして、これらのファンクタを二つのゲーム、「Connect Four」（垂直三目並べ）と Stonehenge（ley-lines の作成を含む）に適用します。

二人プレーヤー用ゲームの問題

二人プレーヤーを含むゲームは、記号処理プログラミングの古典的なアプリケーションの代表的なものの一つで、少なくとも二つの理由で問題解決のよい例を提供しています。

- 全解探索（brute force）以外の方法で、可能な指手の中から最適解を得るために大量の解を解析する必要があります。
  例えば、チェスの場合、普通、可能な指し手はだいたい 30 ほどであり、一ゲームは各プレーヤごとに 40 手前後です。よって、一プレーヤーの完全な指手の木を探索するには、30^{80} 程度の探索木が必要になるでしょう。
- 解の質が検証可能です。特に、あるプログラムから提案された解の質を、他のプログラムから提案された解と比べることでテストすることができます。

まず、出発点として特定の正しい指手の状態を与えられてたとして、全ての可能な指手の完全なリストを探索できるものと仮定します。このようなプログラムでは、それぞれの指し手に対する「スコア」を評価する関数と同様に、開始地点に基づき正しい指し手を生成する関数が必要になります。評価関数は、勝利する指し手の状態には最高値を、敗北する指し手の状態には最低値をつける必要があります。初期状態の指し手を選択すれば可能な全ての指し手の木を構成することができます。この木では、ノードが指し手の状態に対応し、隣り合うノード同士はある局面から一手指すことで得られ、勝ち負けあるいは結果なしの示す局面を持つ葉を持ちます。一度、木が構成されれば、その木を探索することで、勝利するためのルートがあるのか、あるいはないのか、つまりゲームに負けるのかが決定できます。そして、最短パスが目的とするゴールを得るために選択されます。

このような木の全体の大きさは、完全に表現するには一般的に大きすぎるので、木なかでの実際に作る部分を制限する必要があります。最初の戦略は探索木の「深さ」を制限すること、つまり、評価されるべき指し手とそれへの返し手の数を制限することです。このようにして、木の高さと同様に木の幅も減らすことができます。これらの方法では、葉のノードはゲームの終盤近くになるまで見つかることは稀でしょう。

一方、さらなる評価が必要なために選択された指し手の数を制限する方法もあり得ます。このためには、最も適切な指し手以外の評価はさけ、明らかに最前の指し手と思われるものから試して行きます。これにより、木の枝を全て取り除くことができます。これが次節で説明する**ミニマックス** *(minimax)* $\alpha\beta$ アルゴリズムです。

ミニマックス $\alpha\beta$

本節では、*minimax* 探索と $\alpha\beta$ カットを用いたその改良版について述べます。このアルゴリズムの実装には、ゲームの表現形式とその評価関数とともに、パラメタ付きモジュール `FAlphabeta` を用います。ゲームのプレーヤーを A、B として区別します。

ミニマックス

ミニマックス *(minimax)* アルゴリズムは、探索を終了する深さに制限がある深さ優先の探索アルゴリズムです。ミニマックスアルゴリズムには次の二つの関数が必要です。

- 現在の局面から可能な指し手を生成する関数と、
- ゲームの局面を評価する関数

ある正しい局面からはじめて、このアルゴリズムは、全ての正しい指し手からなる木を必要な深さまで探索します。木の葉には評価関数を用いて計算されたスコアが付加されます。正のスコアはプレーヤー A にとって良い局面であることを表し、負のスコアはプレーヤー A にとって良くない局面であること、つまりプレーヤー B にとって良い局面であることを表します。それぞれのプレーヤーにとって、ある局面から他の局面への遷移は、（プレーヤー A）にとって最大化されるか（プレーヤー B にとって）最小化されるかのいずれかです。各々のプレーヤは自分にとって利益が最大となるように指し手を選択します。プレーヤー A にとって最良のプレーをするためには、深さ 1 の探索では、その指し手による新しい局面でのスコアが最大となるような指し手を決定するようにします。

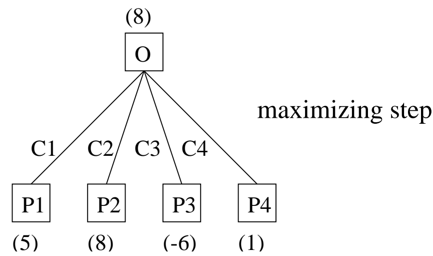


図 17.1: ある局面における探索の最大化。

図 17.1 では、プレーヤー A は局面 O からはじめて、四つの正しい指し手を探し、この木を作成、評価します。このスコアに基づき、最良の局面はスコア 8 で P2 となります。この値は局面 0 に伝播し、この局面からプレーヤー A が C2 に移動した場合にスコア 8 を得るような、新しい局面への指し手があることを示しています。一般的に深さ 1 の探索は、敵のありうる反応を考慮していないので、不十分です。このような浅い探索は、プログラムが、駒が守られているのを認識しないで、あるいは、（チェックメイトのためにクィーンを差し出すような序盤の指し手のように）何か駒を失う結果になるのを認識しないで、（チェスにおけるクィーンの捕獲のような）直接的、具体的な利益を欲深く探索する結果となります。探索の深さを 2 にすることで、少なくとも、このような簡単な反手を認識することができます。

図 17.2 は、プレーヤー B のありうる反応を考慮にいれた探索木の補助的な解析を表しています。この探索では、プレーヤー B の最善手を考慮にいれています。このため、*minimax* アルゴリズムは深さ 2 のスコアを最小にしています。

スコア 8 の局面となる指し手 P2 は結局スコア-3 の局面になってしまいます。結果として、プレーヤー B が D5 をを指せば、Q5 のスコアは-3 となります。このような熟考に基づき、指し手 C1 がスコア-1 で損失を抑えられるの、良い指し手となります。

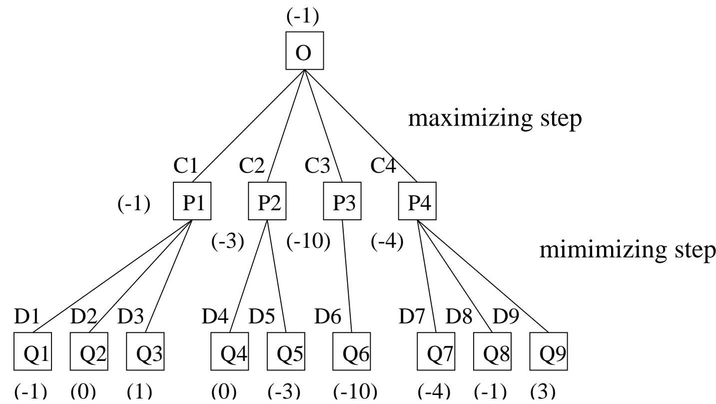


図 17.2: 深さ 2 の探索における最大化と最小化。

ほとんどのゲームでは、敵に打つ指し手強制したり、敵が指し手を間違えることを期待して局面をグチャグチャにするようにして、敵を混乱させることができます。深さ 2 の浅い探索では、この種の戦法には全く不十分でしょう。この種の戦略がプログラムによって良く探索されることはめったにありません。ゲームの終盤に向かうありそうな局面の進化に関して何の展望も持っていないからです。

探索の深さが増えるにつれて組み合わせ「爆発」の形で難しさが生じます。例えば、チェスでは、先読みを 2 手増やすと、組み合わせが 1000 倍（30×30）程度増加します。よって、もし深さ 10m で探索すると、局面は 5^{14} になり、探索するは大きすぎることになってしまいます。このため、何とかして探索木を刈り取る必要があります。

図 17.2 で、深さ 1 におけるこの局面のスコアが枝 P1 で見つかった局面のスコアよりも悪い限りにおいて、枝 P3 を探索するのは意味がないかも知れないことに注意してください。加えて、枝 P4 は完全には探索される必要はありません。Q7 の計算に基づき、すでに完全に計算してある P1 よりも悪いスコアとなります。Q8 と Q9 のための計算は、Q7 よりもスコアが良くても、この状況では改善できません。モードを最小化する際には、最も低いスコアは落されます。そして、このような枝には新しい指し手の可能性がないことがわかります。ミニマックスの変種 $\alpha\beta$ は、探索する必要のある枝の数を減らすためにこのアプローチを用います。

ミニマックス-$\alpha\beta$

スコアを最大にするノードの**下限**を α 切断（α cut）と呼び、スコアを最小にするノードの**上限**を β 切断（β cut）と呼びます。図 17.3 は、枝 P1 の下限が-1 だという知識に基づき、枝 P3 と枝 P4 で行われた切断を表しています。

探索木が広くそして深くなると切断の数は増え、これは部分木が大きいことを表します。

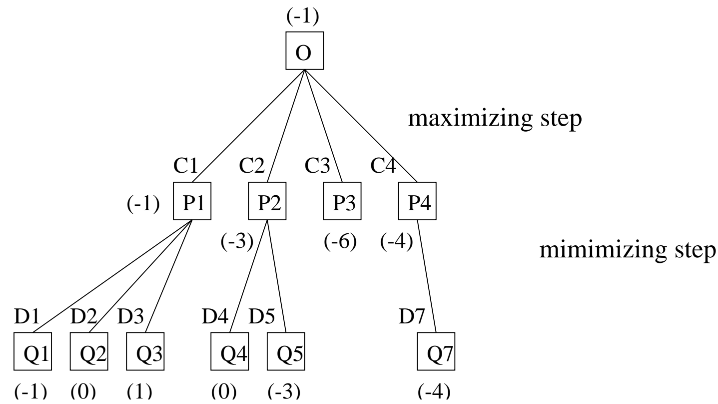


図 17.3: 一段階の max-min への α の制限。

$\alpha\beta$ ミニマックスのためのパラメタ付きモジュール

この種の二人プレーヤー用ゲームに一般的に再利用可能な、このアルゴリズムを実装するパラメタ付きモジュール `FAlphabeta` を作ります。

インターフェース。 プレーヤーを表す `REPRESENTATION` と、局面の評価するための `EVAL`、二つのシグネチャを宣言します。

```
# module type REPRESENTATION =
  sig
    type game
    type move
    val game_start : unit → game
    val legal_moves: bool → game → move list
    val play: bool → move → game → game
  end ;;
module type REPRESENTATION =
  sig
    type game
    and move
    val game_start : unit -> game
    val legal_moves : bool -> game -> move list
    val play : bool -> move -> game -> game
  end

# module type EVAL =
  sig
    type game
    val evaluate: bool → game → int
    val moreI : int
    val lessI: int
    val is_leaf: bool → game → bool
```

```
    val is_stable: bool → game → bool
    type state = G | P | N | C
    val state_of : bool → game → state
  end ;;
module type EVAL =
  sig
    type game
    val evaluate : bool -> game -> int
    val moreI : int
    val lessI : int
    val is_leaf : bool -> game -> bool
    val is_stable : bool -> game -> bool
    type state = G | P | N | C
    val state_of : bool -> game -> state
  end
```

型gameと型moveは抽象型です。プレーヤーは真偽として表現されます。関数legal_movesは、プレーヤーと局面を引数として取り、可能な指し手のリストを返します。関数playは、プレーヤー、指し手、局面を引数として取り、新しい局面を返します。moreIとlessIの値は、関数evaluateによって返される値の限界を表します。述語is_leafは、プレーヤーが与えられた局面で手を打てるかどうかを調べます。述語is_stableは、あるプレーヤへの局面が安定状態にあるかどうかを示します。これらの関数の結果は、指定された深さに至ったときに、探索の追求に影響します。

シグネチャALPHABETAは、使用したいパラメタ付きモジュールの完全な適用結果に対応します。これにより、アルゴリズムの実装に用いる補助的な関数の違いを隠すことができます。

```
# module type ALPHABETA = sig
    type game
    type move
    val alphabeta : int → bool → game → move
  end ;;
module type ALPHABETA =
  sig type game and move val alphabeta : int -> bool -> game -> move end
```

関数alphabetaは、パラメタとして、探索の深さ、プレーヤー、ゲームの局面を取り、次の指し手を返します。

ここで、ファンクタの実装に対応する、関数シグネチャFALPHABETAを定義しましょう。

```
# module type FALPHABETA = functor (Rep : REPRESENTATION)
    → functor (Eval : EVAL with type game = Rep.game)
      → ALPHABETA with type game = Rep.game
                  and type move = Rep.move ;;
module type FALPHABETA =
  functor (Rep : REPRESENTATION) ->
    functor
      (Eval : sig
                type game = Rep.game
```

```
            val evaluate : bool -> game -> int
            val moreI : int
            val lessI : int
            val is_leaf : bool -> game -> bool
            val is_stable : bool -> game -> bool
            type state = G | P | N | C
            val state_of : bool -> game -> state
          end) ->
  sig
    type game = Rep.game
    and move = Rep.move
    val alphabeta : int -> bool -> game -> move
  end
```

実装。 パラメタ付きモジュール FAlphabetaO は、二つのパラメータ Rep と Eval の間の、型 game の分割を明示的にします。このモジュールには、六つの関数と二つの例外があります。プレーヤー true はスコアを最大にするために、一方プレーヤー false はスコアを最小にするために探索を行います。関数 maxmin_iter は、プレーヤー true の指し手と枝刈りパラメータ α に基づき、各々の枝について最良のスコアの最大値を計算します。

関数 maxmin は四つのパラメータを取ります。depth は実際の計算の深さ、node はゲームの局面、α と β は枝刈りのパラメータです。もし、ノードが木の葉であるかあるいは深さの最大に達していたら、この関数は、その局面の評価値を返します。これ以外の場合では、この関数はプレーヤー true の全ての指し手に対して、探索関数とを渡し、残りの探索の深さ（minmax）を減じてから、maxmin_iter を適用します。後者は、プレーヤー false からの反手の結果のスコアを最小にするような探索を行います。

実際の指しては、例外を用いて実装されます。関数 maxmin から始まる可能な指し手をまがる繰り返しの中で、指し手 β が見つかれば、即座にそれが返され、その値は例外を用いて伝播されます。関数 minmax_iter と minmax は、もう一方のプレーヤ用の同等な関数です。関数 search は、スコアと指し手のリストの中で見つかった最良値に基づいて、指し手を決定します。

このモジュールの主な関数 alphabeta は、指定されたプレーヤーのある局面における可能な指し手を計算し、指定された深さまで探索し、最良手を返します。

```
# module FAlphabetaO
    (Rep : REPRESENTATION) (Eval : EVAL with type game = Rep.game) =
  struct
    type game = Rep.game
    type move = Rep.move
    exception AlphaMovement of int
    exception BetaMovement of int

    let maxmin_iter node minmax_cur beta alpha cp =
      let alpha_resu =
        max alpha (minmax_cur (Rep.play true cp node) beta alpha)
      in if alpha_resu >= beta then  raise (BetaMovement alpha_resu)
```

```
  else alpha_resu

let minmax_iter node maxmin_cur alpha beta cp =
  let beta_resu =
    min beta (maxmin_cur (Rep.play false cp node) alpha beta)
  in if beta_resu <= alpha then raise (AlphaMovement beta_resu)
  else beta_resu

let rec maxmin depth node alpha beta =
  if (depth < 1 & Eval.is_stable true node)
      or Eval.is_leaf true node
  then Eval.evaluate true node
  else
    try  let prev = maxmin_iter node (minmax (depth - 1)) beta
    in List.fold_left prev alpha (Rep.legal_moves true node)
    with BetaMovement a → a

and minmax depth node beta alpha =
  if (depth < 1 & Eval.is_stable false node)
      or Eval.is_leaf false node
  then Eval.evaluate false node
  else
    try let prev = minmax_iter node (maxmin (depth - 1)) alpha
    in List.fold_left prev beta (Rep.legal_moves false node)
    with AlphaMovement b → b

let rec search a l1 l2 = match (l1,l2) with
  (h1 :: q1, h2 :: q2) → if a = h1 then h2  else search a q1 q2
| ([], [] ) → failwith  ("AB: "^(string_of_int a)^" not found")
| (_ ,  _) → failwith  "AB: length differs"

(* val alphabeta : int -> bool -> Rep.game -> Rep.move *)
let alphabeta depth player level =
  let alpha = ref Eval.lessI and beta = ref Eval.moreI in
  let l = ref [] in
  let cpl = Rep.legal_moves player level in
  let eval =
    try
      for i = 0 to (List.length cpl) - 1 do
        if player then
          let b = Rep.play player (List.nth cpl i) level in
          let a = minmax (depth-1) b !beta !alpha
          in l := a :: !l ;
          alpha := max !alpha a ;
          (if !alpha >= !beta then raise (BetaMovement !alpha))
        else
          let a = Rep.play player (List.nth cpl i) level in
          let b = maxmin (depth-1) a !alpha !beta
          in l := b :: !l ;
          beta := min !beta b ;
          (if !beta <= !alpha then raise (AlphaMovement !beta))
```

```
            done ;
            if player then !alpha else !beta
          with
            BetaMovement a → a
          | AlphaMovement b → b
        in
        l := List.rev !l ;
        search eval !l cpl
    end ;;
module FAlphabetaO :
  functor (Rep : REPRESENTATION) ->
    functor
      (Eval : sig
                type game = Rep.game
                val evaluate : bool -> game -> int
                val moreI : int
                val lessI : int
                val is_leaf : bool -> game -> bool
                val is_stable : bool -> game -> bool
                type state = G | P | N | C
                val state_of : bool -> game -> state
              end) ->
      sig
        type game = Rep.game
        and move = Rep.move
        exception AlphaMovement of int
        exception BetaMovement of int
        val maxmin_iter :
          Rep.game ->
          (Rep.game -> int -> int -> int) -> int -> int -> Rep.move -> int
        val minmax_iter :
          Rep.game ->
          (Rep.game -> int -> int -> int) -> int -> int -> Rep.move -> int
        val maxmin : int -> Eval.game -> int -> int -> int
        val minmax : int -> Eval.game -> int -> int -> int
        val search : int -> int list -> 'a list -> 'a
        val alphabeta : int -> bool -> Rep.game -> Rep.move
      end
```

最後に、モジュール FAlphabetaO を次のシグネチャに結び付けます。

```
# module FAlphabeta = (FAlphabetaO : FALPHABETA) ;;
module FAlphabeta : FALPHABETA
```

後者のモジュールは、異なるゲームの表現と異なるゲームを行う関数で利用することができます。

ゲームプログラムの構造

二人プレーヤー用ゲームのためのプログラムの構造は、この種のゲーム全てに適用可能な部分と同様、当該のゲームに固有の部分に分割することができます。このため、特定のモジュールによってパラメタ化される、パラメタ付きモジュールを用いることを提案します。これにより、共通部分をそのたびに書き直す必要がなくなります。図 17.4 が、選択した構成です。

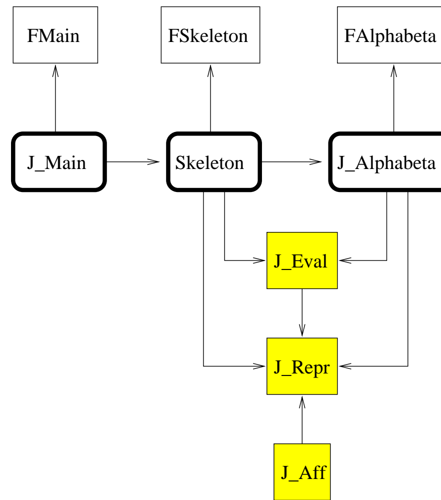


図 17.4: ゲームアプリケーションの構成。

印のついてないモジュールは、アプリケーションで共通の部分に対応します。これらのモジュールはパラメタ付きモジュールです。ここにもモジュール FAlphabeta があります。灰色の印のモジュールは、あるゲームに固有なものとして設計されたものです。三つの主なモジュールは、ゲームの表現（J_Repr）、ゲームの表示（J_Disp）、評価関数（J_Eval）に関するものです。灰色の枠で囲われたモジュールは、パラメタ付きモジュールに、ゲーム固有の単純モジュールを適用することで得られます。

モジュール FAlphabeta についてはすでに述べました。他の二つの共通モジュールは、メインループを含む FMain と、プレーヤーを管理する FSkeleton です。

モジュール FMain

モジュール FMain は、ゲームプログラムを実行するためのメインループを含んでいます。このモジュールは、シグネチャモジュール SKELETON を用いてパラメタ化されています。このシグネチャモジュールは以下の定義を用いてプレーヤーとの相互作用を記述しています。

It is parameterized using the signature module SKELETON, describing the interaction with a player using the following definition:

```
# module type SKELETON = sig
```

```
    val home: unit → unit
    val init: unit → ((unit → unit) * (unit → unit))
    val again: unit → bool
    val exit: unit → unit
    val won: unit → unit
    val lost: unit → unit
    val nil: unit → unit
    exception Won
    exception Lost
    exception Nil
  end ;;
module type SKELETON =
  sig
    val home : unit -> unit
    val init : unit -> (unit -> unit) * (unit -> unit)
    val again : unit -> bool
    val exit : unit -> unit
    val won : unit -> unit
    val lost : unit -> unit
    val nil : unit -> unit
    exception Won
    exception Lost
    exception Nil
  end
```

関数 init は、それぞれのプレーヤのアクション関数の組を生成します。他の関数は、相互作用を制御します。モジュール FMain は二つの関数、プレーヤーを順々に実行する play_game と、メインループを制御する main を含んでいます。

```
# module FMain (P : SKELETON) =
    struct
      let play_game movements = while true do (fst movements) () ;
        (snd movements) () done

      let main () = let finished = ref false
      in P.home ();
      while not !finished do
        ( try  play_game (P.init ())
        with P.Won → P.won ()
        | P.Lost  → P.lost ()
        | P.Nil   → P.nil () );
        finished := not (P.again ())
      done ;
      P.exit ()
  end ;;
module FMain :
  functor (P : SKELETON) ->
    sig
      val play_game : (unit -> 'a) * (unit -> 'b) -> unit
      val main : unit -> unit
```

```
end
```

モジュール FSkeleton

パラメタ付きモジュール FSkeleton は、プレーヤーの性質（自動化されているかいないか）とプレーヤーの順序に基づき、本節の最初で述べた規則に応じて、それぞれのプレーヤの指し手を制御します。ゲーム、ゲームの状態、評価関数、図 17.4 で示した $\alpha\beta$ 検索を表現するにはいろいろなパラメータが必要です。

ゲームの表示に必要となるシグネチャから始めましょう。

```
# module type DISPLAY = sig
    type game
    type move
    val home: unit → unit
    val exit: unit → unit
    val won: unit → unit
    val lost: unit → unit
    val nil: unit → unit
    val init: unit → unit
    val position : bool → move → game → game → unit
    val choice : bool →  game → move
    val q_player : unit → bool
    val q_begin : unit → bool
    val q_continue : unit → bool
  end ;;
module type DISPLAY =
  sig
    type game
    and move
    val home : unit -> unit
    val exit : unit -> unit
    val won : unit -> unit
    val lost : unit -> unit
    val nil : unit -> unit
    val init : unit -> unit
    val position : bool -> move -> game -> game -> unit
    val choice : bool -> game -> move
    val q_player : unit -> bool
    val q_begin : unit -> bool
    val q_continue : unit -> bool
  end
```

ゲームと指し手が、型を持つ、全てのパラメタ付きモジュールで共有されなければならないことは価値がありません。二つの主な関数は playH と playM で、（関数 Disp.choice を用いて）人間のプレーヤーの指し手と、自動化されたプレーヤの指し手をそれぞれ制御します。関数 init は、プレーヤの性質と Disp.q_player のための返答の種類を決定します。

```
# module FSkeleton
      (Rep   : REPRESENTATION)
      (Disp  : DISPLAY with type game = Rep.game and  type move = Rep.move)
      (Eval  : EVAL with type game = Rep.game)
      (Alpha : ALPHABETA with type game = Rep.game and  type move = Rep.move) =
    struct
      let depth = ref 4
      exception Won
      exception Lost
      exception Nil
      let won = Disp.won
      let lost = Disp.lost
      let nil = Disp.nil
      let again = Disp.q_continue
      let play_game = ref (Rep.game_start())
      let exit = Disp.exit
      let home = Disp.home

      let playH player () =
        let choice = Disp.choice player !play_game in
        let old_game = !play_game
        in play_game := Rep.play player choice !play_game ;
        Disp.position player choice old_game !play_game ;
        match Eval.state_of player !play_game with
          Eval.P → raise Lost
        | Eval.G → raise Won
        | Eval.N → raise Nil
        | _      → ()

      let playM player () =
        let choice = Alpha.alphabeta !depth player !play_game in
        let old_game = !play_game
        in play_game := Rep.play player choice !play_game ;
        Disp.position player choice old_game !play_game ;
        match Eval.state_of player !play_game with
          Eval.G → raise Won
        | Eval.P → raise Lost
        | Eval.N → raise Nil
        | _      → ()

      let init () =
        let a = Disp.q_player () in
        let b = Disp.q_player()
        in play_game := Rep.game_start () ;
        Disp.init () ;
        match (a,b) with
          true,true   → playM true, playM false
        | true,false  → playM true, playH false
        | false,true  → playH true, playM false
        | false,false → playH true, playH false
  end ;;
```

```
module FSkeleton :
  functor (Rep : REPRESENTATION) ->
    functor
      (Disp : sig
                type game = Rep.game
                and move = Rep.move
                val home : unit -> unit
                val exit : unit -> unit
                val won : unit -> unit
                val lost : unit -> unit
                val nil : unit -> unit
                val init : unit -> unit
                val position : bool -> move -> game -> game -> unit
                val choice : bool -> game -> move
                val q_player : unit -> bool
                val q_begin : unit -> bool
                val q_continue : unit -> bool
              end) ->
      functor
        (Eval : sig
                  type game = Rep.game
                  val evaluate : bool -> game -> int
                  val moreI : int
                  val lessI : int
                  val is_leaf : bool -> game -> bool
                  val is_stable : bool -> game -> bool
                  type state = G | P | N | C
                  val state_of : bool -> game -> state
                end) ->
        functor
          (Alpha : sig
                     type game = Rep.game
                     and move = Rep.move
                     val alphabeta : int -> bool -> game -> move
                   end) ->
          sig
            val depth : int ref
            exception Won
            exception Lost
            exception Nil
            val won : unit -> unit
            val lost : unit -> unit
            val nil : unit -> unit
            val again : unit -> bool
            val play_game : Disp.game ref
            val exit : unit -> unit
            val home : unit -> unit
            val playH : bool -> unit -> unit
            val playM : bool -> unit -> unit
            val init : unit -> (unit -> unit) * (unit -> unit)
          end
```

The independent parts of the game are thus implemented. One may then begin programming different sorts of games. This modular organization facilitates making modifications to the movement scheme or to the evaluation function for a game as we shall soon see.

Connect Four

次に、Connect Four として知られる簡単なゲーム、垂直三目並べをみてみましょう。このゲームは、それぞれが六つの行からなる七つの列で表されます。プレーヤーは順番に、ある列に各自の色のコマを置くと、コマはその列の一番下まで落ちていきます。もし、その列が一杯なら、どちらのプレーヤーもそれ以上置くことは出来ません。ゲームは、一方のプレーヤーが、（垂直でも、水平でも、斜めでも）四つのコマを一直線に並べたときに終了、そのプレーヤーが勝利するか、全ての列がコマで埋め尽くされたときには引き分けとなります。図 17.5 は、完全なゲームを表しています。

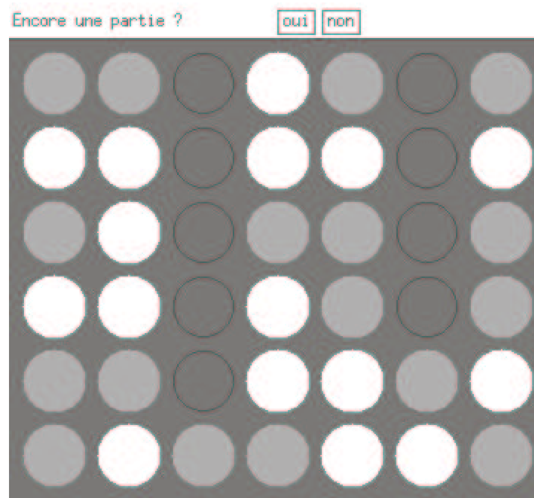


図 17.5: Connect Four の例

四つの灰色のコマが右下へと斜めに並んで勝ちとなっています。

ゲームの表現: モジュール C4_rep。 このゲームは、行列ベースの表現を用いることにします。行列のそれぞれの要素は空か、各プレーヤのコマで占められます。指し手は列により番号付けられます。正しい指し手は最後（一番上）の行が埋まっていない列です。

```
# module C4_rep = struct
    type cell = A | B | Empty
    type game = cell array array
    type move = int
    let col = 7 and row = 6
    let game_start () = Array.create_matrix row col Empty
```

```
    let legal_moves b m =
      let l = ref [] in
      for c = 0 to col-1 do if m.(row-1).(c) = Empty then l := (c+1) :: !l done;
      !l

    let augment mat c =
      let l = ref row
      in while !l > 0 & mat.(!l-1).(c-1) = Empty do  decr l done ; !l + 1

    let player_gen cp m e =
      let mj = Array.map Array.copy  m
      in  mj.((augment mj cp)-1).(cp-1) <- e ; mj

    let play b cp m = if b then player_gen cp m A else player_gen cp m B
  end ;;
module C4_rep :
  sig
    type cell = A | B | Empty
    and game = cell array array
    and move = int
    val col : int
    val row : int
    val game_start : unit -> cell array array
    val legal_moves : 'a -> cell array array -> int list
    val augment : cell array array -> int -> int
    val player_gen : int -> cell array array -> cell -> cell array array
    val play : bool -> int -> cell array array -> cell array array
  end
```

このモジュールが、シグネチャREPRESENTATION の制約を満たすことは簡単に確かめられます。

```
# module C4_rep_T = (C4_rep : REPRESENTATION) ;;
module C4_rep_T : REPRESENTATION
```

ゲームの表示: モジュール C4_text。 モジュール C4_text は、シグネチャDISPLAY とコンパチブルな、ゲーム Connect Four のためのテキストベースのインターフェースを記述しています。このモジュールは、特段、複雑なわけではありませんが、それでもどのようにモジュールが組み合わせられるのかは良くわかるでしょう。

```
# module C4_text = struct
    open C4_rep
    type game = C4_rep.game
    type move = C4_rep.move

    let print_game mat =
      for l = row - 1 downto 0 do
```

```
        for c = 0 to col - 1 do
          match mat.(l).(c) with
            A     → print_string "X "
          | B     → print_string "O "
          | Empty → print_string ". "
        done;
        print_newline ()
      done ;
      print_newline ()

    let home () = print_string "C4 ...\n"
    let exit () = print_string "Bye for now ... \n"
    let question s =
      print_string s;
      print_string " y/n ? " ;
      read_line() = "y"
    let q_begin () = question "Would you like to begin?"
    let q_continue () = question "Play again?"
    let q_player () = question "Is there to be a machine player ?"

    let won  ()= print_string "The first player won" ; print_newline ()
    let lost () = print_string "The first player lost" ; print_newline ()
    let nil () = print_string "Stalemate" ; print_newline ()

    let init () =
      print_string "X: 1st player   O: 2nd player";
      print_newline () ; print_newline () ;
      print_game (game_start ()) ; print_newline()

    let position b c aj j = print_game j

    let is_move = function '1'..'7' → true | _ → false

    exception Move of int
    let rec choice player game =
      print_string ("Choose player" ^ (if player then "1" else "2") ^ " : ") ;
      let l = legal_moves player game
      in try  while true do
        let i = read_line()
        in ( if (String.length i > 0)  && (is_move  i.[0])
        then let c = (int_of_char i.[0]) - (int_of_char '0')
        in if List.mem c l then raise (Move c) );
        print_string "Invalid move - try again"
      done ;
        List.hd l
      with  Move i → i
      | _ → List.hd l
  end ;;
module C4_text :
  sig
    type game = C4_rep.game
```

```
    and move = C4_rep.move
    val print_game : C4_rep.cell array array -> unit
    val home : unit -> unit
    val exit : unit -> unit
    val question : string -> bool
    val q_begin : unit -> bool
    val q_continue : unit -> bool
    val q_player : unit -> bool
    val won : unit -> unit
    val lost : unit -> unit
    val nil : unit -> unit
    val init : unit -> unit
    val position : 'a -> 'b -> 'c -> C4_rep.cell array array -> unit
    val is_move : char -> bool
    exception Move of int
    val choice : bool -> C4_rep.cell array array -> int
  end
```

これが、シグネチャDISPLAY の制約を満たすことはすぐに確かめられます。

```
# module C4_text_T = (C4_text : DISPLAY) ;;
module C4_text_T : DISPLAY
```

評価関数: モジュール C4_eval。 ゲームプレーヤーの質は、主に局面の評価関数に依存しています。モジュール C4_eval は、特定のプレーヤーにおける局面の評価値を求める、関数 evaluate を定義します。この関数は、対角線と同様に四つのコンパス方位に対して、eval_bloc を呼び出します。eval_bloc は、指定された行にいくつのコマが並んでいるのかを計算するために eval_four を呼び出します。表 value は、0/1/2/3 個の同じ色のコマを持つブロックの値を提供します。例外 Four は、四つのコマが並んだときに発生します。

```
# module C4_eval = struct open C4_rep type game = C4_rep.game
    let value =
      Array.of_list [0; 2; 10; 50]
    exception Four of int
    exception Nil_Value
    exception Arg_invalid
    let lessI = -10000
    let moreI = 10000
    let eval_four m l_dep c_dep delta_l delta_c =
      let n = ref 0 and e = ref Empty
      and x = ref c_dep  and  y = ref l_dep
      in try
        for i = 1 to 4 do
          if !y<0 or !y>=row or !x<0 or !x>=col then raise Arg_invalid ;
          ( match m.(!y).(!x) with
            A     → if !e = B then raise Nil_Value ;
              incr n ;
```

```
              if !n = 4 then raise (Four moreI) ;
              e := A
          | B     → if !e = A then raise Nil_Value ;
              incr n ;
              if !n = 4 then raise (Four lessI);
              e := B;
          | Empty → () ) ;
              x := !x + delta_c ;
          y := !y + delta_l
        done ;
        value.(!n) * (if !e=A then 1 else -1)
      with
        Nil_Value | Arg_invalid  → 0

    let eval_bloc m e cmin cmax lmin lmax dx dy =
      for c=cmin to cmax do for l=lmin to lmax do
        e := !e + eval_four m l c dx dy
      done done

    let evaluate b m =
      try let evaluation = ref 0
      in (* evaluation of rows *)
      eval_bloc m evaluation 0 (row-1) 0 (col-4) 0 1 ;
      (* evaluation of columns *)
      eval_bloc m evaluation 0 (col-1) 0 (row-4) 1 0 ;
      (* diagonals coming from the first line (to the right) *)
      eval_bloc m evaluation 0 (col-4) 0 (row-4) 1 1 ;
      (* diagonals coming from the first line (to the left) *)
      eval_bloc m evaluation 1 (row-4) 0 (col-4) 1 1 ;
      (* diagonals coming from the last line (to the right) *)
      eval_bloc m evaluation 3 (col-1) 0 (row-4) 1 (-1) ;
      (* diagonals coming from the last line (to the left) *)
      eval_bloc m evaluation 1 (row-4) 3 (col-1) 1 (-1) ;
      !evaluation
      with Four v → v

    let is_leaf b m = let v = evaluate b m
    in v=moreI or v=lessI or legal_moves b m = []

    let is_stable b j = true

    type state = G | P | N | C

    let state_of player m =
      let v = evaluate player m
      in if v = moreI then if player then G else P
      else if v = lessI then if player then P else G
      else if legal_moves player m = [] then N else C
  end ;;
module C4_eval :
  sig
```

```
    type game = C4_rep.game
    val value : int array
    exception Four of int
    exception Nil_Value
    exception Arg_invalid
    val lessI : int
    val moreI : int
    val eval_four :
      C4_rep.cell array array -> int -> int -> int -> int -> int
    val eval_bloc :
      C4_rep.cell array array ->
      int ref -> int -> int -> int -> int -> int -> int -> unit
    val evaluate : 'a -> C4_rep.cell array array -> int
    val is_leaf : 'a -> C4_rep.cell array array -> bool
    val is_stable : 'a -> 'b -> bool
    type state = G | P | N | C
    val state_of : bool -> C4_rep.cell array array -> state
  end
```

モジュール C4_eval は、シグネチャEVAL とコンパチブルです。

```
# module C4_eval_T = (C4_eval : EVAL) ;;
module C4_eval_T : EVAL
```

二つの評価関数をそれぞれ対戦相手としてゲームをするには、evaluate を適切な評価関数に適用するように変更する必要があります。

モジュールの組み立て Connect Four ゲームを実現するために必要となる全てのコンポーネントが実装されました。後は、図 17.4 に基づいてこれらを組み合わせるだけです。まず、パラメタ付きモジュール FSkeleton を C4_rep、C4_text、C4_eval に適用した C4_skeleton を作り、パラメタ付きモジュール FAlphaBeta を C4_rep と C4_eval に適用します。

```
# module C4_skeleton =
    FSkeleton (C4_rep) (C4_text) (C4_eval)  (FAlphabeta (C4_rep) (C4_eval)) ;;
module C4_skeleton :
  sig
    val depth : int ref
    exception Won
    exception Lost
    exception Nil
    val won : unit -> unit
    val lost : unit -> unit
    val nil : unit -> unit
    val again : unit -> bool
    val play_game : C4_text.game ref
    val exit : unit -> unit
```

```
    val home : unit -> unit
    val playH : bool -> unit -> unit
    val playM : bool -> unit -> unit
    val init : unit -> (unit -> unit) * (unit -> unit)
  end
```

ここで、先の C4_skeleton の適用の結果に、パラメタ付きモジュール FMain を適用することで、主となるモジュール C4_main を得ます。

```
# module C4_main = FMain(C4_skeleton) ;;
module C4_main :
  sig
    val play_game : (unit -> 'a) * (unit -> 'b) -> unit
    val main : unit -> unit
  end
```

ゲームは、関数 C4_main.main に () を適用することで開始します。

ゲームをテストする。 一度、一般的なゲームの枠組みを記述すれば、いろいろな方法でゲームを実行できます。プログラムは単に指し手が正しいかどうかチェックするだけで、二人の人間のプレーヤーが対戦することもできます。一人の人間が、プログラム相手に対戦することもできまし、また、プログラム同士が対戦することもできます。最後の対戦は人間にとってはおもしろくありませんが、人間の反応を待たずにテストを実行するのが簡単になります。以下のゲームはこのシナリオの例です。

```
# C4_main.main () ;;
C4 ...
Is there to be a machine player ? y/n ? y
Is there to be a machine player ? y/n ? y
X: 1st player   O: 2nd player

. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
```

一度、初期局面が決まれば、（プログラムによって制御される）プレーヤ 1 は指し手を計算し、それを実行します。

```
. . . . . . .
. . . . . . .
. . . . . . .
```

```
. . . . . . .
. . . . . . .
. . . . . X .
```

（常にプログラムが制御する）プレーヤー 2 は、それに対する指し手を計算し、ゲームが終了する指し手を見つけるまで、ゲームが進行していきます。この例では、プレーヤー 1 が以下の最終局面に基づいて勝利します。

```
. O O O . O .
. X X X . X .
X O O X . O .
X X X O . X .
X O O X X O .
X O O O X X O
Player 1 wins
Play again(y/n) ? n
Good-bye ...
- : unit = ()
```

グラフィカルインターフェース　ゲームの楽しさを改善するために、シグネチャDISPLAY とコンパチブルな新しいモジュール C4_graph を定義することで、このプログラムのグラフィカルインターフェースを定義します。このモジュールは、グラフィカルウインドウを開き、マウスによって操作できます。このモジュールのソースコードは、CD-ROM（1 ページ参照）のサブディレクトリー Applications にあります。

```
# module C4_graph = struct
    open C4_rep
    type game = C4_rep.game
    type move = C4_rep.move
    let r = 20          (* color of piece *)
    let ec = 10         (* distance between pieces *)
    let dec = 30        (* center of first piece *)
    let cote = 2*r + ec (* height of a piece looked at like a checker *)
    let htexte = 25     (* where to place text *)
    let width = col * cote + ec          (* width of the window *)
    let height = row * cote + ec + htexte (* height of the window *)
    let height_of_game = row * cote + ec   (* height of game space *)
    let hec = height_of_game + 7  (* line for messages *)
    let lec = 3                    (* columns for messages *)
    let margin = 4                  (* margin for buttons *)
    let xb1 = width / 2   (* position x of button1 *)
    let xb2 = xb1 + 30      (* position x of button2 *)
    let yb = hec - margin    (* position y of the buttons *)
    let wb = 25             (* width of the buttons *)
    let hb = 16             (* height of the buttons *)

  (* val t2e : int -> int *)
  (* Convert a matrix coordinate into a graphical coordinate *)
```

```
  let t2e i = dec + (i-1)*cote

(* The Colors *)
  let cN = Graphics.black   (* trace *)
  let cA = Graphics.red     (* Human player *)
  let cB = Graphics.yellow  (* Machine player *)
  let cF = Graphics.blue    (* Game Background color *)
(* val draw_table : unit -> unit :  Trace an empty table *)
  let draw_table () =
    Graphics.clear_graph();
    Graphics.set_color cF;
    Graphics.fill_rect 0 0 width height_of_game;
    Graphics.set_color cN;
    Graphics.moveto 0 height_of_game;
    Graphics.lineto width height_of_game;
    for l = 1 to row do
      for c = 1 to col do
        Graphics.draw_circle (t2e c) (t2e l) r
      done
    done

(* val draw_piece : int -> int -> Graphics.color -> unit *)
(* 'draw_piece l c co' draws a piece of color co at coordinates l c *)
  let draw_piece l c col =
    Graphics.set_color col;
    Graphics.fill_circle (t2e c) (t2e l) (r+1)

(* val augment : Rep.item array array -> int -> Rep.move *)
(* 'augment m c' redoes the line or drops the piece for c in m *)
  let augment mat c =
    let l = ref row in
    while !l > 0 & mat.(!l-1).(c-1) = Empty do
      decr l
    done;
    !l

(* val conv : Graphics.status -> int *)
(* convert the region where player has clicked in controlling the game *)
  let conv st =
    (st.Graphics.mouse_x - 5) / 50 + 1

(* val wait_click : unit -> Graphics.status *)
(* wait for a mouse click *)
  let wait_click () = Graphics.wait_next_event [Graphics.Button_down]

(* val choiceH : Rep.game -> Rep.move *)
(* give opportunity to the human player to choose a move *)
(* the function offers possible moves *)
  let rec choice player  game  =
    let c = ref 0 in
    while not ( List.mem !c (legal_moves player game) ) do
```

```
      c := conv ( wait_click() )
    done;
    !c
(* val home : unit -> unit : home screen *)
  let home () =
    Graphics.open_graph
      (" " ^ (string_of_int width) ^ "x" ^ (string_of_int height) ^ "+50+50");
    Graphics.moveto (height/2) (width/2);
    Graphics.set_color cF;
    Graphics.draw_string "C4";
    Graphics.set_color cN;
    Graphics.moveto 2 2;
    Graphics.draw_string "by Romuald COEFFIER & Mathieu DESPIERRE";
    ignore (wait_click ());
    Graphics.clear_graph()

(* val end : unit -> unit ,  the end of the game *)
  let exit () = Graphics.close_graph()

(* val draw_button : int -> int -> int -> int -> string -> unit *)
(* 'draw_button x y w h s' draws a rectangular button at coordinates *)
(* x,y with width w and height h and appearance s *)
  let draw_button x y w h s =
    Graphics.set_color cN;
    Graphics.moveto x y;
    Graphics.lineto x (y+h);
    Graphics.lineto (x+w) (y+h);
    Graphics.lineto (x+w) y;
    Graphics.lineto x y;
    Graphics.moveto (x+margin) (hec);
    Graphics.draw_string s

(* val draw_message : string -> unit  * position message s *)
  let draw_message s =
    Graphics.set_color cN;
    Graphics.moveto lec hec;
    Graphics.draw_string s

(* val erase_message : unit -> unit  erase the starting position *)
  let erase_message () =
    Graphics.set_color Graphics.white;
    Graphics.fill_rect 0 (height_of_game+1) width htexte

(* val question : string -> bool *)
(* 'question s' poses the question s, the response being obtained by *)
(*    selecting one of two buttons, 'yes' (=true) and 'no' (=false) *)
  let question s =
    let rec attente () =
      let e = wait_click () in
      if (e.Graphics.mouse_y < (yb+hb)) & (e.Graphics.mouse_y > yb) then
        if (e.Graphics.mouse_x > xb1) & (e.Graphics.mouse_x < (xb1+wb)) then
```

```
          true
        else
          if (e.Graphics.mouse_x > xb2) & (e.Graphics.mouse_x < (xb2+wb)) then
            false
          else
            attente()
      else
      attente () in
    draw_message s;
    draw_button xb1 yb wb hb "yes";
    draw_button xb2 yb wb hb "no";
    attente()

(* val q_begin : unit -> bool *)
(* Ask, using function 'question', if the player wishes to start *)
(* (yes=true) *)
  let q_begin () =
    let b = question "Would you like to begin ?" in
    erase_message();
    b

(* val q_continue : unit -> bool *)
(* Ask, using function 'question', if the player wishes to play again *)
(* (yes=true) *)
  let q_continue () =
    let b = question "Play again ?" in
    erase_message();
    b

  let q_player () =
    let b = question "Is there to be a machine player?" in
    erase_message ();
    b
(* val won : unit -> unit *)
(* val lost : unit -> unit *)
(* val nil : unit -> unit *)
(* Three functions for these three cases *)
  let won () =
    draw_message "I won :-)" ; ignore (wait_click ()) ; erase_message()
  let lost () =
    draw_message "You won :-("; ignore (wait_click ()) ; erase_message()
  let nil () =
    draw_message "Stalemate" ; ignore (wait_click ()) ; erase_message()

(* val init : unit -> unit *)
(* This is called at every start of the game for the position *)
  let init  = draw_table

  let position b c aj nj  =
    if b then
      draw_piece (augment nj c) c cA
```

```
      else
        draw_piece (augment nj c) c cB

   (* val drawH : int -> Rep.item array array -> unit *)
   (* Position when the human player chooses move cp in situation j *)
    let drawH cp j =  draw_piece (augment j cp) cp cA

   (* val drawM : int -> cell array array -> unit*)
   (* Position when the machine player chooses move cp in situation j *)
    let drawM cp j =  draw_piece (augment j cp) cp cB
  end ;;
module C4_graph :
  sig
    type game = C4_rep.game
    and move = C4_rep.move
    val r : int
    val ec : int
    val dec : int
    val cote : int
    val htexte : int
    val width : int
    val height : int
    val height_of_game : int
    val hec : int
    val lec : int
    val margin : int
    val xb1 : int
    val xb2 : int
    val yb : int
    val wb : int
    val hb : int
    val t2e : int -> int
    val cN : Graphics.color
    val cA : Graphics.color
    val cB : Graphics.color
    val cF : Graphics.color
    val draw_table : unit -> unit
    val draw_piece : int -> int -> Graphics.color -> unit
    val augment : C4_rep.cell array array -> int -> int
    val conv : Graphics.status -> int
    val wait_click : unit -> Graphics.status
    val choice : 'a -> C4_rep.cell array array -> int
    val home : unit -> unit
    val exit : unit -> unit
    val draw_button : int -> int -> int -> int -> string -> unit
    val draw_message : string -> unit
    val erase_message : unit -> unit
    val question : string -> bool
    val q_begin : unit -> bool
    val q_continue : unit -> bool
    val q_player : unit -> bool
    val won : unit -> unit
```

```
    val lost : unit -> unit
    val nil : unit -> unit
    val init : unit -> unit
    val position : bool -> int -> 'a -> C4_rep.cell array array -> unit
    val drawH : int -> C4_rep.cell array array -> unit
    val drawM : int -> C4_rep.cell array array -> unit
  end
```

また、パラメタ付きモジュールFSkeletonの適用結果である新しいスケルトン（C4_skeletonG）を生成しましょう。

```
# module C4_skeletonG =
    FSkeleton (C4_rep) (C4_graph) (C4_eval) (FAlphabeta (C4_rep) (C4_eval)) ;;
```

表示のためたのパラメータだけが、FSkeleton のテキスト版と異なっています。これによって、グラフィカルユーザーインターフェース版の Connect Four の主なモジュールを作ることができます。

```
# module C4_mainG = FMain(C4_skeletonG) ;;
module C4_mainG :
  sig
    val play_game : (unit -> 'a) * (unit -> 'b) -> unit
    val main : unit -> unit
  end
```

式 C4_mainG.main() の評価により、グラフィカルなウインドウが、図 17.5 のように開いて、ユーザーとの相互作用を制御します。

Stonehenge

Reiner Knizia により作られた Stonehenge は「ley-lines」を含むゲームです。ルールを理解するのは簡単ですが、このゲームにおける興味は、その可能な指し手の数の多さにあります。このゲームのルールは以下の場所にあります。

リンク: http://www.cix.co.uk/~convivium/files/stonehen.htm

ゲームの初期局面は図 17.6 で表されます。

ゲームの表示

ゲームの目的は、15 個のうち少なくとも 8 個の「ley-lines」（clear line）を得ることです。ley-lines に沿った灰色の位置にコマ（巨石:megalith）を置くことで、その行を取ることができます。

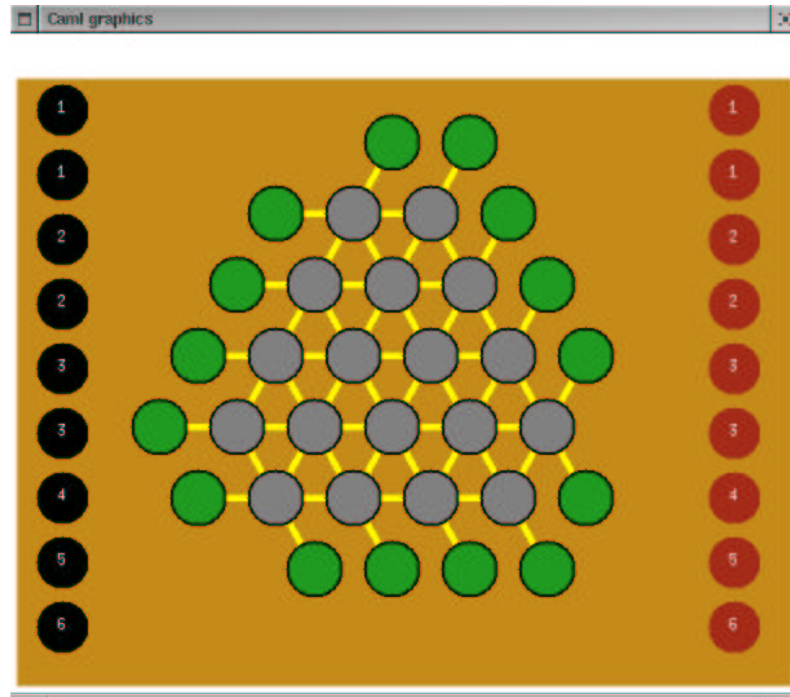


図 17.6: Stonehenge の初期局面

プレーヤーは各々順番に 1 から 6 まで番号を振られた九つのコマを、18 の灰色の内部の位置の一つに置きます。すでに置かれている場所に置くことはできません。コマが置かれるたびに、一つ以上の ley-lines を得るか、あるいは失います。

Ley-line は、一方のプレーヤの線上のコマの価値の合計がもう一方のプレーヤのそれを上回ったら勝ちになります。

例えば、図 17.7 で、黒のプレーヤーが価値 3 のコマを置くことでゲームが開始し、赤のプレーヤが価値「2」のコマを置き、そして黒のプレーヤが「6」を置くことで、この行を取ります。

赤のプレーヤが「4」のコマを置くいてもこの行を取ることができます。この行は完全に埋まってはいないのですが、黒のプレーヤが赤のプレーヤのスコアを上回ることはできないので、赤のプレーヤの勝ちになります。

赤のプレーヤが「4」ではなく「3」を置いても、その行を取ることができます。基本的には、黒の最強のコマが 5 であるこの ley-line の場合には選択できるのは一通りなので、この行に関しては黒のプレーヤが赤のプレーヤを倒すことはできません。

行全体にわたってスコアが等しい場合、最後のコマを置いたプレーヤが相手のスコアよりも強く**なければ**、その行は失うことになります。図 17.8 は、この状況の例です。

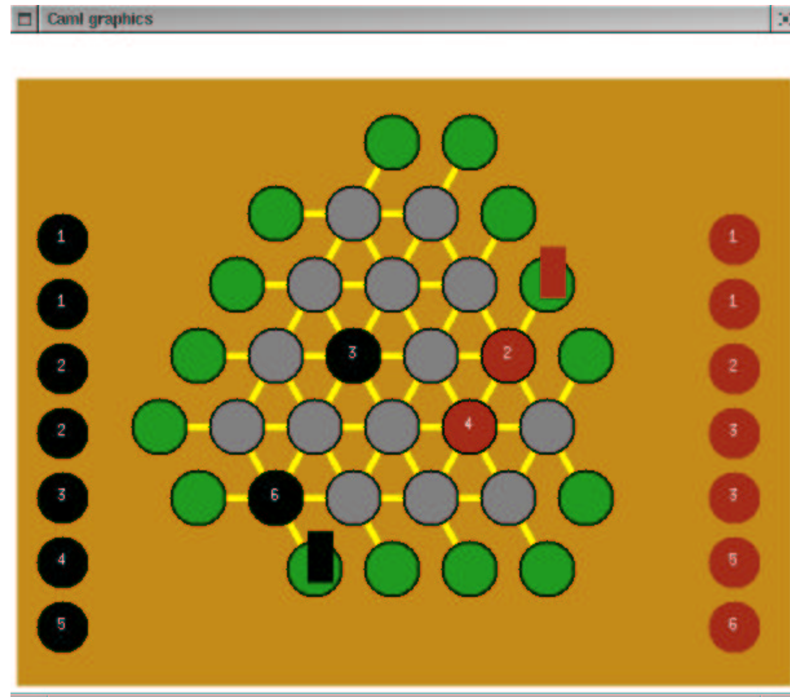


図 17.7: 4 手後の局面

赤の最後のコマは「4」です。「4」が置かれた行では、スコアは同点です。赤がコマを最後に置くのですが、相手を倒すことは出来ないので、黒のブロックで示されるように赤はこの行を失います。

関数 play が、行の配置におけるこの緻密さ調停し占める役割を担っています。

このゲームに引き分けはありません。全部で 15 行あり、それぞれがゲームの時点で一方に占められるので、その時点でプレーヤーは少なくとも 8 行を取って勝ちになります。

探索の複雑さ

この新しいゲームを完全に実装する前に、一方の可能な指し手の数と同様に、ゲーム中の二つの指し手の間で可能な全ての指し手の数を見積もるのは大切です。この値は、ミニマックス-$\alpha\beta$ アルゴリズムが、合理的な最大の深さを見積もるのに用いることもできます。

Stonehenge ゲームでは、一方の可能な指し手の数は、最初二人のプレーヤのコマの数、つまり 18 に基づいています。可能な指し手の数はゲームが進むにつれて減っていきます。最初の指し手では、6 種類のコマと 18 個所の自由な置場所があります。二回目の指し手では、二番目のプレーヤーは、6 種類のコマと、コマを置くことが出来るのが 17 個所です（可能な指し手は 102)。ゲームの最初の局面で深さ 2 から 4 まで指すと、選択肢の数は 10^4 程度から 10^8 程度にまでなります。

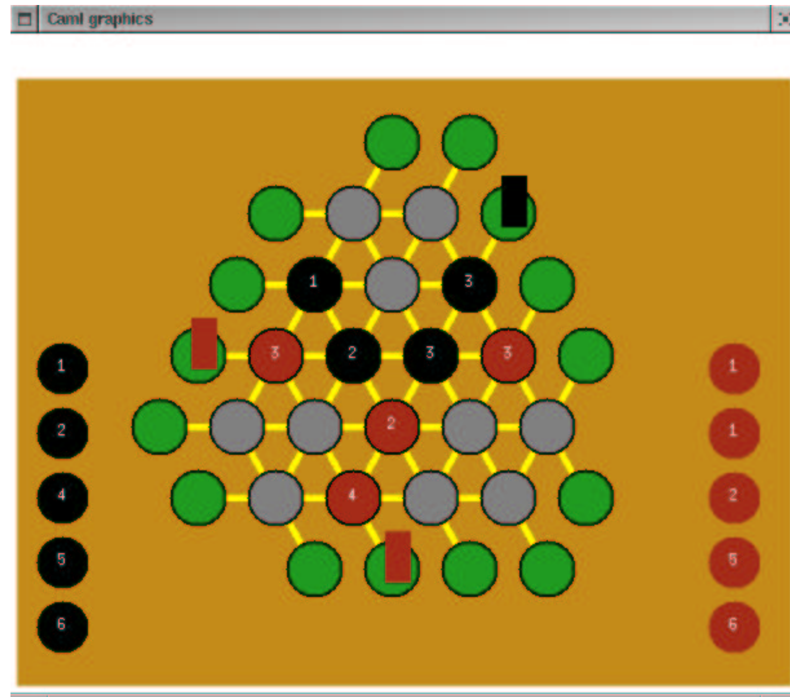


図 17.8: 6 手後の局面。

一方、ゲームの終盤近く、最後の 8 手では、複雑は大きく減ります。(全てのコマが異なるという)悲観的な計算では、二千三百万程度になります。

$$4 * 8 * 4 * 7 * 3 * 6 * 3 * 5 * 2 * 4 * 2 * 3 * 1 * 2 * 1 * 1 = 23224320$$

ゲームの最初の指し手では、深さ 2 程度の計算するのが適切なようです。これは、置くコマがない時には、評価関数と、そのゲーム初盤の局面を評価する能力にも依存します。一方、ゲームの終盤では、深さを 4 から 6 程度まで簡単に増やすことができますが、弱い局面から逆転を狙うにはおそらく遅すぎるでしょう。

実装

まず、ゲームの表現と調停を記述してしまいましょう。そうすればあとは評価関数に集中することができます。

このゲームの実装は、図 17.4 で述べた Connect Four に使われた構成に従っています。二つの主な難しさは、コマの配置に関するゲームの規則に従うことと、評価関数でしょう。この評価関数は、評価値の有効性を保ったまま、出来る限り速く局面を評価できなければ行けません。

ゲームの表現。 このゲームでは四つの重要なデータ構造があります。

- プレーヤのコマ（型 *piece*）、
- 局面 (型 *placement*),
- 15 個の ley-lines、
- 18 個のコマの置場所。

それぞれの場所に固有の番号を振ります。

```
      1---2
     / \ / \
    3---4---5
   / \ / \ / \
  6---7---8---9
 / \ / \ / \ / \
10--11--12--13--14
 \ / \ / \ / \ /
  15--16--17--18
```

それぞれのコマの置場所は、三つの ley-lines に属します。ley-line にもそれぞれ番号を振ります。この記述はリスト *lines* の宣言にありますが、このリストはベクター（vector_l）に変換されます。コマの置場所は、空か置かれたコマとその持ち主の情報を持っています。コマの置場所には、そこを通る行の数も保存します。このテーブルは、lines_per_case によって計算され、num_line_per_case と名付けられます。

このゲームは、18 のケースのベクターで表現され、15 の ley-lines のベクターが勝ったか負けたかを表し、残りのコマのリストが二人のプレーヤを表します。関数 game_start は、これら四つの要素を生成します。

プレーヤーの可能な指し手の計算は、自由に置ける場所に対して利用可能なコマの直積に帰着されます。いろいろな補助関数により、ある行におけるあるプレーヤのスコアを数えたり、ある行の空な置場所の数を計算したり、その行がすでに取られているかどうかを確かめたりすることができます。手を指し、置くべきコマを決定する play を実装するだけでよいことになります。この関数は、モジュール Stone_rep のソースリストの最後に記述します。

```
# module Stone_rep = struct
    type player = bool
    type piece =  P of int
    let int_of_piece = function P x → x
    type placement = None | M of player
    type case = Empty | Occup of player*piece
    let value_on_case = function
        Empty → 0
      | Occup (j, x) → int_of_piece x

    type game = J of case array * placement array * piece list * piece list
    type move =  int * piece
```

```
let lines = [
  [0;1]; [2;3;4]; [5; 6; 7; 8;]; [9; 10; 11; 12; 13]; [14; 15; 16; 17];
  [0; 2; 5; 9]; [1; 3; 6; 10; 14]; [4; 7; 11; 15]; [8; 12; 16]; [13; 17];
  [9; 14]; [5; 10; 15]; [2; 6; 11; 16]; [0; 3; 7; 12; 17]; [1; 4; 8; 13]  ]

let vector_l = Array.of_list lines

let lines_per_case v =
  let t =  Array.length v  in
  let r = Array.create 18 [||] in
  for i = 0 to 17  do
    let w = Array.create 3 0
    and p = ref 0 in
    for j=0 to t-1 do if List.mem i v.(j) then (w.(!p) <- j; incr p)
    done;
    r.(i) <- w
  done;
  r

let num_line_per_case = lines_per_case vector_l
let rec lines_of_i i l = List.filter (fun t → List.mem i t) l

let lines_of_cases l =
  let a = Array.create 18 l in
  for i=0 to 17 do
    a.(i) <-  (lines_of_i i l)
  done; a

let ldc = lines_of_cases lines

let game_start ()= let lp = [6; 5; 4; 3; 3; 2; 2; 1; 1] in
J ( Array.create 18 Empty, Array.create 15 None,
    List.map (fun x → P x) lp, List.map (fun x → P x) lp )

let rec unicity l = match l with
   [] → []
| h::t → if List.mem h t then unicity t else h:: (unicity t)

let legal_moves player (J (ca, m, r1, r2)) =
  let r = if player then r1 else r2 in
  if r = [] then []
  else
    let l = ref [] in
    for i = 0 to 17 do
      if value_on_case ca.(i) = 0 then l:= i:: !l
    done;
    let l2 = List.map (fun x→
      List.map (fun y→ x,y) (List.rev(unicity r)) ) !l in
    List.flatten l2
let copy_board p = Array.copy p
```

```
let  carn_copy m = Array.copy m
let rec play_piece stone l = match l with
   []  →  []
| x :: q → if x=stone  then q
else x :: (play_piece stone q)

let count_case player case = match case with
  Empty → 0
| Occup (j,p) → if j = player then (int_of_piece p) else  0

let count_line player line pos =
  List.fold_left (fun x y → x +  count_case player pos.(y)) 0 line

let rec count_max n = function
     [] → 0
  | t :: q →
      if (n>0) then
        (int_of_piece t) + count_max (n-1) q
      else
        0

let rec nbr_cases_free ca  l  = match l with
   []    → 0
| t :: q → let c = ca.(t) in
  match c with
    Empty → 1 + nbr_cases_free ca q
  | _     → nbr_cases_free ca q

let a_placement i ma =
  match ma.(i) with
    None → false
  | _ → true

let which_placement i ma =
  match ma.(i) with
    None → failwith "which_placement"
  | M j  → j

let is_filled l ca =  nbr_cases_free ca l = 0

(* function play : arbitrates the game *)
let play player move game =
  let (c, i) = move in
  let J (p, m, r1, r2) = game in
  let nr1,nr2 = if player then play_piece i r1,r2
  else r1,  play_piece i r2  in
  let np = copy_board p in
  let nm = carn_copy m in
  np.(c)<-Occup(player,i);  (* on play le move *)
  let lines_of_the_case = num_line_per_case.(c) in
```

```
      (* calculation of the placements of the three lines *)
      for k=0 to 2 do
        let l =  lines_of_the_case.(k) in
        if not (a_placement l nm) then (
          if is_filled vector_l.(l) np then (
            let c1 = count_line player vector_l.(l) np
            and c2 = count_line (not player) vector_l.(l) np in
            if (c1 > c2) then nm.(l) <- M player
            else  ( if c2 > c1 then nm.(l) <- M (not player)
            else nm.(l) <- M (not player ))))
      done;

      (* calculation of other placements *)
      for k=0 to 14 do
        if not (a_placement k nm ) then
          if is_filled vector_l.(k) np then failwith "player"
          else
            let c1 = count_line player  vector_l.(k) np
            and c2 = count_line (not player) vector_l.(k) np in
            let cases_free = nbr_cases_free np vector_l.(k) in
            let max1  = count_max cases_free
                (if player then nr1 else nr2)
            and max2 = count_max cases_free
                (if player then nr2 else nr1) in
            if c1 >= c2 + max2 then nm.(k) <- M player
            else if c2 >= c1 + max1 then nm.(k) <- M (not player)
      done;
      J(np,nm,nr1,nr2)
   end ;;
module Stone_rep :
  sig
    type player = bool
    and piece = P of int
    val int_of_piece : piece -> int
    type placement = None | M of player
    and case = Empty | Occup of player * piece
    val value_on_case : case -> int
    type game = J of case array * placement array * piece list * piece list
    and move = int * piece
    val lines : int list list
    val vector_l : int list array
    val lines_per_case : int list array -> int array array
    val num_line_per_case : int array array
    val lines_of_i : 'a -> 'a list list -> 'a list list
    val lines_of_cases : int list list -> int list list array
    val ldc : int list list array
    val game_start : unit -> game
    val unicity : 'a list -> 'a list
    val legal_moves : bool -> game -> (int * piece) list
    val copy_board : 'a array -> 'a array
    val carn_copy : 'a array -> 'a array
    val play_piece : 'a -> 'a list -> 'a list
```

```
    val count_case : player -> case -> int
    val count_line : player -> int list -> case array -> int
    val count_max : int -> piece list -> int
    val nbr_cases_free : case array -> int list -> int
    val a_placement : int -> placement array -> bool
    val which_placement : int -> placement array -> player
    val is_filled : int list -> case array -> bool
    val play : player -> int * piece -> game -> game
  end
```

関数 play は、三つのステージに分割されます:

1. ゲームの局面をコピーし、この局面で一手指す。
2. 現在考慮中の三つの行のうちの一行のコマの置場所の決定。
3. 他の ley-lines の扱い。

第二ステージでは、指し手の局面を通して三つの行のいずれもまだ取られていないことを検証し、そして、それを取ることができるどうかをチェックします。後者の場合、プレーヤーそれぞれについてスコアを計算し、どちらが厳密に高い点を持っているかを決定し、その行を高い点を取ったプレーヤのものとします。スコアが等しい場合、その行は最近のプレーヤーの敵のものとなります。基本的には、ちょうど一つの場合だけの行はありません。行は少なくとも二つのコマによって埋められることになります。よって、もし今、手を指したプレーヤが相手のスコアと同じなった場合には、その行を取ることは期待できず、相手の手にわたることになります。もし行がまだ埋まっていなければ、「第三ステージ」で解析されることになります。

第三ステージでは、まだいずれのプレーヤのものにもなっていない、それぞれの行について、その行が埋まっているかどうかを調べ、その行が相手によって取られることがないかどうかをチェックします。この場合には、この行はすぐに相手のものになります。このテストを行うためには、その行のあるプレーヤの潜在的に取得可能な（つまり、最も良いコマを使って取得可能な）合計スコアの最大値を計算する必要があります。もし、この行がいまだ係争中であれば、これ以上何も行われません。

評価。 ゲームの初盤では大量の局面を評価する必要があるため、評価関数は簡単である必要があります。基本的な考えは、すぐに自分が強いコマを置いてしまって、後で相手が強いコマを使えるようになってしまうほど、簡単にはしすぎないということです。

二つの基準を使います。一つは取った行数で、もう一つは残りのコマの価値を計算することによって得られる将来の指し手の見積もりです。プレーヤー 1 について次の式を使います。

$$score \ = \ 50 * (c_1 - c_2) + 10 * (pr_1 - pr_2)$$

ここで c_i は、取った行の数、pr_i はプレーヤ i に残っているコマの合計です。

この式は、取った行（$c_1 - c_2$）と、見積もり（$pr_1 - pr_2$）の差がプレーヤー 1 に有利になるとき、正の結果を返します。よって、コマ 6 を置くのは、少なくとも 2 行を取れる

のでなければ、適切とは言えません。一行を取ると評価値 50 を得ますが、「6」のコマを使うと 10 × 6 ポイントのコストがかかるので、同じスコアとなる、つまり 10 ポイントのロスとなる、コマ「1」を指すのが好ましいでしょう。

```
# module Stone_eval = struct
    open Stone_rep
    type game = Stone_rep.game

    exception Done of bool
    let moreI = 1000 and  lessI = -1000

    let nbr_lines_won  (J(ca, m,r1,r2))  =
      let c1,c2 = ref 0,ref 0 in
      for i=0 to 14 do
        if a_placement i m then if which_placement i m then incr c1 else incr c2
      done;
      !c1,!c2

    let rec nbr_points_remaining lig = match lig with
       [] → 0
    | t :: q → (int_of_piece t) + nbr_points_remaining q

    let evaluate player game  =
      let (J (ca,ma,r1,r2)) = game in
      let c1,c2  = nbr_lines_won  game in
      let pr1,pr2 = nbr_points_remaining r1, nbr_points_remaining r2 in
      match player with
        true → if c1 > 7 then moreI else 50 * (c1 - c2) + 10 * (pr1 - pr2)
      | false → if c2 > 7 then lessI else 50 * (c1 - c2) + 10 * (pr1 - pr2)

    let is_leaf player game =
      let v = evaluate player game  in
      v = moreI or v = lessI or legal_moves player game  = []

    let is_stable player game = true

    type state = G | P | N | C
    let state_of player m =
      let v = evaluate player m in
      if v = moreI then if player then G else P
      else
        if v = lessI
        then if player then P else G
        else
          if legal_moves player  m = [] then  N else C
  end;;
module Stone_eval :
  sig
    type game = Stone_rep.game
    exception Done of bool
    val moreI : int
    val lessI : int
```

```
    val nbr_lines_won : Stone_rep.game -> int * int
    val nbr_points_remaining : Stone_rep.piece list -> int
    val evaluate : bool -> Stone_rep.game -> int
    val is_leaf : bool -> Stone_rep.game -> bool
    val is_stable : 'a -> 'b -> bool
    type state = G | P | N | C
    val state_of : bool -> Stone_rep.game -> state
  end

# module Stone_graph = struct
    open Stone_rep
    type piece = Stone_rep.piece
    type placement = Stone_rep.placement
    type case = Stone_rep.case
    type game = Stone_rep.game
    type move = Stone_rep.move

   (* brightness for a piece *)
    let brightness = 20

   (* the colors  *)
    let cBlack  = Graphics.black
    let cRed    = Graphics.rgb 165 43 24
    let cYellow = Graphics.yellow
    let cGreen  = Graphics.rgb 31 155 33 (*Graphics.green*)
    let cWhite  = Graphics.white
    let cGray   = Graphics.rgb 128 128 128
    let cBlue   = Graphics.rgb 196 139 25 (*Graphics.blue*)

   (* width and height *)
    let width = 600
    let height = 500
   (* the border at the top of the screen from which drawing begins *)
    let top_offset = 30

   (* height of foundaries *)
    let bounds = 5

   (* the size of the border on the left side of the virtual table *)
    let virtual_table_xoffset = 145

   (* left shift for the black pieces *)
    let choice_black_offset = 40

   (* left shift for the red pieces *)
    let choice_red_offset = 560

   (* height of a case for the virtual table *)
    let virtual_case_size = 60

   (* corresp : int*int -> int*int  *)
   (* establishes a correspondence between a location in the matrix *)
```

```
(* and a position on the virtual table servant for drawing *)
  let corresp cp =
    match cp with
      0  → (4,1)
    | 1 → (6,1)
    | 2 → (3,2)
    | 3 → (5,2)
    | 4 → (7,2)
    | 5 → (2,3)
    | 6 → (4,3)
    | 7 → (6,3)
    | 8 → (8,3)
    | 9 → (1,4)
    | 10 → (3,4)
    | 11 → (5,4)
    | 12 → (7,4)
    | 13 → (9,4)
    | 14 → (2,5)
    | 15 → (4,5)
    | 16 → (6,5)
    | 17 → (8,5)
    | _ → (0,0)

  let corresp2 ((x,y) as cp) =
    match cp with
      (0,0) → 0
    |  (0,1) → 1
    |  (1,0) → 2
    |  (1,1) → 3
    |  (1,2) → 4
    |  (2,0) → 5
    |  (2,1) → 6
    |  (2,2) → 7
    |  (2,3) → 8
    |  (3,0) → 9
    |  (3,1) → 10
    |  (3,2) → 11
    |  (3,3) → 12
    |  (3,4) → 13
    |  (4,0) → 14
    |  (4,1) → 15
    |  (4,2) → 16
    |  (4,3) → 17
    |  (x,y) → print_string "Err ";
        print_int x;print_string " ";
        print_int y; print_newline() ; 0

  let col = 5
  let lig = 5

(* draw_background : unit -> unit  *)
```

```
(* draw the screen background  *)
  let draw_background () =
    Graphics.clear_graph() ;
    Graphics.set_color cBlue ;
    Graphics.fill_rect bounds bounds width (height-top_offset)

(* draw_places : unit -> unit  *)
(* draw the pieces at the start of the game *)
  let draw_places () =
    for l = 0 to 17 do
      let cp = corresp l in
      if cp <> (0,0) then
        begin
          Graphics.set_color cBlack ;
          Graphics.draw_circle
            ((fst cp)*30 + virtual_table_xoffset)
            (height - ((snd cp)*55 + 25)-50) (brightness+1) ;
          Graphics.set_color cGray ;
          Graphics.fill_circle
            ((fst cp)*30 + virtual_table_xoffset)
            (height - ((snd cp)*55 + 25)-50) brightness
        end

    done

(* draw_force_lines : unit -> unit  *)
(* draws ley-lines  *)
  let draw_force_lines () =
    Graphics.set_color cYellow ;
    let lst = [((2,1),(6,1)); ((1,2),(7,2)); ((0,3),(8,3));
               ((-1,4),(9,4)); ((0,5),(8,5)); ((5,0),(1,4));
               ((7,0),(2,5)); ((8,1),(4,5)); ((9,2),(6,5));
               ((10,3),(8, 5)); ((3,6),(1,4)); ((5,6),(2,3));
               ((7,6),(3,2)); ((9,6),(4,1)); ((10,5),(6,1))] in
    let rec lines l =
      match l with
         []  → ()
      | h :: t → let deb = fst h and complete = snd h in
        Graphics.moveto
          ((fst deb) * 30 + virtual_table_xoffset)
          (height -     ((snd deb) * 55 + 25) -50) ;
        Graphics.lineto
          ((fst complete) * 30 + virtual_table_xoffset)
          (height -     ((snd complete) * 55 + 25) -50) ;
        lines t
    in lines lst

(* draw_final_places : unit -> unit  *)
(* draws final cases for each ley-line *)
(* coordinates represent in the virtual array
 used for positioning *)
```

```
let draw_final_places () =
  let lst = [(2,1); (1,2); (0,3); (-1,4); (0,5); (3,6); (5,6);
             (7,6); (9,6); (10,5); (10,3); (9,2); (8,1); (7,0);
             (5,0)] in
  let rec final l =
    match l with
      [] → ()
    | h :: t →  Graphics.set_color cBlack ;
        Graphics.draw_circle
          ((fst h)*30 + virtual_table_xoffset)
          (height - ((snd h)*55 + 25)-50) (brightness+1) ;
        Graphics.set_color cGreen ;
        Graphics.fill_circle
          ((fst h)*30 + virtual_table_xoffset)
          (height - ((snd h)*55 + 25)-50) brightness ;
        final t
  in final lst


(* draw_table : unit -> unit  *)
(* draws the whole game  *)
let draw_table () =
  Graphics.set_color cYellow ;
  draw_background () ;
  Graphics.set_line_width 5 ;
  draw_force_lines () ;
  Graphics.set_line_width 2 ;
  draw_places () ;
  draw_final_places ();
  Graphics.set_line_width 1

(* move -> couleur -> unit  *)
let draw_piece  player  (n_case,P cp)  = (*   (n_caOccup(c,v),cp) col =*)
  Graphics.set_color (if player then cBlack else cRed); (*col;*)
  let co = corresp n_case in
  let x = ((fst co)*30 + 145) and y = (height - ((snd co)*55 + 25)-50) in
  Graphics.fill_circle x y brightness ;
  Graphics.set_color cWhite ;
  Graphics.moveto (x - 3) (y - 3) ;
  let dummy = 5 in
  Graphics.draw_string (string_of_int cp) (*;*)
(*    print_string "---";print_int n_case; print_string " "; print_int cp ;print_newline() *)

(* conv : Graphics.status -> int  *)
(* convert a mouse click into a position on a virtual table permitting *)
(* its drawing *)
let conv st =
  let xx = st.Graphics.mouse_x and yy = st.Graphics.mouse_y in
  let y = (yy+10)/virtual_case_size - 6 in
  let dec =
```

```
      if y = ((y/2)*2) then 60 else 40 in
    let offset = match (-1*y) with
      0 → -2
    | 1 → -1
    | 2 → -1
    | 3 → 0
    | 4 → -1
    | _ → 12 in
    let x = (xx+dec)/virtual_case_size - 3 + offset in
    (-1*y, x)

  (* line_number_to_aff : int -> int*int *)
  (* convert a line number into a polition on the virtual table serving *)
  (* for drawing *)
  (* the coordinate returned corresponds to the final case for the line *)
  let line_number_to_aff n =
    match n with
      0 → (2,1)
    | 1 → (1,2)
    | 2 → (0,3)
    | 3 → (-1,4)
    | 4 → (0,5)
    | 5 → (5,0)
    | 6 → (7,0)
    | 7 → (8,1)
    | 8 → (9,2)
    | 9 → (10,3)
    | 10 → (3,6)
    | 11 → (5,6)
    | 12 → (7,6)
    | 13 → (9,6)
    | 14 → (10,5)
    | _ → failwith "line" (*raise Rep.Out_of_bounds*)

  (* draw_lines_won : game -> unit  *)
  (* position a marker indicating the player which has taken the line *)
  (* this is done for all lines *)
  let  drawb  l  i =
    match l with
      None → failwith "draw"
    | M j  → let pos = line_number_to_aff i in
  (*                print_string "''''";
   print_int i;
   print_string "---";
   Printf.printf "%d,%d\n" (fst pos) (snd pos);
*)                  Graphics.set_color (if j then cBlack else cRed);
      Graphics.fill_rect ((fst pos)*30 + virtual_table_xoffset-bounds)
        (height - ((snd pos)*55 + 25)-60) 20 40

  let draw_lines_won om nm  =
    for i=0 to 14 do
      if om.(i) <> nm.(i) then drawb nm.(i) i
```

```
  done
(********************
let black_lines = Rep.lines_won_by_player mat Rep.Noir and
red_lines = Rep.lines_won_by_player mat Rep.Rouge
in
print_string "black : "; print_int (Rep.list_size black_lines);
print_newline () ;
print_string "red : "; print_int (Rep.list_size red_lines);
print_newline() ;

let rec draw l col =
match l with
[] -> ()
| h::t -> let pos = line_number_to_aff h in
Graphics.set_color col ;
Graphics.fill_rect ((fst pos)*30 + virtual_table_xoffset-bounds)
(height - ((snd pos)*55 + 25)-60) 20 40 ;
draw t col
in draw black_lines cBlack ;
draw red_lines cRed
**************************************************)

(* draw_poss : item list -> int -> unit  *)
(* draw the pieces available for a player based on a list *)
(* the parameter "off" indicates the position at which to place the list *)
let draw_poss player lst off =
  let c = ref (1) in
  let rec draw l =
    match l with
      []  →  ()
    | v :: t → if player  then Graphics.set_color cBlack
    else Graphics.set_color cRed;
        let x = off   and
            y = 0+(!c)*50   in
        Graphics.fill_circle x y brightness ;
        Graphics.set_color cWhite ;
        Graphics.moveto (x - 3) (y - 3) ;
        Graphics.draw_string (string_of_int v) ;
        c := !c + 1 ;
        draw t

  in draw (List.map (function P x → x) lst)

(* draw_choice : game -> unit *)
(* draw the list of pieces still available for each player *)
let draw_choice (J (ca,ma,r1,r2)) =
  Graphics.set_color cBlue ;
  Graphics.fill_rect (choice_black_offset-30) 10 60
    (height - (top_offset + bounds)) ;
  Graphics.fill_rect (choice_red_offset-30) 10 60
    (height - (top_offset + bounds)) ;
  draw_poss true r1 choice_black_offset ;
```

```
  draw_poss false r2 choice_red_offset

(* wait_click : unit -> unit *)
(* wait for a mouse click *)
let wait_click () = Graphics.wait_next_event [Graphics.Button_down]


(* item list -> item  *)
(* return, for play, the piece chosen by the user *)
let select_pion player  lst =
  let ok = ref false and
      choice = ref 99 and
      pion = ref (P(-1))
  in
  while not !ok do
    let st = wait_click () in
    let size = List.length lst in
    let x = st.Graphics.mouse_x and y = st.Graphics.mouse_y in
    choice := (y+25)/50 - 1 ;
    if !choice <= size && ( (player && x < 65 )
                       || ( (not player) && (x > 535))) then ok := true
    else ok := false ;
    if !ok then
      try
        pion := (List.nth  lst !choice) ;
        Graphics.set_color cGreen ;
        Graphics.set_line_width 2 ;
        Graphics.draw_circle
          (if player then choice_black_offset else choice_red_offset)
          ((!choice+1)*50) (brightness + 1)
      with _  → ok := false ;
  done ;
  !pion

(* choiceH : game -> move  *)
(* return a move for the human player.
 return the choice of the number, the case, and the piece *)
let rec choice player game = match game with (J(ca,ma,r1,r2)) →
  let choice = ref (P(-1))
  and c = ref (-1, P(-1)) in
  let lcl = legal_moves player game in
  while not (List.mem !c lcl) do
    print_newline();print_string "CHOICE";
    List.iter (fun (c,P p) → print_string "["; print_int c;print_string " ";
      print_int p;print_string "]")
      (legal_moves player game);
    draw_choice game;
    choice :=  select_pion player  (if player then r1 else r2) ;
(*       print_string "choice "; print_piece !choice;*)
    c :=  (corresp2  (conv (wait_click())), !choice)
(*       let (x,y) = !c in
```

```
   (print_string "...";print_int x; print_string " "; print_piece y;
   print_string " -> ";
   print_string "END_CHOICE";print_newline())
*)     done ;
     !c (* case, piece *)


  (* home : unit -> unit  *)
  (* place a message about the game *)
  let home () =
    Graphics.open_graph
      (" " ^ (string_of_int (width + 10)) ^ "x" ^ (string_of_int (height + 10))
       ^ "+50+50") ;
    Graphics.moveto (height / 2) (width / 2) ;
    Graphics.set_color cBlue ;
    Graphics.draw_string "Stonehenge" ;
    Graphics.set_color cBlack ;
    Graphics.moveto 2 2 ;
    Graphics.draw_string "Mixte Projets Maîtrise & DESS GLA" ;
    wait_click () ;
    Graphics.clear_graph ()

  (* exit : unit -> unit *)
  (* close everything ! *)
  let exit () =
    Graphics.close_graph ()

  (* draw_button : int -> int -> int -> int -> string -> unit  *)
  (* draw a button with a message *)
  let draw_button x y w h s =
    Graphics.set_line_width 1 ;
    Graphics.set_color cBlack ;
    Graphics.moveto x y ;
    Graphics.lineto x (y+h) ;
    Graphics.lineto (x+w) (y+h) ;
    Graphics.lineto (x+w) y ;
    Graphics.lineto x y ;
    Graphics.moveto (x+bounds) (height - (top_offset/2)) ;
    Graphics.draw_string s

  (* draw_message : string -> unit  *)
  (* position a message *)
  let draw_message s =
    Graphics.set_color cBlack;
    Graphics.moveto 3 (height - (top_offset/2)) ;
    Graphics.draw_string s

  (* erase_message : unit -> unit  *)
  (* as the name indicates *)
  let erase_message () =
    Graphics.set_color Graphics.white;
```

```
  Graphics.fill_rect 0 (height-top_offset+bounds) width top_offset

(* question : string -> bool  *)
(* pose the user a question, and wait for a yes/no response *)
let question s =
  let xb1 = (width/2) and xb2 = (width/2 + 30) and wb = 25 and hb = 16
  and yb = height - 20  in
  let rec attente () =
    let e = wait_click () in
    if (e.Graphics.mouse_y < (yb+hb)) & (e.Graphics.mouse_y > yb) then
      if (e.Graphics.mouse_x > xb1) & (e.Graphics.mouse_x < (xb1+wb)) then
        true
      else
        if (e.Graphics.mouse_x > xb2) & (e.Graphics.mouse_x < (xb2+wb)) then
          false
        else
          attente()
    else
      attente () in
  draw_message s;
  draw_button xb1 yb wb hb "yes";
  draw_button xb2 yb wb hb "no";
  attente()

(* q_begin : unit -> bool *)
(* Ask if the player wishes to be the first player or not *)
let q_begin () =
  let b = question "Would you like to play first ?" in
  erase_message();
  b

(* q_continue : unit -> bool *)
(* Ask if the user wishes to play the game again *)
let q_continue () =
  let b = question "Play again ?" in
  erase_message();
  b
(* won : unit -> unit  *)
(* a message indicating the machine has won *)
let won () = draw_message "I won :-)"; wait_click(); erase_message()

(* lost : unit -> unit  *)
(* a message indicating the machine has lost *)
let lost () = draw_message "You won :-("; wait_click(); erase_message()

(* nil : unit -> unit  *)
(* a message indicating stalemate *)
let nil () = draw_message "Stalemate"; wait_click(); erase_message()

(* init : unit -> unit *)
(* draw the initial game board *)
```

```
  let init () = let game = game_start () in
  draw_table () ;
  draw_choice game

  (* drawH : move -> game -> unit *)
  (* draw a piece for the human player *)
  (* let drawH cp j =  draw_piece cp cBlack ;
   draw_lines_won j
*)
  (* drawM : move -> game -> unit *)
  (* draw a piece for the machine player *)
  (* let drawM cp j =  draw_piece cp cRed ;
   draw_lines_won j
*)
  let print_placement m = match m with
    None → print_string "None "
  | M j  → print_string ("Pl "^(if j then "1 " else "2 "))

  let position player move
      (J(ca1,m1,r11,r12))
      (J(ca2,m2,r21,r22) as new_game) =
    draw_piece player move;
    draw_choice new_game;
  (*  print_string "_______OLD___________________\n";
   Array.iter print_placement m1; print_newline();
   List.iter print_piece r11; print_newline();
   List.iter print_piece r12; print_newline();
   print_string "_______NEW__________________\n";
   Array.iter print_placement m2; print_newline();
   List.iter print_piece r21; print_newline();
   List.iter print_piece r22; print_newline();
*)  draw_lines_won m1 m2

  (*
   if player then draw_piece move cBlack
   else  draw_piece move cRed
*)
  let q_player () =
    let b = question "Is there a machine playing?" in
    erase_message ();
    b
  end;;
Characters 11114-11127:
Warning: this expression should have type unit.
Characters 13197-13209:
Warning: this expression should have type unit.
Characters 13345-13357:
Warning: this expression should have type unit.
Characters 13478-13490:
Warning: this expression should have type unit.
module Stone_graph :
  sig
```

```
type piece = Stone_rep.piece
and placement = Stone_rep.placement
and case = Stone_rep.case
and game = Stone_rep.game
and move = Stone_rep.move
val brightness : int
val cBlack : Graphics.color
val cRed : Graphics.color
val cYellow : Graphics.color
val cGreen : Graphics.color
val cWhite : Graphics.color
val cGray : Graphics.color
val cBlue : Graphics.color
val width : int
val height : int
val top_offset : int
val bounds : int
val virtual_table_xoffset : int
val choice_black_offset : int
val choice_red_offset : int
val virtual_case_size : int
val corresp : int -> int * int
val corresp2 : int * int -> int
val col : int
val lig : int
val draw_background : unit -> unit
val draw_places : unit -> unit
val draw_force_lines : unit -> unit
val draw_final_places : unit -> unit
val draw_table : unit -> unit
val draw_piece : bool -> int * Stone_rep.piece -> unit
val conv : Graphics.status -> int * int
val line_number_to_aff : int -> int * int
val drawb : Stone_rep.placement -> int -> unit
val draw_lines_won :
  Stone_rep.placement array -> Stone_rep.placement array -> unit
val draw_poss : bool -> Stone_rep.piece list -> int -> unit
val draw_choice : Stone_rep.game -> unit
val wait_click : unit -> Graphics.status
val select_pion : bool -> Stone_rep.piece list -> Stone_rep.piece
val choice : bool -> Stone_rep.game -> int * Stone_rep.piece
val home : unit -> unit
val exit : unit -> unit
val draw_button : int -> int -> int -> int -> string -> unit
val draw_message : string -> unit
val erase_message : unit -> unit
val question : string -> bool
val q_begin : unit -> bool
val q_continue : unit -> bool
val won : unit -> unit
val lost : unit -> unit
val nil : unit -> unit
```

```
    val init : unit -> unit
    val print_placement : Stone_rep.placement -> unit
    val position :
      bool ->
      int * Stone_rep.piece -> Stone_rep.game -> Stone_rep.game -> unit
    val q_player : unit -> bool
  end
```

組み立て。 シグネチャDISPLAY とコンパチブルなグラフィカルインターフェースを書く、モジュール Stone_graph を記述します。Stonehenge ゲームのための適切な引数を渡して、パラメタ付きモジュール FSkeleton を適用することで、C4_skeletonG と似た Stone_skeletonG を作ります。

```
# module Stone_skeletonG = FSkeleton (Stone_rep)
                                      (Stone_graph)
                                      (Stone_eval)
                                      (FAlphabeta (Stone_rep) (Stone_eval)) ;;
module Stone_skeletonG :
  sig
    val depth : int ref
    exception Won
    exception Lost
    exception Nil
    val won : unit -> unit
    val lost : unit -> unit
    val nil : unit -> unit
    val again : unit -> bool
    val play_game : Stone_graph.game ref
    val exit : unit -> unit
    val home : unit -> unit
    val playH : bool -> unit -> unit
    val playM : bool -> unit -> unit
    val init : unit -> (unit -> unit) * (unit -> unit)
  end
```

主なモジュール Stone_mainG を生成することができます。

```
# module Stone_mainG = FMain(Stone_skeletonG) ;;
module Stone_mainG :
  sig
    val play_game : (unit -> 'a) * (unit -> 'b) -> unit
    val main : unit -> unit
  end
```

Stone_mainG.main () を起動すると、図 17.6 のように窓が開きます。誰がプレーするのかを表示するダイアローグを表示した後で、ゲームが始まります。人間のプレーヤーはコマを選択してそれを置きます。

さらに学びたい人のために

これらのアプリケーションの構成では、本節で示した二つのゲームのために FAlphabeta と FSkeleton の直接的な再利用が可能になる、複数のパラメタ付きモジュールを用いています。Stonehenge ゲームでは、play のために必要となる、REPRESENTATION には現れない、Stone_rep からの関数がいくつか、評価関数で使われています。これが、モジュール Stone_rep が REPRESENTATION によって閉じられていない理由です。この、ゲームの特定の側面のためのモジュールの分割によって、（図 17.4 で示した）ゲームの枠組みの依存性を脆くすることなく、逐次的開発が可能になります。

最初の強化点としては、局面と指し手が与えられたとき、それに先立つ局面の決定が簡単になるようなゲームが含まれます。このような場合には、関数 play のためにゲームのコピーをわざわざ作るのではなく、バックトラックが可能になるような指し手の履歴を保存するほうが、より効率的でしょう。Stonehenge ゲームではなく、Connect 4 がこのケースです。

第二の強化点は、もう一方のプレーヤーが次の指し手を選択している間に、未来の局面を評価することによって、プレーヤーの返手の時間を利用することです。このためには、スレッド（19 章参照）を使うことができ、これにより、並行計算が可能になります。もし、プレーヤの返手がすでに探索したものであれば、新しい局面から再開しなければいけない場合以外は、時間のメリットは直接的です。

第三の強化点は、序盤の指し手の辞書を作成し、探索することです。Stonehenge ゲームでこれを行うこともできますが、序盤における可能な指し手の探索が極めて大きく複雑な他の多くのゲームでも有効です。最初の局面からの「最善」手の見積もりと事前計算とその結果をある種のデータベースに保持することで、多くのものが得られます。偶然の要素を導入することで、似たような、あるいは同等の価値の指し手の集合からランダムに指し手を選択することで、少量の「スパイス」（と、おそらく予測不可能性）をゲームに加えることもできるでしょう。

第四の見方は、探索の深さを固定値に限定するのではなく、計算によってかかる時間の上限値を制限するようにすることです。このようにすれば、残りの指してが限定されてきたときに、プログラムが効率的により深いところまで探索することができます。この変更は、探索する深さを増やして再探索できるように minmax を変更する必要があります。

minmax によってパラメタ化された、ゲームに依存する発見的方法は、探索時にどの枝を刈り込んで、どの枝を捨てることができるかを、選択することができるでしょう。

また、更なる実装や再実装が必要になるような他のゲームもたくさんあります。多くの古典的ゲーム、チェッカー、オセロ、アバロン... などが例証となるでしょう。しかし、にもかかわらず、また多くのあまり知られていないゲームが、容易にコンピュータによってプレー可能です。チェッカーや Nuba ゲームなどの学生プロジェクトが Web 上で見つけられます。

リンク: http://www.gamecabinet.com/rules/Nuba.html

カードゲームやサイコロゲームのような、確率的性質を伴うゲームではミニマックス-$\alpha\beta$ アルゴリズムを指し手の選択の確率を考慮に入れるように変更する必要があります。

ゲームのインターフェースについては、21 章で、コストをかけずに最後の指し手をやり直せるような機能を持つ、web ベースのインターフェースを構築する際に、再び取り上げます。これにより、ゲームとその相互作用について記述している一つの要素を変更するだけで、二人プレーヤ用ゲームをサポートする機能を拡張することができる、モジュール化された構成の利点を享受できます。

ファンシー・ロボット

本節では、グラフィクスライブラリからオブジェクトの使用例を示します。単純継承、メソッドオーバーライド、動的ディスパッチの概念を再び取り上げます。また、パラメタ付きクラスがどのように便利に使用されるかも、みてみることにします。

このアプリケーションにはオブジェクトについて二つの主なカテゴリがあります。仮想世界とロボットです。仮想世界はロボットが進化する状態空間を表現しています。いろいろな種類のロボットのクラスを作りますが、それぞれのクラスはそれぞれの戦略で仮想世界の中を動きまわります。

ここでロボットと仮想世界の間の主なやり取りは非常に簡単です。仮想世界は完全にゲームの制御の元にあります。仮想世界は、一ステップごとに、それぞれのロボットに次の移動一を知っているかどうかを尋ねます。ロボットはそれぞれ次の移動位置を公正かつ盲目的に決定します。ロボットは、仮想世界の座標空間についても、他のロボットの存在にていても、一切知りません。もし、ロボットが要求した位置が移動可能なら、仮想世界はそのロボットその位置へ移動します。

仮想世界は、インターフェースを通してロボットの進化を表示します。この例の計画と開発の（相対的な）複雑さは、振る舞い（ここではロボットの進化）とそのインターフェース（ここでは、進化の追跡）の間に常に必要になる分離です。

一般的記述　このアプリケーションは 2 段階で開発します。

1. 仮想世界と、考察されるロボットの多様な集合のための純粋な計算を行うクラスを提供する定義の集まり。
2. 前述の集まりを用いて、インターフェースで追加が必要となるものの定義の集まり。このようなインターフェースの例として二つをあげます。一つは、初歩的なテキストベースのインターフェースで、もう一つはグラフィックライブラリを用いた、より手の込んだものです。

最初の節では、 ロボットの抽象的定義を与えます。そして（557 ページで）、仮想世界の純粋な抽象的定義を与えます。次の節（558 ページ）では、ロボットのためのテキストベースのインターフェースを与え、四番目の節（560 ページ）で仮想世界のインターフェースを与えます。562 ページでは、ロボットのためのグラフィカルインターフェースを導入し、最後に（565 ページで）グラフィカルインターフェースのための仮想世界を定義します。

「抽象」ロボット

最初にすることは、ロボットが移動する環境、つまり、ロボットを表示するインターフェースに関するいかなる考察とも独立に、抽象的にロボットを調査することです。

```
# class virtual robot (i0:int) (j0:int) =
    object
      val mutable i = i0
      val mutable j = j0
      method get_pos = (i,j)
      method set_pos (i', j') = i <- i'; j <- j'
      method virtual next_pos : unit → (int * int)
    end ;;
```

ロボットは、その位置（i と j）を知っている、あるいは知っていると信じており、その位置を要求するものには伝えることができ（get_pos）、正確にどこにいるべきかを知っている場合にはこの位置に関する知識を変更でき（set_pos）、新しい位置に向かって移動することを決定できる（next_pos）ような存在です。

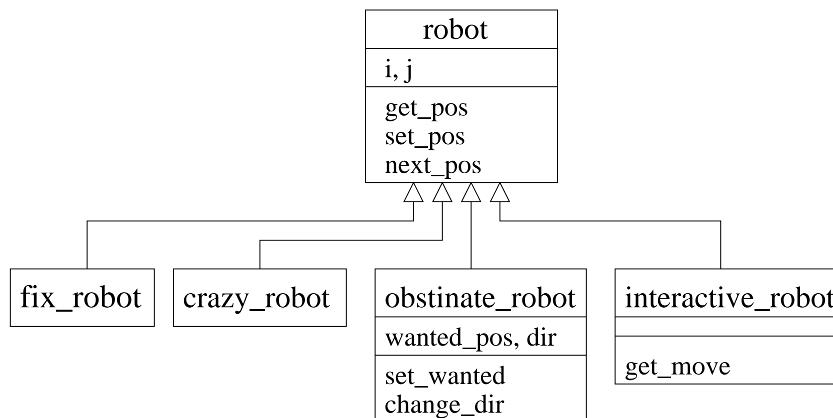


図 17.9: 純粋ロボットのクラス階層

プログラムの可読性を改善するために、絶対方位に基づく相対的移動を定義します。

```
# type dir = North | East | South | West | Nothing  ;;

# let walk (x,y) = function
      North → (x,y+1) | South → (x,y-1)
    | West  → (x-1,y) | East  → (x+1,y)
    | Nothing → (x,y) ;;
val walk : int * int -> dir -> int * int = <fun>

# let turn_right = function
```

```
    North → East | East → South | South → West | West → North | x → x ;;
val turn_right : dir -> dir = <fun>
```

この図式は、本節でロボットの動き型をより正確にみるために定義する四つの異なるロボットの種 (図 17.9 参照) が基底クラスとする仮想クラス robots によって示されます。

- 決して動かない固定ロボット:

```
# class fix_robot i0 j0 =
    object
      inherit robot i0 j0
      method next_pos() = (i,j)
    end ;;
```

- ランダムに動く気違いロボット:

```
# class crazy_robot i0 j0 =
    object
      inherit robot i0 j0
      method next_pos () = ( i+(Random.int 3)-1 , j+(Random.int 3)-1 )
    end ;;
```

- 可能なときには常に一方向に進もうとする頑固ロボット:

```
# class obstinate_robot i0 j0 =
    object(self)
      inherit robot i0 j0
      val mutable wanted_pos = (i0,j0)
      val mutable dir = West

      method private set_wanted_pos d = wanted_pos <- walk (i,j) d
      method private change_dir = dir <- turn_right dir
      method next_pos () = if (i,j) = wanted_pos
        then let np = walk (i,j) dir in ( wanted_pos <- np ; np )
        else ( self#change_dir ; wanted_pos <- (i,j) ; (i,j) )
   end ;;
```

- 外部の操縦者の命令に従う対話ロボット

```
# class virtual interactive_robot i0 j0 =
    object(self)
      inherit robot i0 j0
      method virtual private get_move : unit → dir
      method next_pos () = walk (i,j) (self#get_move ())
     end ;;
```

対話ロボットの場合には、命令のやり取りができるインターフェースによって振る舞いが制御されているという点で、他のロボットと違います。これを扱うために、命令をやり取りする仮想メソッドを与えます。結果として、クラス interactive_robot は抽象クラスのままです。

四つの特殊化されたクラスだけがクラス robot から継承をするのではなく、同じ型を持つ他のクラスも継承を行います。基本的には、付加したメソッドは、これらのクラスのインスタンスの型シグネチャには現れないプライベートメソッドです（453 ページ参照）。全てのロボットが同じ型のオブジェクトだと考えたいなら、この特性は不可欠です。

純粋仮想世界

純粋仮想世界は、インターフェースとは独立した仮想世界です。これは、ロボット占めることができる位置の状態空間と理解することができます。これは、サイズが l × h の格子状の形式をとり、座標が仮想世界中で正しいかどうかを保証するメソッド is_legal、ある位置にすでにロボットがいるかどうかを示すメソッド is_free を持ちます。

実際には、仮想世界はその表面に現れる robots のリスト管理し、新しいロボットが仮想世界に入るのを許します。

最後に、仮想世界はメソッド run によって可視的にされ、仮想世界は活動することができます。

```
# class virtual ['robot_type] world (l0:int) (h0:int) =
    object(self)
      val l = l0
      val h = h0
      val mutable robots = ( [] : 'robot_type list )
      method add r = robots <- r::robots
      method is_free p = List.for_all (fun r → r#get_pos <> p)  robots
      method virtual is_legal : (int * int) → bool

      method private run_robot r =
        let p = r#next_pos ()
        in if (self#is_legal p) & (self#is_free p) then r#set_pos p

      method run () =
        while true do List.iter (function r → self#run_robot r) robots done
    end ;;
class virtual ['a] world :
  int ->
  int ->
  object
    constraint 'a =
      < get_pos : int * int; next_pos : unit -> int * int;
        set_pos : int * int -> unit; .. >
    val h : int
    val l : int
    val mutable robots : 'a list
    method add : 'a -> unit
    method is_free : int * int -> bool
    method virtual is_legal : int * int -> bool
    method run : unit -> unit
    method private run_robot : 'a -> unit
  end
```

Objective Caml の型システムでは、ロボットの型が決定されていない状態でおくことは出来ません（464 ページ参照）。この問題を解決するために、その型はクラス robot の型に縛り付けられていると考えることもできます。しかし、これでは robot と完全に同じ型を持つ仮想世界を作ることができません。結果として、その代わりに、他にも仮想世界を作りたいロボットの型で world をパラメタ化します。これによって、型パラメータを、テキストロボットとグラフィカルロボット、どちらでも実体化することができます。

文字ロボット

テキストオブジェクト　テキストインターフェースを通して制御可能なロボットを得るために、テキストオブジェクトのクラス（txt_object）を定義します。

```
# class txt_object (s0:string) =
    object
      val name = s0
       method get_name = name
    end ;;
```

インターフェースクラス: 抽象テキストロボット　robots と txt_object、双方からの継承によってテキストロボットのための抽象クラス txt_robot が得られます。

```
# class virtual txt_robot i0 j0 =
    object
      inherit robot i0 j0
      inherit txt_object "Anonymous"
   end ;;
class virtual txt_robot :
  int ->
  int ->
  object
    val mutable i : int
    val mutable j : int
    val name : string
    method get_name : string
    method get_pos : int * int
    method virtual next_pos : unit -> int * int
    method set_pos : int * int -> unit
  end
```

このクラスは、テキストインターフェース（560 ページ参照）を持つ仮想世界を定義しています。この仮想世界の住人は、txt_robot のオブジェクトでも（このクラスは抽象クラスだからです）、このクラスを継承したクラスのオブジェクトでもあり得ません。クラス txt_robot は、ある意味で、コンパイラがテキストインターフェースの仮想世界にお

ける住人のメソッドの型（計算とインターフェース）を特定するための**インターフェースクラス**です。このような仕様としてのクラスを用いることで、計算とインターフェースの間ん保持したい分離が得られます。

具体テキストロボット　具体テキストロボットは、単純に多重継承によって得られます。図 17.10 は、このクラス階層を示しています。

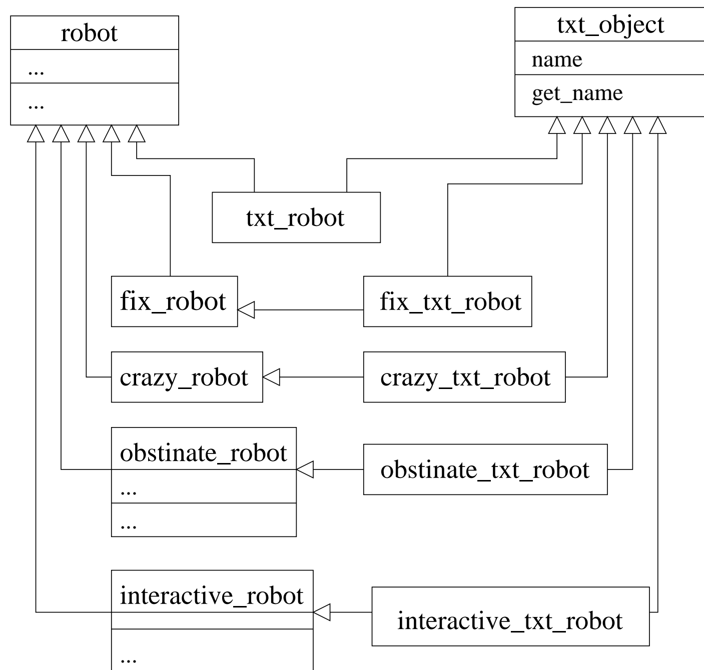


図 17.10: テキストロボットのためのクラス階層

```
# class fix_txt_robot i0 j0 = object inherit fix_robot i0 j0 inherit
  txt_object "Fix robot" end ;;

# class crazy_txt_robot i0 j0 =
    object
      inherit crazy_robot i0 j0
      inherit txt_object "Crazy robot"
    end ;;

# class obstinate_txt_robot i0 j0 =
    object
      inherit obstinate_robot i0 j0
      inherit txt_object "Obstinate robot"
    end ;;
```

対話的ロボットでは、実際に動作する実装のために、ユーザーと対話するためのメソッドを定義する必要があります。

```
# class interactive_txt_robot i0 j0 =
    object
      inherit interactive_robot i0 j0
      inherit txt_object "Interactive robot"
      method private get_move () =
        print_string "Which dir : (n)orth (e)ast (s)outh (w)est ? ";
        match read_line() with
            "n" → North  | "s" → South
          | "e" → East   | "w" → West
          |  _  → Nothing
    end ;;
```

テキストの仮想世界

テキストインターフェースの仮想世界は、次のように純粋仮想世界から導出できます:

1. 汎用クラス world からその型パラメータを txt_robot で指示されるクラスによって具体化することで継承し、
2. 異なるテキストのメソッドを取り込むためにメソッド run を再定義

```
# class virtual txt_world (l0:int) (h0:int) =
    object(self)
      inherit [txt_robot] world l0 h0 as super

      method private display_robot_pos r =
        let (i,j) = r#get_pos in Printf.printf "(%d,%d)" i j

      method private run_robot r =
        let p = r#next_pos ()
        in if (self#is_legal p) & (self#is_free p)
          then
            begin
              Printf.printf "%s is moving from " r#get_name ;
              self#display_robot_pos r ;
              print_string " to " ;
              r#set_pos p;
              self#display_robot_pos r ;
            end
          else
            begin
              Printf.printf "%s is staying at " r#get_name ;
              self#display_robot_pos r
            end ;
          print_newline () ;
```

```
        print_string"next - ";
        ignore (read_line())

    method run () =
      let print_robot r =
        Printf.printf "%s is at " r#get_name ;
        self#display_robot_pos r ;
        print_newline ()
      in
        print_string "Initial state :\n";
        List.iter print_robot robots;
        print_string "Running :\n";
        super#run()  (* 1 *)
  end ;;
```

同じメソッドの再定義の中で、上位クラスの run の呼び出し（プログラム中で (* 1 *) と印のあるメソッド呼び出し）に注意してください。これはメソッドディスパッチの二つのタイプ、静的と動的（450 ページ参照）があることを示しています。*super#run* の呼び出しは静的です。これが、スーパークラスに名前をつけている理由です。メソッドが再定義される時に、再定義されるメソッドを呼び出せる必要があります。一方、*super#run* の中で *self#run_robot* というメソッド呼び出しがあります。これは動的なディスパッチです。クラス world で定義されるメソッドではなく、クラス txt_world で定義されるメソッドが実行されます。world のメソッドが実行されたなら、何も表示されず、txt_world のメソッドは使い物になりません。

平面矩形テキスト仮想世界　は、残りの抽象メソッド is_legal を実装することで得られます。

```
# class closed_txt_world l0 h0 =
  object(self)
    inherit txt_world l0 h0
    method is_legal (i,j) = (0<=i) & (i<l) & (0<=j) & (j<h)
  end ;;
```

次のようにタイプすると実行することができます:

```
let w = new closed_txt_world 5 5
and r1 = new fix_txt_robot 3 3
and r2 = new crazy_txt_robot 2 2
and r3 = new obstinate_txt_robot 1 1
and r4 = new interactive_txt_robot 0 0
in w#add r1; w#add r2; w#add r3; w#add r4; w#run () ;;
```

しばらくの間、ロボットの仮想世界のためのグラフィカルインターフェースの実装については飛ばします。

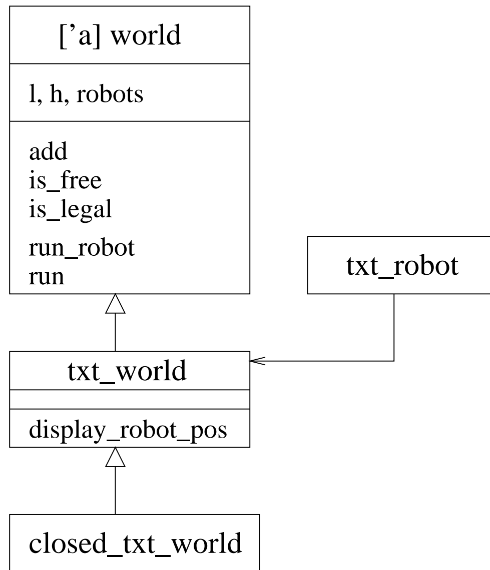


図 17.11: 平面矩形テキスト仮想世界のクラス階層

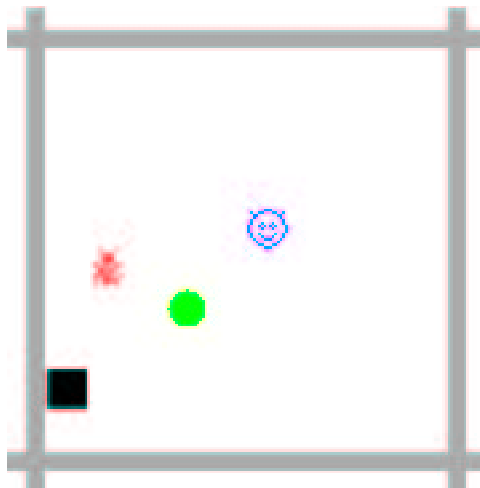


図 17.12: The graphical world of robots

グラフィカルなロボット

グラフィカルモードでのロボットは、テキストモードと同じアプローチで以下のように実装することができます。

1. 汎用のグラフィカルロボットを定義し、

2. グラフィカルロボットの抽象クラスを、ロボットとグラフィカルオブジェクトからの多重継承で定義します（558 ページのインターフェースクラスと似ています)。
3. 多重継承を通して、ロボットの特定の振る舞いを定義します。

汎用グラフィックオブジェクト

簡単なグラフィックオブジェクトは、引数としてピクセルの座標をとりそれを表示する display メソッドをを処理するオブジェクトです。

```
# class virtual graph_object =
    object
      method virtual display : int → int → unit
    end ;;
```

この仕様から、非常に複雑な振る舞いのグラフィックオブジェクトを実装することも可能です。いまのところは、オブジェクトを表現するビットマップを表示する、クラス graph_item で十分でしょう。

```
# class graph_item x y im =
    object (self)
      val size_box_x = x
      val size_box_y = y
      val bitmap = im
      val mutable last = None

      method private erase = match last with
      Some (x,y,img) → Graphics.draw_image img x y
        | None → ()

      method private draw i j = Graphics.draw_image bitmap i j
      method private keep i j =
        last <- Some (i,j,Graphics.get_image i j size_box_x size_box_y) ;

      method display i j = match last with
      Some (x,y,img) → if x<>i || y<>j
        then ( self#erase ; self#keep i j ; self#draw i j )
        | None → ( self#keep i j ; self#draw i j )
    end ;;
```

graph_item のオブジェクトは、それが描画される画像の一部を後で再描画するために保存します。加えて、画像が移動しなければ、再描画されることもありません。

```
# let foo_bitmap = [|[| Graphics.black |]|] ;;
# class square_item x col =
    object
      inherit graph_item x x (Graphics.make_image foo_bitmap)
```

```
      method private draw i j =
        Graphics.set_color col ;
        Graphics.fill_rect (i+1) (j+1) (x-2) (x-2)
    end ;;

# class disk_item r col =
    object
      inherit graph_item (2*r) (2*r) (Graphics.make_image foo_bitmap)
      method private draw i j =
        Graphics.set_color col ;
        Graphics.fill_circle (i+r) (j+r) (r-2)
    end ;;

# class file_bitmap_item name =
    let ch = open_in name
    in let x = Marshal.from_channel ch
    in let y = Marshal.from_channel ch
    in let im = Marshal.from_channel ch
    in let () = close_in ch
    in object
    inherit graph_item x y (Graphics.make_image im)
       end ;;
```

graph_item を、ディスクから読み込まれる十字、ディスクその他のビットマップで特殊化します。

抽象グラフィックロボット　は、ロボットでもあり、グラフィックオブジェクトでもあります。

```
# class virtual graph_robot i0 j0 =
    object
      inherit robot i0 j0
      inherit graph_object
    end ;;
```

固定ロボット、気違いロボット、頑固ロボットのグラフィックロボット　は、特殊化されたグラフィックオブジェクトです。

```
# class fix_graph_robot i0 j0 =
    object
      inherit fix_robot i0 j0
      inherit disk_item 7 Graphics.green
    end ;;

# class crazy_graph_robot i0 j0 =
    object
```

```
      inherit crazy_robot i0 j0
      inherit file_bitmap_item "crazy_bitmap"
     end ;;

# class obstinate_graph_robot i0 j0 =
    object
      inherit obstinate_robot i0 j0
      inherit square_item 15 Graphics.black
    end ;;
```

対側型グラフィックロボット　は、次の移動を決めるために、モジュール Graphics のプリミティブ key_pressed と read_key を使用しています。ここでも、テンキー（NumLock ボタンが押されている時）の 8、6、2、4 を押すことができます。このようにして、ユーザはシミュレーションの各ステップ毎に方向を入力する必要はありません。

```
# class interactive_graph_robot i0 j0 =
    object
      inherit interactive_robot i0 j0
      inherit file_bitmap_item "interactive_bitmap"
      method private get_move () =
        if not (Graphics.key_pressed ()) then Nothing
        else match Graphics.read_key() with
          '8' → North | '2' → South | '4' → West  | '6' → East | _ → Nothing
    end ;;
```

グラフィカルな世界

パラメータ'a_robot をグラフィカルロボットの抽象クラス graph_robot で実体化して純粋仮想世界から継承することで、グラフィカルインターフェースを持つ仮想世界を得ることができます。テキストモードの仮想世界と同様に、グラフィカルな仮想世界は、一般的活性化メソッド run と同様に、ロボットの振る舞いを実装するためにそれ自身のメソッド run_robot を持ちます。

```
# let delay x = let t = Sys.time () in while (Sys.time ()) -. t < x do () done ;;

# class virtual graph_world l0 h0 =
    object(self)
      inherit [graph_robot] world l0 h0 as super
      initializer
        let gl = (l+2)*15 and  gh = (h+2)*15 and lw=7 and cw=7
        in Graphics.open_graph (" "^(string_of_int gl)^"x"^(string_of_int gh)) ;
           Graphics.set_color (Graphics.rgb 170 170 170) ;
           Graphics.fill_rect 0 lw gl lw ;
           Graphics.fill_rect (gl-2*lw) 0 lw gh ;
           Graphics.fill_rect 0 (gh-2*cw) gl cw ;
```

```
      Graphics.fill_rect lw 0 lw gh

    method run_robot r = let p = r#next_pos ()
                         in delay 0.001 ;
                            if (self#is_legal p) & (self#is_free p)
                            then ( r#set_pos p ; self#display_robot r)

    method display_robot r = let (i,j) = r#get_pos
                             in r#display (i*15+15) (j*15+15)

    method run() = List.iter self#display_robot robots ;
                   super#run()
  end ;;
```

グラフィカルウインドウは、このクラスのオブジェクトが初期化される時に生成されることに注意してください。

平面矩形グラフィック仮想世界 は、平面矩形テキスト仮想世界とほとんど同じ方法で得ることができます。

```
# class closed_graph_world l0 h0 =
  object(self)
   inherit graph_world l0 h0
   method is_legal (i,j) = (0<=i) & (i<l) & (0<=j) & (j<h)
  end ;;
class closed_graph_world :
  int ->
  int ->
  object
    val h : int
    val l : int
    val mutable robots : graph_robot list
    method add : graph_robot -> unit
    method display_robot : graph_robot -> unit
    method is_free : int * int -> bool
    method is_legal : int * int -> bool
    method run : unit -> unit
    method run_robot : graph_robot -> unit
  end
```

グラフィック版のアプリケーションは、以下のように入力することでテストすることができます。

```
let w = new closed_graph_world 10 10 ;;
w#add (new fix_graph_robot 3 3) ;;
w#add (new crazy_graph_robot 2 2) ;;
w#add (new obstinate_graph_robot 1 1) ;;
```

```
w#add (new interactive_graph_robot 5 5) ;;
w#run () ;;
```

さらに学びたい人のために

異なる仮想世界でのメソッド run_robot の実装は、ロボットが潜在的に仮想世界の任意の位置へ、その移動先が空いていてかつ正しい時には、移動することが出来ることを示唆しています。残念ながら、ロボットが任意の場所に移動するのを妨げるものは何もありません。仮想世界はそれを止めることは出来ないのです。一つの改善方法は、仮想世界から制御可能なロボットの位置を持たせる、というものです。ロボットが移動しようとした時には、仮想世界は新しい位置に移動可能かどうかだけでなく、その移動が公認された移動であるかどうかも検証します。この場合、ロボットのクラスが仮想世界のクラスとは独立していなければいけないという結果から、ロボットは実際の位置を仮想世界に問い合わせることができる必要があります。ロボットクラスは、型パラメータとして、仮想世界のクラスを取ります。

この変更により、ロボットが自分の走行する仮想世界に問い合わせがそして、仮想世界とは独立して振る舞うことが出来るようにロボットを定義できるようになります。そして、他のロボットを追い掛けたり、避けたり、邪魔したりといったことをするロボットを実装することができます。

他の拡張としては、ロボットがお互いに通信し、情報を交換し、チームを作れるようにするものがあるでしょう。

以降の章では、ロボットの実行をお互いに独立に行えるような機能を見ていきます。つまり、スレッドを用いることで Threads（601 ページ参照）、ロボットは独立したプロセスとして実行されます。これにより、ロボットが**クライアント**になりリモートマシンで実行され、移動を通知したり *server* として振る舞う仮想世界からの他の情報についての問い合わせたりというような分散計算（625 ページ参照）を行うこともできる利点があります。この問題は、658 ページで取り上げます。

Part IV

並行分散プログラミング

第四部では共有メモリモデル、分散メモリモデルを使い並列プログラミングの概念について学びます。並行アルゴリズムを表現したり、分散アプリケーションを実装するのに並列スーパーコンピュータは必要ありません。ここではまずこの後の章で使うことになる用語について説明します。

逐次プログラミング *sequential programming* では命令は順番に実行されていました。言い替えると実行される命令の間には因果関係がありました。逐次プログラムは決定性を持っています。つまり同じ入力に対しては常に停止するか、または常に無限ループに陥ります。停止する場合には常に同じ結果を生じます。決定性とは同じ状況であれば常に同じ結果へと導かれることを表しています。これまで見た中で Objective Caml での例外は外部の情報を取得する `Sys.time` のような関数だけです。

並列プログラミング *parallel programming* ではプログラムはいくつかの能動的なプロセスに分割されています。それぞれのプロセスは逐次的ですが、別のプロセスに属する命令は並列に「同時に」実行されます。同時に実行される命令の間には因果関係はありません。同じ状況で実行をはじめても異なった結果を生じる可能性があります。ある命令の結果が他の命令によってキャンセルされることもあります。したがって並列プログラムは決定的ではありません。同じプログラムを同じ入力で実行しても常に停止するとか限りません。また停止した場合でも常に同じ結果を生み出すとは限りません。

並列プログラムの実行を制御するために新しい二つの概念が必要です。

- 複数のプロセスの間で待ち合わせる手段である同期。
- 複数のプロセスの間のコミュニケーション手段であるメッセージ通信。

同期とはある決まった時点に関係するすべてのプロセスが到達するまでプロセスの実行を制御する手段です。因果関係として考えると同期は、ある出来事を起こす前に複数の主体の間で同じ状況を作り出すことができます。メッセージ通信は限定された時間的な制約 (メッセージを送る前にそのメッセージは受け取られることはない) を作り出すことができます。メッセージ通信には一対一、一対多、多対多などの様々な形態があります。

図 17.13 は並列プログラミングで使われる代表的なメモリモデルを表しています。メモリモデルの違いによって同期と通信の方法に差ができてきます。

それぞれのプロセス P_i が逐次プログラムの実行に対応しています。プロセスは共有メモリ (**M**) を利用するか、または通信媒体 (**medium**) を利用して相互作用を行います。この全体が一つの並列アプリケーションを表しています。

共有メモリモデル　共有メモリモデルでは、プロセスはあらかじめ定められたメモリ領域を利用して間接的に通信を行います。定められたメモリ領域に値を書き込むことがメッセージの送信であり、その領域を読むことがメッセージの受信となります。同期を行うためには、共有メモリを利用して相互排他等の機能を実現します。

このモデルは共有資源に対して並列にアクセスできる場合に使われます。ほとんどのオペレーティングシステムは共有メモリモデルを採用しています。

分散メモリモデル　このモデルではそれぞれの逐次プロセス P_i は、他のプロセスから直接アクセスできない自分だけのメモリ M_i を持っています。プロセスはメディアを通

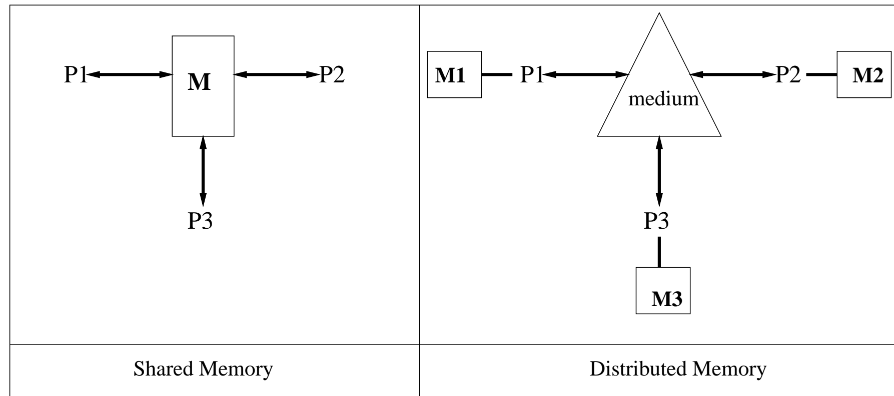


図 17.13: メモリモデル

して情報を送りコミュニケーションを行います。このモデルではメディアの実装が主要な問題となります。メディアの使って他のプロセスと通信するやり方をプロトコルと言います。

プロトコルは層状になっています。下位プロトコルの原始的な機能を利用して、上位プロトコルでは洗練された機能が提供されています。

コミュニケーションにはいくつか種類があります。メディアに情報を蓄える機能があるかどうかでブロック型、非ブロック型のコミュニケーションのどちらになるか決まります。情報の送り手と受け手が同期しなければ情報を送れない場合、同期型通信 *synchronous communication* と呼びます。

メディアが情報を蓄える機能を持っている場合、メッセージを一度保存することができるので非同期、非ブロック型のコミュニケーションを行えます。メディアには保存できるメッセージの容量、メッセージ配送の順序、遅延時間、信頼性などの違いがあります。

最後にメディアが情報を蓄える機能を持たずに非同期型の配送を行う場合があります。この場合、受け手のプロセスがちょうど準備ができている時に限り、メッセージを受け取ることができますが、そうでない時はメッセージが失われることになります。

分散メモリモデルでは通信は明示的ですが、同期はメッセージ通信に付随して非明示的に起こります。これは共有メモリモデルの場合と対称的になっています。

物理的並列性と論理的並列性　分散メモリモデルは物理的並列性と論理的並列性の両方を持っています。物理的並列性とは、ネットワークでつながれた物理的に別々のコンピュータなどを意味しています。論理的並列性とは、一つのコンピュータの内部での Unix のプロセス同士の持つ並列性などを意味しています。

共有メモリを持たない分散メモリモデルは強い物理的並列性を持っていますがネットワークを利用して共有メモリモデルをシミュレートすることは可能です。

第 4 部では Objective Caml を使って共有メモリモデル、分散メモリモデルの両方の並列アプリケーションを作る方法を学びます。アプリケーションを実装するにあたって、Unix システムコールインターフェースを提供している `Unix` ライブラリと軽量なプロセスを実装している `Thread` ライブラリを利用します。`Unix` ライブラリの大部分、とりわけ記述子を扱う関数は Windows 環境へも移植されています。記述子を扱う関数はファイルの入出力、パイプ *pipe* やソケット *socket* を利用した通信を扱う機能を担当しています。

第 18 章では `Unix` の本質的な概念について解説します。特にプロセスが、外界や他のプロセスと通信を行う方法について詳しく見ていきます。この章ではプロセスとは「Unix のプロセス」を意味しています。プロセスは `fork` システムコールによって生成されます。`fork` システムコールはそれを実行したプロセスの実行環境やメモリイメージを複製することで新しいプロセスを生成し、元のプロセスの親子関係のリンクに登録します。プロセス同士の相互作用はシグナルとパイプを使って実装されています。

第 19 章は `Thread` ライブラリについて解説します。スレッドは Unix のプロセスとは異なり、実行コンテクストだけを新しく作成するため Unix プロセスよりも非常に軽くなっています。同じプロセスが生成したスレッドの間ではメモリが共有されています。プログラミングスタイルに応じて Objective Caml 軽量プロセスは共有メモリモデル (命令型スタイル) と独立メモリモデル (関数型スタイル) の両方の機能を提供しています。`Thread` ライブラリではスレッドを使ったプログラミングを支援するいくつかのモジュールを提供しています。それらのモジュールでは、スレッドを実行開始、停止する機能、ロックと排他制御の機能、チャンネルを使ったスレッド間通信の機能などを提供しています。このモデルでは並列実行のために全体の実行時間が短縮されることはありませんが、並列アルゴリズムの定式化が簡単になります。

第 20 章はインターネット上で動作する分散アプリケーションの構築を目指しています。この章ではインターネットの低レベルプロトコルを利用します。別々のマシンで動いているプロセスは通信ソケットを使ってお互いに通信を行うことができます。ソケットを利用した通信は非同期的でほとんどの場合 1 対 1 です。分散アプリケーションを構成するプロセスが通信を行う時、それらのプロセスの役割は一般に非対称的です。特に多くのタイプはクライアントサーバ型アーキテクチャと呼ばれています。サーバはクライアントから要求が来るのを待っており要求が来たら、その要求に応じて適切な処理を行います。クライアントはサーバに要求を送り、結果を待ちます。インターネットの多くのサービスはこのようなアーキテクチャにしたがっています。

第 21 章は一つのライブラリと二つの完結したアプリケーションを扱います。このライブラリはプロトコルを与えるとクライアントとサーバの間の通信機構を提供します。最初のアプリケーションは第 17 章で扱ったロボットの分散バージョンです。二つ目のアプリケーションは第 6 章で扱ったデータベースを今度は HTTP サーバとして構築します。

18

通信とプロセス

プログラミング言語とオペレーティングシステムの間のインターフェースの中で通信とプロセスという概念は極めて重要です。この章ではこの二つの概念について取り組んでいきます。第 8 章で既に紹介しましたが Sys モジュールを使えば別のプログラムに値を渡したり、起動したりすることができます。これはプロセスと通信について一種の予備的な学習をしていたことになりますが、この章ではプロセスの概念とプロセス間通信について更に詳しく見ていきます。

「プロセス」という用語は実行中のプログラムを意味しています。プロセスは並列アプリケーションを考える時のの基本的な概念です。この章では Unix システムで使われている古典的なモデルに基づいてプロセスを説明します。このプロセスモデルではプロセスは別のプロセスによって作られ、作った方と作られた方の間に親子関係が成り立ちます。親プロセスは子プロセスが終了するのを待つことができます。並列実行されているプロセスは原則として分散メモリモデルに従っています。

「通信」という用語には少なくとも 3 つの側面があります。

- まず**記述子**を使った入出力があります。Unix でのファイル記述子は単なる記憶メディア上の読み書きという意味だけではなく、非常に広い意味を持っています。第 20 章では記述子を使った、別のコンピュータ上のプログラムとの通信について扱います。
- 次に 2 つのプロセスの間の**パイプ**を使った通信があります。これはプロセス間で待ち行列の原則に基づいた文字データの交換を行う機構です。
- 最後に**シグナル**があり、これは特別な機能を提供しています。

この章で使われている Unix の関数は Objective Caml の配布ファイルに含まれている `Unix` モジュールの関数を意味しています。関数の仕様や名前の付け方などの慣習は Unix から取られたものですが Windows プラットフォームでも多くの関数が利用できます。プラットフォームによる利用可能な関数の違いは本文中で示されています。

この章のあらまし

第 1 節は Unix モジュールの使い方を簡単に見ていきます。特にエラー処理の仕方と Windows 環境での互換性について扱います。

第 2 節では Unix の意味でのファイル記述子について解説します。Pervasives モジュールで提供されているものよりも低レベルの入出力を扱います。

プロセスは第 3 節で紹介されています。プロセスの生成、終了、Unix モデルでのプロセスの親子関係について扱います。

第 4 節はパイプとシグナルというプロセス間の基本的な通信手段について解説します。

最後の 2 節は第 19 章と第 20 章の予習として軽量プロセスとソケットについて扱います。

Unix モジュール

このモジュールはほとんどの重要な Unix ライブラリ関数のインターフェースを提供しています。関数の大部分は Windows へ移植され利用できます。必要なときにはどの関数が使えないかどうか説明をしますが、図 18.1 に Windows 上で使用できる関数の概略が示されています。

Unix ライブラリは Objective Caml の標準ライブラリに含まれていません。そのため -custom オプション (第 7 章 197 ページ) を指定した場合などは Unix ライブラリをリンクするように明示的にコンパイラに指示しなければなりません。バイトコード、ネイティブコード、トップレベルを生成するにはそれぞれ以下のように指定する必要があります。

```
$ ocamlc -custom unix.cma  fichiers.ml -cclib -lunix
$ ocamlopt unix.cma fichiers.ml -cclib -lunix
$ ocamlmktop -custom -o unixtop unix.cma -cclib -lunix
```

トップレベルを生成する機能があるのは、小さな部分ごとに仕様を決定し、テストを行いながら大きなアプリケーションを開発することを可能にするためです。

プラットフォームの Unix のバージョンの違いによりシステムライブラリが想定された場所にないかもしれません。その時は-ccopt オプション (第 7 章) を使ってライブラリのパスを指定しなければなりません。

Windows 環境でコンパイルするには次のように指定します。

```
$ ocamlc -custom unix.cma  fichiers.ml %CAMLLIB%\libunix.lib wsock32.lib
$ ocamlopt unix.cma fichiers.ml %CAMLLIB%\libunix.lib wsock32.lib
$ ocamlmktop -custom -o unixtop.exe unix.cma %CAMLLIB%\libunix.lib wsock32.lib
```

生成されるトップレベルの名前は unixtop.exe になります。

エラー処理

Objective Caml のプログラムの中でシステムコールを呼び出し、エラーが発生した場合 Unix_error 例外が投げられます。Unix_error 例外を補足することでエラー処理を記述できます。この例外は 3 つの値を持っています。1 つ目はエラーの種類を示す値で、この値は error_message 関数を呼び出してエラーメッセージを得ることができます。2 つ目の値はエラーを発生させた関数の名前の文字列です。3 つ目の値はエラーを発生させた関数の引数が文字列であった時にその引数を保持しています。

一般的なエラー関数は以下のようになります。

```
# let wrap_unix funct arg =
    try (funct arg) with
     Unix.Unix_error (e,fm,argm) →
       Printf.printf "%s %s %s" (Unix.error_message e) fm argm ;;
val wrap_unix : ('a -> unit) -> 'a -> unit = <fun>
```

この関数は引数として関数とその引数を受け取り、実行させます。関数の実行中に Unix のエラーが発生すればエラーに関する情報を表示します。これと同じ働きをする関数が Unix モジュールでも定義されています。

```
# Unix.handle_unix_error ;;
- : ('a -> 'b) -> 'a -> 'b = <fun>
```

システムコールの互換性

通信とプロセスに関係する関数のうちこの章で使われているものが Windows 上で動作するかどうか図 18.1 に示しました。中でも重要なのは新しいプロセスを生成する fork とプロセスにシグナルを送る kill が使えないことです。

追加として fork が実装されていないため子プロセスの終了を待つ関数である wait も実装されていません。

ファイル記述子

第 3 章で組み込みモジュールである Pervasives モジュールの関数について説明しました。これらの関数は入出力チャネルを使ってファイルにアクセスする機能を提供していました。ファイル記述子はそれよりも低いレベルでファイルにアクセスする方法を与えています。

ファイル記述子とは Unix.file_descr 型の抽象的な値であり、ファイルへの参照、アクセス権、アクセスモード (読み書き)、ファイル内の位置などのファイルを扱うのに必要な情報が含まれています。

これらの記述子は標準で定義されており、それぞれ標準入力、標準出力、標準エラー出力に対応しています。

関数	Unix	Windows	備考
openfile	×	×	
close	×	×	
dup	×	×	
dup2	×	×	
read	×	×	
write	×	×	
lseek	×	×	
execv	×	×	
execve	×	×	
execvp	×	×	
execvpe	×	×	
fork	×		create_process を使用のこと
getpid	×	×	
sleep	×	×	
wait	×		
waitpid	×	×	PID を明示的に与える必要あり
create_process	×	×	
create_process_env	×	×	
kill	×		
pipe	×	×	
mkfifo	×		
open_process	×		コマンドは /bin/sh で解釈
close_process	×		

図 18.1: この章で使われる Unix モジュールの関数の互換性

```
# ( Unix.stdin , Unix.stdout , Unix.stderr ) ;;
- : Unix.file_descr * Unix.file_descr * Unix.file_descr =
(<abstr>, <abstr>, <abstr>)
```

以上の記述子は標準入出力チャネルとは違うので注意してください。

```
# ( Pervasives.stdin , Pervasives.stdout , Pervasives.stderr ) ;;
- : in_channel * out_channel * out_channel = (<abstr>, <abstr>, <abstr>)
```

記述子とチャネルの間の変換関数は 581 ページで扱っています。

File Access Rights. Unix ではすべてのファイルには所有者とグループ[1]が定義されています。(1) ファイルの所有者 (2) グループのメンバー (3) それ以外のユーザのそれぞれに対して読み書き実行の権限を設定できるようになっています。

この 3 種類のユーザに対してそれぞれ 3 つの権限があるので一般に 9 ビットの表記法でアクセス権を表記します。最初の 3 ビットが所有者の権限であり、次の 3 ビットがグループの権限であり、最後の 3 ビットがその他のユーザの権限を表しています。3 ビットの権限の中では最初のビットが読む権限であり、次のビットが書く権限であり、最後のビットが実行する権限を表しています。これらのビットは一般にそれぞれ `r`, `w`, `x` と省略するのが普通で、また「権限を持たないこと」は「`-`」によって表記します。例えば所有者は読み書きできるが、それ以外のユーザは読むことしかできないファイルは `rw-r--r--`と表記されます。この権限は整数で `420` (二進数で `0b110100100` になる) とも表され、また 8 進数で `0o644` と書く表記もよく使われています。これらのアクセス権は Windows では正常に解釈されないことがあります。

ファイル操作

ファイルを開く ファイルを開くとそのファイルの記述子が得られます。ファイルを使用する目的によってファイルのモードを指定することができます。ファイルのモードは図 18.2 に示されている *open_flag* 型の値によって指定します。

O_RDONLY	読み込み専用
O_WRONLY	書き込み専用
O_RDWR	読み書き両用
O_NONBLOCK	非ブロック型
O_APPEND	ファイルの最後に追加
O_CREAT	ファイルがない時は新しく作る
O_TRUNC	ファイルがある時に長さを 0 にする
O_EXCL	ファイルがある時にエラーになる

図 18.2: *open_flag* 型の値

これらのモードは複数組み合わせることができます。`openfile` 関数は *open_flag* 型の値のリストを引数として受け取ります。

```
# Unix.openfile ;;
- : string -> Unix.open_flag list -> Unix.file_perm -> Unix.file_descr =
<fun>
```

第一引数にはファイルの名前を指定します。最後の引数は整数[2]です。アクセス権はファイルを新しく作成する場合のみ意味を持ちます。

読み込みのためにファイルを開く例を以下に示します。もしファイルが存在しない場合には新しくファイルを作成し `rw-r--r--`というアクセス権を設定します。

1. Unix では個々のユーザは一つ以上のグループに所属することになっています。グループはアクセス権を管理する単位を提供しています。
2. *file_perm* 型は *int* 型の別名です。

```
# let file = Unix.openfile "test.dat" [Unix.O_RDWR; Unix.O_CREAT] 0o644 ;;
val file : Unix.file_descr = <abstr>
```

ファイルを閉じる　Unix.close 関数にファイル記述子を引数として適用するとそのファイル記述子に結びつけられたファイルを閉じます。

```
# Unix.close ;;
- : Unix.file_descr -> unit = <fun>
# Unix.close file ;;
- : unit = ()
```

ファイル記述子のリダイレクト　同じファイルを指すファイル記述子が複数必要になる時があります。このような時にファイル記述子のコピーを作ることができます。

```
# Unix.dup ;;
- : Unix.file_descr -> Unix.file_descr = <fun>
```

あるファイル記述子を別のファイル記述子に結びつけることができます。

```
# Unix.dup2 ;;
- : Unix.file_descr -> Unix.file_descr -> unit = <fun>
```

例えば標準エラー出力に表示された内容を特定のファイルに書き込むようにしたい場合には以下のようにします。

```
# let error_output = Unix.openfile "err.log" [Unix.O_WRONLY;Unix.O_CREAT] 0o644 ;;
val error_output : Unix.file_descr = <abstr>
# Unix.dup2 Unix.stderr error_output ;;
- : unit = ()
```

このようにすると標準エラー出力へ書き込まれた内容は err.log へと書き込まれます。

ファイル入出力

Unix.read と Unix.write はファイルに対して文字列を読み書きする関数です。

```
# Unix.read ;;
- : Unix.file_descr -> string -> int -> int -> int = <fun>
# Unix.write ;;
- : Unix.file_descr -> string -> int -> int -> int = <fun>
```

ファイル記述子と文字列の他に整数を 2 つ引数に与えています。最初の整数が文字列の中の読み書きする位置を示しており、2 つ目の整数が読み書きする文字数を表しています。結果として返される整数は実際に読み書きした文字数を表しています。

```
# let mode = [Unix.O_WRONLY;Unix.O_CREAT;Unix.O_TRUNC] in
  let fl = Unix.openfile "file" mode 0o644 in
  let str = "012345678901234565789" in
  let n = Unix.write fl str 4 5
  in Printf.printf "We wrote %s to the file\n" (String.sub str 4 n) ;
```

```
      Unix.close fl ;;
We wrote 45678 to the file
- : unit = ()
```

ファイルからの読み込みも全く同様です。

```
# let fl = Unix.openfile "file" [Unix.O_RDONLY] 0o644 in
  let str = String.make 20 '.' in
  let n = Unix.read fl str 2 10 in
   Printf.printf "We read  %d characters" n;
   Printf.printf " and got the string %s\n" str;
   Unix.close fl ;;
We read  5 characters and got the string ..45678............
- : unit = ()
```

通常はファイルの中でそれまでに読み書きした位置から続けて読み書きを行いますが、この位置を変更することができます。

```
# Unix.lseek ;;
- : Unix.file_descr -> int -> Unix.seek_command -> int = <fun>
```

最初の引数がファイル記述子で、2 番目の引数は相対位置を表しています。最後の引数は第 2 引数で指定された位置がファイルのどの場所を基準にしたものか指定しています。この基準には 3 種類の指定ができます。

- SEEK_SET: ファイルの先頭を基準
- SEEK_CUR: ファイルの現在位置を基準
- SEEK_END: ファイルの最後を基準

指定した位置に矛盾がある時は例外が発生するか、または結果として 0 が返されます。

入出力チャネル Unix モジュールはファイル記述子と Pervasives モジュールの入出力チャネルの間の変換関数を提供しています。

```
# Unix.in_channel_of_descr  ;;
- : Unix.file_descr -> in_channel = <fun>
# Unix.out_channel_of_descr ;;
- : Unix.file_descr -> out_channel = <fun>
# Unix.descr_of_in_channel  ;;
- : in_channel -> Unix.file_descr = <fun>
# Unix.descr_of_out_channel ;;
- : out_channel -> Unix.file_descr = <fun>
```

変換によって得られた入出力チャネルはバイナリモードとテクストモードの指定を明示的に行わなければなりません。

```
# set_binary_mode_in ;;
- : in_channel -> bool -> unit = <fun>
# set_binary_mode_out ;;
- : out_channel -> bool -> unit = <fun>
```

以下の例では Unix モジュールの関数を利用してファイルを作成し、その後に Unix モジュールの関数を使ってファイルを開き、高レベル入力関数である input_line を使ってファイルの内容を読み込んでいます。

```
# let mode = [Unix.O_WRONLY;Unix.O_CREAT;Unix.O_TRUNC] in
  let f = Unix.openfile "file" mode 0o666 in
  let s = "0123456789\n0123456789\n" in
  let n = Unix.write f s 0 (String.length s)
  in Unix.close f ;;
- : unit = ()
# let f = Unix.openfile "file" [Unix.O_RDONLY;Unix.O_NONBLOCK] 0 in
  let c = Unix.in_channel_of_descr f in
  let s = input_line c
  in print_string s ;
     close_in c ;;
0123456789- : unit = ()
```

アクセス可能性　同時に複数の入出力を扱わなければならない時があります。データがどのチャネルにやってくるのか分からずに、しかも特定のチャネルで入力待ち状態になることも許されない場合があります。次の関数は記述子のリストを渡すとアクセスが可能なものだけを返してくれる関数です。

```
# Unix.select ;;
- : Unix.file_descr list ->
    Unix.file_descr list ->
    Unix.file_descr list ->
    float ->
    Unix.file_descr list * Unix.file_descr list * Unix.file_descr list
= <fun>
```

最初の 3 つの引数はそれぞれ入力、出力、エラー出力の記述子を表しています。最後の引数はこの関数の遅延時間を意味しており、負の遅延時間は待たないことを指定します。結果として返されるのはそれぞれ利用可能な入力、出力、エラー出力の記述子のリストです。

警告　select は Windows 環境では実装されていません。

プロセス

Unix ではプロセスとはプログラムの実行を意味しています。文献 [CDM98b] で Card と Dumas と Mével はプログラムとプロセスの違いを次のように説明しています。「プログラム自身はプロセスではない。プログラムはディスク上に置かれた実行可能ファイルという受動的な存在であるが、プロセスとは次に実行する命令を指すプログラムカウンタや様々な資源を持つ能動的な存在である。」

Unix は **マルチタスク** オペレーティングシステムです。複数のプロセスを同時に実行させることができます。

プロセスは **プリエンプティブ** です。すなわちプロセスの実行はオペレーティングシステムの特別なプロセスによって管理されています。この意味でプロセスは自分の資源を完全に管理しているわけではありません。特に実行時間を自分で決定することはできません。

プロセスは独自のメモリを持っており、他のプロセスとは通常ファイルや通信チャネルを通して情報の交換をします。したがってプロセスは一つのマシン上で分散メモリモデルをシミュレートしています。

システムはプロセスごとに PID (プロセス ID) と呼ばれる固有の識別子を割り当てています。Unix では最初のプロセスを除いてすべてのプロセスは**親プロセス**と呼ばれることになる別のプロセスによって生成されます。

すべてのプロセスのリストは Unix コマンドの ps[3] によって表示させることができます。

```
$ ps -f
PID     PPID     CMD
1767    1763    csh
2797    1767    ps -f
```

`-f` オプションを付けるとそれぞれのプロセスに対してプロセス ID(`PID`)、親プロセスのプロセス ID(`PPID`)、プログラムの名前、引数 (`CMD`) を表示します。この例ではコマンドラインインタープリタ `csh` と `ps` 自身の 2 つのプロセスが表示されています。`ps` の親プロセスは `csh` となっています。

プログラムの実行

実行コンテクスト

実行中のプログラムには 3 つの値が結びつけられています。

1. プログラムを実行したコマンド行の内容。この内容は `Sys.argv` に保存されています。
2. プログラムを実行した環境。この内容は `Sys.getenv` を利用して取り出すことができます。
3. プログラムが終了した時のステータス。

コマンド行　コマンド行にはプログラムを実行する時の引数やオプションを指定することができます。プログラムの振る舞いは引数に指定したものによって変化するかもしれません。以下に例を示します。このプログラムを `argv_ex.ml` というファイルに保存します。

3. この命令のオプションは標準化されていないので、システムによってこの通りに表示されないことがあります。

```
if Array.length Sys.argv = 1 then
 Printf.printf "Hello world\n"
else if Array.length Sys.argv = 2 then
 Printf.printf "Hello %s\n" Sys.argv.(1)
else Printf.printf "%s : too many arguments\n" Sys.argv.(0)
```

このプログラムを次のようにコンパイルします。

```
$ ocamlc -o argv_ex argv_ex.ml
```

実行してみましょう。

```
$ argv_ex
Hello world
$ argv_ex reader
Hello reader
$ argv_ex dear reader
./argv_ex : too many arguments
```

環境変数　実行中に必要な情報を環境変数から読み込みたい時があります。環境変数の名前や数はオペレーティングシステムやユーザの環境設定に依存しています。環境変数の値には getenv 関数を使ってアクセスできます。この関数の引数にはアクセスしたい環境変数の名前を文字列として渡します。

```
# Sys.getenv "HOSTNAME";;
- : string = "zinc.pps.jussieu.fr"
```

実行ステータス

プログラムの実行結果とは通常は整数であり、プログラムが正常に終了したかどうかを表しています。この値の意味はオペレーティングシステムによって変わる可能性があります。プログラムを直ちに終了させ、実行ステータスを返すには次の関数を呼び出します。

```
# Pervasives.exit ;;
- : int -> 'a = <fun>
```

プロセス生成

プログラムの実行は別のプロセスによって開始されます。このプロセスを親プロセスと呼びます。プログラムの実行は新しいプロセスによって担われており、これを子プロセスと呼びます。この 2 つのプロセスの間には 3 種類の関係がありえます。

- 二つのプロセスは独立であり、並行に実行される。
- 親プロセスは子プロセスの終了を待っている。

- 新しいプロセスは親プロセスに取って換わる。

親プロセスが新しいプロセスとして自分の複製を作ることもできます。自分と複製の間では PID だけが異なっています。この処理は有名な `fork` 関数によって行います。`fork` 関数についてはこの後で扱います。

独立プロセス

`Unix` モジュールにはプロセスを生成する、互換性のある関数が提供されています。

```
# Unix.create_process ;;
- : string ->
    string array ->
    Unix.file_descr -> Unix.file_descr -> Unix.file_descr -> int
= <fun>
```

最初の引数はプログラムの名前 (パス) で、2 番目の引数はプログラムへ渡す引数の配列です。最後の 3 つの引数はそれぞれ標準入力、標準出力、標準エラー出力を表す記述子です。この関数の帰り値は生成されたプロセスの PID です。

同じような働きをする関数で環境変数の値を渡す機能を持つものもあります。

```
# Unix.create_process_env ;;
- : string ->
    string array ->
    string array ->
    Unix.file_descr -> Unix.file_descr -> Unix.file_descr -> int
= <fun>
```

これらの関数は Unix と Windows のどちらの環境でも利用できます。

プロセススタック

新しいプロセスを並行に実行させるのが不便な時もあります。子プロセスの実行が終了するまで親プロセスの作業を進められない場合です。次の 2 つの関数は命令を引数として受け取り、実行します。

```
# Sys.command;;
- : string -> int = <fun>
# Unix.system;;
- : string -> Unix.process_status = <fun>
```

この 2 津の関数は帰り値の型が違っています。*process_status* 型については 590 により詳しく説明されています。子プロセスを実行している間、親プロセスの実行はブロックされます。

プロセスの置き換え

現在実行しているプロセスを別の新しいプロセスで置き換えると、並行に実行されているプロセスの数を押えることができます。以下の 4 つの関数によってこの処理を行えます。

```
# Unix.execv ;;
- : string -> string array -> unit = <fun>
# Unix.execve ;;
- : string -> string array -> string array -> unit = <fun>
```

```
# Unix.execvp ;;
- : string -> string array -> unit = <fun>
# Unix.execvpe ;;
- : string -> string array -> string array -> unit = <fun>
```

最初の引数は実行するプログラムの名前です。execvp と execvpe ではプログラムをパスを使って検索します。2 番目の引数は実行するプログラムへの引数です。execve と execvpe の最後の引数では環境変数を指定できます。

複製によるプロセス生成

Unix でプロセスを生成するシステムコールはもともと次のような名前でした。

```
# Unix.fork ;;
- : unit -> int = <fun>
```

fork 関数は新しいプロセスを開始させますが、新しいプログラムとしてではなく、fork 関数を呼び出したプロセスの複製を生成します。したがって新しいプロセスのコードは親プロセスと全く同じです。Unix では同じコードを複数のプロセスの間で共有することができます。この場合でも個々のプロセスは自分固有の実行コンテクストを持っています。このような共有コードを**リエントラント**であると言います。

次の小さなプログラムを見てみましょう。このプログラムでは、呼び出したプロセスの PID を返す getpid 関数を使っています。

```
Printf.printf "before fork : %d\n" (Unix.getpid ()) ;;
flush stdout ;;
Unix.fork () ;;
Printf.printf "after fork : %d\n" (Unix.getpid ()) ;;
flush stdout ;;
```

このプログラムを実行すると次のような結果が得られます。

```
before fork : 12450
after fork : 12450
after fork : 12451
```

fork を呼んだ後は 2 つのプロセスがこのプログラムを実行しています。このために “after fork” の PID が 2 つ表示されています。ここで一方は最初のプロセスの PID と同じですが、他方は PID が別の値になっています。この新しい PID が子プロセスの PID であり、fork 関数の帰り値とも一致しています。fork 関数の帰り値は親プロセスでは生成された子プロセスの PID であり、子プロセスでは 0 になります。

共有されたプログラムの中で、自分を実行しているのが親プロセスなのか子プロセスなのか知る方法は fork の帰り値を調べることです。

```
Printf.printf "before fork : %d\n" (Unix.getpid ()) ;;
flush stdout ;;
let pid = Unix.fork () ;;
if pid=0 then  (* -- Code of the child *)
  Printf.printf "I am  the child: %d\n" (Unix.getpid ())
else           (* -- Code of the father *)
  Printf.printf "I am the father: %d of child: %d\n"  (Unix.getpid ()) pid ;;
flush stdout ;;
```

このプログラムを実行すると以下のような結果になります。

```
before fork : 12460
I am the father: 12460 of child: 12461
I am  the child: 12461
```

帰り値にパターンマッチを使うこともできます。

```
match Unix.fork () with
   0  → Printf.printf "I am the child: %d\n" (Unix.getpid ())
| pid → Printf.printf "I am the father: %d of child: %d\n"
                       (Unix.getpid ()) pid ;;
```

プロセスはシステムの資源を占有し、また簡単に生成できるため一つのプロセスが生成できる子プロセスの数はオペレーティングシステムの設定によって制限されています。以下の例では 2 世代のプロセスを生成します。プロセス同士は祖父母、親、叔父叔母、いとこ等の関係を持つことになります。

```
let pid0 = Unix.getpid ();;
let print_generation1 pid ppid =
  Printf.printf "I am %d, son of %d\n" pid ppid;
  flush stdout ;;

let print_generation2 pid ppid pppid  =
  Printf.printf "I am %d, son of %d, grandson of %d\n"
                pid ppid pppid;
  flush stdout ;;

match Unix.fork() with
    0 → let pid01 = Unix.getpid ()
         in ( match Unix.fork() with
                  0 → print_generation2 (Unix.getpid ()) pid01 pid0
                | _ → print_generation1 pid01 pid0)
  | _ → match Unix.fork () with
             0 → ( let pid02 = Unix.getpid ()
                     in match Unix.fork() with
                            0 → print_generation2 (Unix.getpid ()) pid02 pid0
                          | _ → print_generation1 pid02 pid0 )
           | _ → Printf.printf "I am %d, father and grandfather\n" pid0 ;;
```

このプログラムの実行結果は以下のようになります。

```
I am 12565, father and grandfather
I am 12566, son of 12565
I am 12570, son of 12566, grandson of 12565
I am 12567, son of 12565
I am 12571, son of 12567, grandson of 12565
```

実行の順序と継続時間

同期をせずにいくつかのプロセスを生成した場合、驚くべき結果をもたらすことがあります。M. Jourdain[4]風の詩を書く次のプログラムによって具体的に見ていきます。

```
match Unix.fork () with
  0 → Printf.printf "fair Marquise " ; flush stdout
| _ → match Unix.fork () with
           0 → Printf.printf "your beautiful eyes " ; flush stdout
         | _ → match Unix.fork () with
                    0 → Printf.printf "make me die " ; flush stdout
                  | _ → Printf.printf "of love\n" ;  flush stdout ;;
```

このプログラムの実行結果は次のようになるかもしれません。

```
of love
fair Marquise your beautiful eyes make me die
```

プロセスの実行順序を何らかの方法で保証できないと、大抵の場合困ります。一般的な言い方をすれば複数のプロセスを利用しているアプリケーションではプロセスを同期させなければならない状況があります。同期を実現する方法は採用している並列モデルによって変わりますが、プロセス間通信や共有メモリを利用して実現します。この話題についてはこの後に続く 2 章の中で詳しく見ていきます。ここでは簡単な方法により詩を改善します。

- 子プロセスが詩を書く前に適切な時間だけ待つようにする。
- 前の句を書く子プロセスの終了を待って、次の句を書くプロセスを生成するようにする。

遅延時間　プロセスは次の関数を呼ぶことで実行を一時停止できます。

```
# Unix.sleep ;;
- : int -> unit = <fun>
```

引数として与えられた数字の秒間、プロセスは実行を一時停止します。

この関数を使ってプログラムを書き換えます。

```
match Unix.fork () with
    0 → Printf.printf "fair Marquise " ; flush stdout
  | _ → Unix.sleep 1 ;
        match Unix.fork () with
            0 → Printf.printf"your beautiful eyes "; flush stdout
          | _ → Unix.sleep 1 ;
            match Unix.fork () with
                0 → Printf.printf"make me die "; flush stdout
              | _ → Unix.sleep 1 ; Printf.printf "of love\n" ; flush stdout ;;
```

4. モリエール『町人貴族』第 2 幕第 4 場
リンク: http://www.site-moliere.com/pieces/bourgeoi.htm

このプログラムを実行すると次のような結果が得られます。

```
fair Marquise your beautiful eyes make me die of love
```

しかしながらこの方法は確実ではありません。何らかの都合により、一時停止したプロセスが実行を再開した後に出力が表示される場合が理論的にあり得るからです。したがってプロセスの実行順序を保証するより確実な方法を選びたいと思います。

子プロセスの終了の待機　親プロセスは次の関数を呼ぶと子プロセスが終了するのを待ちます。

```
# Unix.wait ;;
- : unit -> int * Unix.process_status = <fun>
```

親プロセスの実行は子プロセスの一つが終了するまで一時停止します。もし子プロセスを 1 つも持たない状態で wait を呼び出すと Unix_error 例外が投げられます。wait の帰り値についてはこの後で説明します。ここではとりあえず wait を使って詩を読むプログラムを完成させましょう。

```
match Unix.fork () with
    0 → Printf.printf "fair Marquise " ; flush stdout
  | _  → ignore (Unix.wait ()) ;
          match Unix.fork () with
              0 → Printf.printf "your beautiful eyes " ; flush stdout
            | _ → ignore (Unix.wait ()) ;
                  match Unix.fork () with
                      0 → Printf.printf "make me die " ; flush stdout
                    | _ → ignore (Unix.wait ()) ;
                   Printf.printf "of love\n" ;
                   flush stdout
```

実行させると、確かに正しい結果が得られます。

```
fair Marquise your beautiful eyes make me die of love
```

警告　fork は Unix システムのみに限定された機能です。

プロセスの死、プロセスの葬儀

wait 関数は子プロセスの終了を待つだけの関数ではなく、子プロセスの死を完了させる責任を持っています。

プロセスが新しく生成された時にシステムはそのプロセスをプロセステーブルに登録します。プロセステーブルにはすべてのプロセスが登録されています。プロセスが終了した時にそのプロセスの登録はそのままではプロセステーブルから削除されません。プロセスが終了した時に終了ステータスを親プロセスに伝えるために登録は保存されています。親プロセスは wait を呼んで子プロセスの終了ステータスを受け取る義務があります。これを行わずに放置した場合、そのプロセスはプロセステーブルに残り続けます。これを**ゾンビプロセス**と呼びます。

オペレーティングシステムを立ち上げるとすべてのプロセスの先祖である init と呼ばれる最初のプロセスが実行を開始します。オペレーティングシステムが通常のサービス

を行うようになった後の init の本質的な任務とは、子プロセスが終了する前に親プロセスが死んだ時にその子プロセスを養子として引き取り、親プロセスに代わって wait を呼び出すことです。

特定のプロセスの終了の待機

wait と同様の働きをする関数に waitpid があります。この関数は Unix と Windows の両方の環境で提供されています。

```
# Unix.waitpid ;;
- : Unix.wait_flag list -> int -> int * Unix.process_status = <fun>
```

最初の引数には待機条件を指定します。2 番目の引数には終了を待つプロセスまたはプロセスグループを指定します。

プロセスの終了後、親プロセスは wait 関数または waitpid 関数を呼び出すことで終了したプロセスの数とその終了ステータスを知ることができます。終了ステータスは *Unix.process_status* 型の値です。この型には 3 種類の構築子があり、それぞれ整数を一つ引数として持っています。

- WEXITED n: プロセスは正常に終了し、その帰り値は n です。
- WSIGNALED n: プロセスはシグナル n のために終了しました。
- WSTOPPED n: プロセスはシグナル n を受けて一時停止しました。

最後の値は STOP シグナルを検出することのできる waitpid 関数でのみ意味を持ちます。シグナルについては 594 ページで更に詳しく扱います。

待機義務の放棄

子プロセスの終了の面倒を自分自身で見る代わりに、その義務を別のプロセスに譲り渡すことができます。「二段 fork」をすると、子プロセスの終了を待機する義務を init に譲り渡します。プロセス P_0 がプロセス P_1 を生成し、プロセス P_1 がプロセス P_2 を生成し、プロセス P_1 が終了したとします。この時 プロセス P_2 は「孤児」となり init が面倒を見るようになります。最初の プロセス P_0 はプロセス P_1 に対してのみ wait を発行する義務を持ちますが、プロセス P_1 はすぐに終了してしまうので待ち時間はわずかなものになるでしょう。

具体的なプログラム例は以下のようになります。

```
# match Unix.fork() with                      (* P0 creates P1 *)
    0 → if Unix.fork() = 0 then exit 0 ;      (* P1 creates P2 and terminates *)
        Printf.printf "P2 did its work\n" ;
        exit 0
  | pid → ignore (Unix.waitpid [] pid) ;      (* P0 waits for P1 to terminate *)
          Printf.printf "P0 can do other things without waiting\n" ;;
P2 did its work
P0 can do other things without waiting
- : unit = ()
```

このテクニックは第 20 章でサーバへの要求を扱う時にも使います。

プロセス間通信

複数のプロセスを組み合わせてアプリケーションを構成すると、作業の分割が容易になるかもしれません。もちろんそれぞれのプロセスは完全に独立ではないでしょうが、プロセス間通信を利用して必要な情報のやりとりを行うことができます。

この節では「パイプ」と「シグナル」という二種類のプロセス間通信の方法についてのみ紹介します。この章ではすべてのプロセス間通信の方法について扱いません。第 19 章と第 20 ではこの章とは別のプロセス間通信の方法を利用してアプリケーションを開発します。

通信パイプ

ファイル入出力と同じような方法で 2 つのプロセスが直接に通信することができます。

パイプとは、read 関数、 write 関数などのファイル入出力関数を使って読み書きできる仮想的なファイルのようなものです。正確な大きさはシステムによって変わりますが、ファイルサイズには制限があります。パイプとは待ち行列のように先に書き込まれたデータが先に読み出され、一度読んだデータは取り除かれます。

パイプは 2 つの記述子を持っており、この 2 つの記述子がパイプの両端に対応しています。パイプの片方に対して書き込んだデータを他方から読み出すことができます。パイプは次の関数によって生成されます。

```
# Unix.pipe ;;
- : unit -> Unix.file_descr * Unix.file_descr = <fun>
```

最初の記述子に対して書き込まれた内容は 2 番目の記述子から読み出され、2 番目の記述子に対して書き込まれた内容は最初の記述子から読み出されます。これらの記述子はどのプロセスからでも閉じることができます。

記述子が閉じられていない限り、つまり書き込まれる可能性がある限り、記述子からの読み出しはブロックする可能性があります。記述子が閉じられている場合 read 関数は直ちに 0 を返します。データが完全に詰まっているパイプに書き込もうとした場合、その書き込みは他のプロセスがパイプからデータを読み出すまでブロックします。読み込みの記述子が、その記述子を持つすべてのプロセスによって閉じられているパイプに書き込みを行った場合、sigpipe シグナルが発生します。通常このシグナルが発生するとプロセスは終了します。

以下の例ではパイプを使って孫プロセスが自分のプロセス番号を祖父プロセスに伝えています。

```
let output, input = Unix.pipe();;

let write_pid input =
  try
    let m = "(" ^ (string_of_int (Unix.getpid ())) ^ ")"
    in ignore (Unix.write input m 0 (String.length m)) ;
       Unix.close input
   with
     Unix.Unix_error(n,f,arg) →
       Printf.printf "%s(%s) : %s\n" f arg (Unix.error_message n) ;;
```

```
match Unix.fork () with
    0 → for i=0 to 5 do
          match Unix.fork() with
              0 → write_pid input ; exit 0
            | _ → ()
        done ;
        Unix.close input
  | _  → Unix.close input;
         let s = ref "" and buff = String.create 5
         in while true do
              match Unix.read output buff 0 5 with
                  0 → Printf.printf "My grandchildren are %s\n" !s ; exit 0
                | n → s := !s ^ (String.sub buff 0 n) ^ "."
           done ;;
```

このプログラムの実行結果は例えば次のようになります。

```
My grandchildren are (1259.4).(1259.9).(1259.8).(1259.7).(1259.6).(1259.5).
```

パイプにどのようなデータが送られたか分かるように、連続して読んだ文字列の間にピリオドが入っています。これによって読み込みが同期していないことが分かるでしょう。1 度の書き込みに対して読み込みは 2 度行われています。

名前付きパイプ Unix システムの中には、通常のファイルと同じように名前を持ったパイプを作る機能を提供しているシステムがあります。名前付きパイプを使うと、プロセスの親子関係がなくてもプロセス間通信を行えます。次の関数によって名前付きパイプを生成します。

```
# Unix.mkfifo ;;
- : string -> Unix.file_perm -> unit = <fun>
```

パイプにアクセスする記述子は通常のファイルと同じように `openfile` 関数によって入手できます。名前付きファイルは通常のファイルと同じように見えますが、あくまでもパイプであって、例えば `lseek` 関数をパイプに対して使うことはできません。

警告 | `mkfifo` 関数は Windows 環境では実装されていません。

通信チャネル

`Unix` モジュールには、プログラムを起動すると同時にその入出力チャネルを設定することができる高レベル関数があります。

```
# Unix.open_process ;;
- : string -> in_channel * out_channel = <fun>
```

この関数の引数には起動するプログラムの名前、正確にはコマンドラインインタープリタに打ち込むのと同じパスを指定します。起動するプログラムへの引数も同時に記述することができます。帰り値は起動されたプログラムの標準入出力と結びつけられたファイル記述子です。この関数によって起動されたプログラムは、現在のプログラムと並列に実行されます。

警告 open_process によって起動されるプログラムは Unix のコマンドラインインタープリタである /bin/sh によって実行されます。したがって /bin/sh を持たないシステムではこの関数を使えません。

open_process によって起動したプログラムは次の関数によって終了します。

```
# Unix.close_process ;;
- : in_channel * out_channel -> Unix.process_status = <fun>
```

引数には終了するプログラムに結びつけられている記述子を指定します。帰り値には終了したプログラムの実行ステータスが返されます。

入出力チャネルの一方しか必要でない時にはそのための専用の関数が用意してあります。

```
# Unix.open_process_in ;;
- : string -> in_channel = <fun>
# Unix.close_process_in ;;
- : in_channel -> Unix.process_status = <fun>
# Unix.open_process_out ;;
- : string -> out_channel = <fun>
# Unix.close_process_out ;;
- : out_channel -> Unix.process_status = <fun>
```

以下の例では open_process を使って ocaml の中から ocaml を起動しています。

```
# let n_print_string s = print_string s ; print_string "(* <-- *)" ;;
val n_print_string : string -> unit = <fun>
# let p () =
    let oc_in, oc_out = Unix.open_process "/usr/local/bin/ocaml"
    in n_print_string (input_line oc_in) ; print_newline() ;
       n_print_string (input_line oc_in) ; print_newline() ;
       print_char (input_char oc_in) ;
       print_char (input_char oc_in) ;
       flush stdout ;
       let s = input_line stdin
       in output_string oc_out s ;
          output_string oc_out "#quit\n" ;
          flush oc_out ;
          let r = String.create 250 in
          let n = input oc_in r 0 250
          in n_print_string (String.sub r 0 n) ;
             print_string "Thank you for your visit\n" ;
             flush stdout ;
             Unix.close_process (oc_in, oc_out) ;;
val p : unit -> Unix.process_status = <fun>
```

関数 p を呼び出すと Objective Caml のトップレベルが実行を開始します。この例では Objective Caml のバージョン 2.03 が /usr/local/bin にあることを仮定しています。最初に 4 行を読み飛ばしているのはトップレベルのヘッダ部分です。キーボードから let x = 1.2 +. 5.6;; と打ち込まれるとその内容を oc_out (新しく生成されたプロセスの標準入力に結び付いているチャネル) に送っています。その内容は Objective Caml の式として評価され、結果が oc_in に送られます。この結果を input を使って読み込み、画

面に表示します。最後に "Thank you for your visit" と表示しています。新しく呼び出されたトップレベルを終了させるために#*quit*;; という文字列を送っています。

```
# p();;
        Objective Caml version 2.03

# let x = 1.2 +. 5.6;;
val x : float = 6.8
Thank you for your visit
- : Unix.process_status = Unix.WSIGNALED 13
#
```

Unix のシグナル

シグナルとはプロセスと通信する手段の一つです。シグナルには、プログラムが何かを実行している最中であっても受け取ることができるという特徴があります。つまりシグナルはプロセスの実行に割り込むことができます。プロセスがシグナルを受け取るとプロセスの実行が一時中断され、シグナルを処理したあと、割り込まれた地点から実行が再開されます。シグナルの数は少なく（Linux では 32 個)、シグナルを使って渡せる情報も非常に限られています（シグナル番号のみ)。プロセスにはシグナルを受けたときの動作が定義されていますが、ほとんどのシグナルに対しては再定義できます。

シグナルを扱う関数やデータ構造は Sys モジュールと Unix モジュールに分かれて定義されています。Sys モジュールでは POSIX 標準 (文献 [Ste92] 参照) に準拠したシグナルの定義といくつかの関数が定義されています。Unix モジュールではプロセスにシグナルを送る関数 kill が定義されています。Windows 環境では利用できるシグナルは sigint だけになります。

シグナルが発生する原因はいくつもあります。キーボードから送ることができますし、メモリへの不正アクセスからも生じます。プロセスが別のプロセスにシグナルを送るには次の関数を使います。

```
# Unix.kill ;;
- : int -> int -> unit = <fun>
```

この関数の最初の引数はシグナルを受けるプロセスの PID です。2 番目の引数は送りたいシグナルを指定します。

シグナルの処理

シグナルを受けた時にプロセスが行える反応は 3 種類のカテゴリに分類されており、それぞれのカテゴリに対して signal_behavior 型を持つ以下の構築子が定義されています。

- Signal_default: システムによって定義された標準の反応を表しています。多くの場合、プロセスを終了させます。設定によってはプロセスを終了させる時にそのプロセスの状態を記述したファイル (core ファイル) を生成することがあります。
- Signal_ignore: シグナルを無視します。
- Signal_handle: シグナルを受けた時の動作がユーザによって再定義されていることを表しています。この構築子は *int -> unit* 型を持つ関数を引数として持って

おり、その関数がシグナルを受けた時の動作を定義しています。この関数の引数には受けたシグナルの番号が渡されます。

プロセスがシグナルを受けると、そのプロセスの実行はシグナルを処理する関数へと移行します。シグナルを受けた時の動作を変更する関数は Sys モジュールで提供されています。

```
# Sys.set_signal;;
- : int -> Sys.signal_behavior -> unit = <fun>
```

最初の引数は対象となるシグナルの番号です。2 番目の引数は新しく登録する動作です。

Sys モジュールはシグナルの動作を変更する別の関数も提供しています。

```
# Sys.signal ;;
- : int -> Sys.signal_behavior -> Sys.signal_behavior = <fun>
```

この関数は set_signal とほとんど同じですが、再定義する直前の動作を帰り値として返します。この関数を使うと、シグナルに定義された動作を返す関数を簡単に定義できます。

```
# let signal_behavior s =
    let b = Sys.signal s Sys.Signal_default
    in Sys.set_signal s b ; b ;;
val signal_behavior : int -> Sys.signal_behavior = <fun>
# signal_behavior Sys.sigint;;
- : Sys.signal_behavior = Sys.Signal_handle <fun>
```

しかし一部のシグナルの動作をユーザが定義することはできません。そのためこの関数は特定のシグナルに対しては正しく機能しません。

```
# signal_behavior Sys.sigkill ;;
Exception: Sys_error "Invalid argument".
```

シグナルの例

ここではいくつかの重要なシグナルについて具体的に解説します。

sigint　このシグナルは一般に CTRL-C を押すことで発生します。以下の例ではこのシグナルを受けた時の動作を変更し、メッセージを表示するようにしています。ただし 3 回目にシグナルを受けた時にプロセスを終了します。

以下のプログラムを ctrlc.ml というファイルに保存します。

```
let sigint_handle =
 let n = ref 0
 in function _ → incr n ;
                 match !n with
                     1 → print_string "You just pushed CTRL-C\n"
                   | 2 → print_string "You pushed CTRL-C a second time\n"
                   | 3 → print_string "If you insist ...\n" ; exit 1
                   | _ → () ;;
Sys.set_signal Sys.sigint (Sys.Signal_handle sigint_handle) ;;
```

```
match Unix.fork () with
    0 → while true do () done
  | pid → Unix.sleep 1 ; Unix.kill pid Sys.sigint  ;
          Unix.sleep 1 ; Unix.kill pid Sys.sigint  ;
          Unix.sleep 1 ; Unix.kill pid Sys.sigint  ;;
```

このプログラムは CTRL-C を押す動作と全く同じ効果を引き起こします。実行すると以下のような結果が得られます。

```
$ ocamlc -i -o ctrlc ctrlc.ml
val sigint_handle : int -> unit
$ ctrlc
You just pushed CTRL-C
You pushed CTRL-C a second time
If you insist ...
```

sigalrm 良く使われているもう一つのシグナルが sigalrm です。このシグナルはタイマとして利用でき、以下の関数によって設定します。

```
# Unix.alarm ;;
- : int -> int = <fun>
```

引数には sigalrm を送るまでに待つ秒数を指定します。この関数を 2 回以上呼んだ場合には常に最後に指定した秒数だけが有効であり、以前の状態は捨てられます。帰り値には、関数を呼び出す以前に設定されていた残りの秒数が返されます。

タイマを使った簡単な例を作ってみましょう。ある計算が指定した時間内に終了した時はその結果を返し、時間を過ぎてしまったらあらかじめ決められた値を返す関数 timeout を定義します。関数 timeout は関数 f と f の引数 arg と制限時間 (time)、制限を越えた時に返す値 (default_value) を引数として取ります。

具体的には timeout の仕様は以下のようになります。

1. 後で復元できるように sigalrm に定義されていた動作を保存しておきます。
2. sigalrm を受けた時の動作を変更し、Timeout 例外を投げるようにします。
3. タイマを作動させます。
4. この後は条件によって動作が変わります。
   (a) 制限時間内に実行が終了した時は sigalrm の状態を復元し、計算結果を返します。
   (b) 制限時間内に終らなかった時は sigalrm の状態を復元し、既定値を返します。

この仕様をプログラムにすると以下のようになります。

```
# exception Timeout ;;
exception Timeout
# let sigalrm_handler = Sys.Signal_handle (fun _ → raise Timeout) ;;
val sigalrm_handler : Sys.signal_behavior = Sys.Signal_handle <fun>
# let timeout f arg time default_value =
    let old_behavior = Sys.signal Sys.sigalrm sigalrm_handler in
    let reset_sigalrm () = Sys.set_signal Sys.sigalrm old_behavior
```

```
    in ignore (Unix.alarm time) ;
       try  let res = f arg in reset_sigalrm () ; res
       with exc → reset_sigalrm () ;
                   if exc=Timeout then default_value else raise exc ;;
val timeout : ('a -> 'b) -> 'a -> int -> 'b -> 'b = <fun>
# let iterate n = for i = 1 to n do () done ; n ;;
val iterate : int -> int = <fun>
```

以下に実行例を示します。

```
Printf.printf "1st execution : %d\n" (timeout iterate 10 1 (-1));
Printf.printf "2nd execution : %d\n" (timeout iterate 100000000 1 (-1)) ;;
```

```
1st execution : 10
2nd execution : -1
- : unit = ()
```

sigusr1 と sigusr2　この 2 つのシグナルはプログラマのために用意されたものでオペレーティングシステムでは使われていません。

以下の例では sigusr1 を受けると変数 i の値を表示しています。

```
let i = ref 0  ;;
let write_i s = Printf.printf "signal received (%d) -- i=%d\n" s !i ;
                 flush stdout ;;
Sys.set_signal Sys.sigusr1 (Sys.Signal_handle write_i) ;;

match Unix.fork () with
    0 → while true do incr i done
  | pid → Unix.sleep 0  ; Unix.kill pid Sys.sigusr1  ;
          Unix.sleep 3 ; Unix.kill pid Sys.sigusr1  ;
          Unix.sleep 1  ; Unix.kill pid Sys.sigkill
```

このプログラムの実行結果は以下のようになります。

```
signal received (-12) -- i=0
signal received (-12) -- i=187676247
```

実行結果を良く見ると、i を加算し続けているループの実行中に sigusr1 を受け、処理をした後に再びループの実行が続いていることが分かります。

sigchld　このシグナルは子プロセスが終了した時に親プロセスに送られます。このシグナルを使うと子プロセスの終了処理をこれまでよりも精密に実現できます。

1. sigchld シグナルを処理する関数を定義します。この関数はシグナルを受けた時に終了しているすべての子プロセスを処理します。[5]そしてこの関数はすべての子プ

5. シグナルは非同期的に処理されるため 2 つの子プロセスがきわどいタイミングで終了した場合にシグナルが一つしか処理されないことがありえます。

ロセスがなくなると親プロセスを (Unix_error 例外を利用して) 終了させます。親プロセスの実行をブロックさせないために wait の代わりに waitpid を使います。

2. メインプログラムは sigchld に関連するハンドラを再定義した後に 5 つの子プロセスを生成し、それらがすべて終了するまで無限ループ (**while** true) に入ります。

```
let rec sigchld_handle s =
  try  let pid, _ = Unix.waitpid [Unix.WNOHANG] 0
       in if pid <> 0
          then ( Printf.printf "%d is dead and buried at signal %d\n" pid s ;
                 flush stdout ;
                 sigchld_handle s )
  with Unix.Unix_error(_, "waitpid", _) → exit 0 ;;

let i = ref 0
in Sys.set_signal Sys.sigchld (Sys.Signal_handle sigchld_handle) ;
   while true do
     match Unix.fork() with
         0 → let pid = Unix.getpid ()
              in Printf.printf "Creation of %d\n" pid ; flush stdout ;
                 Unix.sleep (Random.int (5+ !i)) ;
                 Printf.printf "Termination of %d\n" pid ; flush stdout ;
                 exit 0
       | _ → incr i ; if !i = 5 then while true do () done
   done ;;
```

このプログラムの実行結果は以下のようになります。

```
Creation of 12579
Creation of 12580
Creation of 12583
Creation of 12582
Creation of 12581
Termination of 12583
12583 is dead and buried at signal -14
Termination of 12579
12579 is dead and buried at signal -14
Termination of 12581
Termination of 12580
12581 is dead and buried at signal -14
12580 is dead and buried at signal -14
Termination of 12582
12582 is dead and buried at signal -14
```

練習問題

この練習問題ではファイル記述子、プロセス、パイプ、シグナルを扱った 3 つの課題があります。最初の 2 つの課題は Unix システムプログラミングから採用したもので Linux 等のディストリビューションに含まれている C のプログラムと Objective Caml のプログラムとを比べてみることができるでしょう。

語数計算: wc

Unix の wc コマンドを再定義してみましょう。wc コマンドはテクストファイルに含まれている文字数、語数、行数を数えて表示するツールです。語とは空白文字、タブ、改行コードなどによって分割された文字列のことです。

1. ファイル名を引数に取り、そのファイルに含まれる文字数、語数、行数を表示するプログラム (wc1) を作りなさい。
2. wc1 を改良して、3 つのオプション (-c, -w, -l) と複数のファイル名を受け付けるプログラム (wc2) を作りなさい。オプションはそれぞれ文字数、語数、行数のどれを表示するのかを指示しています。複数のファイル名が指定された時はそのファイル名が結果の前に表示されます。

パイプを使ったスペルチェック

この課題ではパイプを使って二つのツールを結合させてみましょう。Unix のシェルの機能である | と同じように一方のツールの出力結果を他方のツールの入力として扱わせる機能を実現します。

1. *string * string list -> string * string list -> unit* 型を持つ関数 pipe_two_progs を定義しなさい。この関数は pipe_two_progs（p1,[a1; ...; an]）(p2,[b1; ...; bp]）という引数を与えると p1 a1 ... an と p2 b1 ... bp という 2 つのプログラムを起動させ、p1 の標準出力を p2 の標準入力へとつなぎます。ただし *ai* と *bi* はそれぞれ p1 と p2 の引数です。
2. 114 ページのスペルチェッカを利用して 誤った語を一行に一語ずつ標準出力に表示するプログラムを作りなさい。
3. 標準入力から語の列を受け取り、辞書式順序で並び換えて標準出力に表示するプログラム を作りなさい。この関数を定義する時に Sort.list 関数を使っても構いません。Sort.list 関数は比較関数を与えると並び換えてくれる関数です。
4. pipe_two_progs にスペルチェッカと並び換えプログラムを適用する関数を書きなさい。
5. プログラムのリストを受け取って、それぞれ前後の標準出力と標準入力を結合する関数 pipe_n_progs を書きなさい。
6. リストの中の重複する語を除去するプログラムを書きなさい。
7. これらの 3 つのプログラムを pipe_n_progs に適用する関数 を書きなさい。

計算状態の対話的な取得

複雑な時間のかかる計算をしている間にユーザが計算の進行状況を知ることができれば便利でしょう。指定された区間に含まれる素数を計算する課題（245 ページ）を使ってこの手法を実装してみましょう。

1. 課題のプログラムを書き換えて 大域変数 result が常に最後に計算された素数を保持しているようにしなさい。

2. `sigint` シグナルを受けると `result` の内容を表示する関数 `sigint_handle` を書きなさい。
3. `sigint` のシグナルハンドラを Modify `sigint_handle` に設定しなさい。
4. プログラムをコンパイルし ある程度時間のかかる計算させなさい。計算の途中で `CTRL-C` を押したり、Unix の `kill` コマンドを利用して `sigint` シグナルをプロセスに送りなさい。

まとめ

この章では `Unix` モジュールが提供している関数について説明を行いました。`Unix` モジュールという名前になっていますが、多くの関数 (図 18.1 参照) は Windows 環境でも利用できます。

プロセスの話題に関しては同じマシン上で同時に走っているプログラム間の通信手法について説明しました。ファイル記述子を使った低水準入出力、シグナル、パイプについても同様に詳しく扱いました。

もっと知りたい人へ

`Unix` モジュールは Unix システムライブラリの関数を提供していますが、Unix 自身のプログラミングパラダイムについてはこの書ではほとんど触れていません。システムプログラミングの標準的な参考書として [ス 00] と [CDM98a] を挙げておきます。

また Caml-Light を題材にして Xavier Leroy の書いたすばらしい解説 [Ler92] があります。これは WWW からもアクセスできます。

リンク: http://pauillac.inria.fr/~xleroy/publi/unix-in-caml.ps.gz

`Unix` モジュールの実装は C 言語と Objective Caml の間のインターフェースの記述例として非常によい題材となっています。多くの関数では単純に型情報だけを書き換えて C のシステム関数を直接呼び出しています。`Unix` モジュールのソースコードは Objective Caml のディストリビューションの中の `otherlibs/unix` と `otherlibs/win32unix` のディレクトリの中にあります。

この章では `Unix` モジュールの機能のいくつかを紹介しました。`Unix` モジュールの持つ通信ソケットとそれを利用したインターネット上の通信に関しては第 20 章で扱います。しかし端末やファイルシステムなどの他の機能についてはこの本では扱っていません。それらの機能については上で紹介した Unix の解説書を参照してください。

19

並行プログラミング

並行性 *concurrency* とは個々の出来事の間に因果関係がない状況を表しています。プログラミング言語では機械語の命令が「同時に」実行される状況を意味しています。これは第 4 部の最初で紹介した「並列性」という概念と同じ定義になっています。前章で説明した `Unix` ライブラリのプロセスも、単一のプロセッサ上で同時実行しているように見せかけているという意味で並行性を持っていると言えます。しかしプロセスと並行性の概念は `fork` システムコールによって生成されるものだけに限定されるのではありません。

Objective Caml 言語は軽いプロセス (スレッド) のためのライブラリを持っています。スレッドとプロセスの最大の違いは、同じ一つのプロセスがそれらを生成した時にメモリが共有されるのかどうかという点にあります。同じプロセスが二つのスレッドを生成した場合、それぞれのスレッドには実行コンテクストのみが新しく割り当てられ、コードセクションとメモリセクションは共有されます。スレッドは必ずしもアプリケーションの実行時間を改善しません。スレッドが良く使われるのは並行アルゴリズムを自然にプログラミングできるようになるという大きな利点があるからです。

命令型、関数型などの言語の性質も並行性のモデルに影響します。命令型のプログラムではスレッドは共有メモリに値を書き込むことができます。プロセス間通信はメモリへの値の書き込みと読み出しによって実現することができます。しかし純粋な関数型プログラムでは副作用を持っていないため、たとえメモリが共有されていたとしてもスレッドは別のスレッドのメモリに影響を与えることができません。この場合実質的に分散メモリモデルを採用しているのと同じであり、プロセス間通信はチャネルを使って行わねばなりません。

Objective Caml 言語のスレッドライブラリは両方のモデルを実装しています。`Thread` モジュールを使って、ある関数呼び出しを新しいプロセスとして実行させることができます。`Mutex` モジュールと `Condition` モジュールは相互排他や条件待機のような同期のための機構を提供しています。`Event` モジュールはイベントを使った値の通信手段を実装しています。送る値にはもちろん関数も使うことができ、スレッド間で計算のやりとりを行うことができます。Objective Caml 言語の常として両方のモデルを混在させることが可能です。

このライブラリは Objective Caml が動作するシステム上で互換性があります。`Unix` モジュールとは異なり、`Thread` ライブラリは OS のプロセスを直接利用しています。

この章のあらまし

第一節ではスレッドが他のスレッドに対してどのように影響を与えることができるかについて説明し、そのあと `Thread` モジュールを解説します。一つのアプリケーションで多くのプロセスを実行させる方法についても示します。

第二節は相互排他 (`Mutex` モジュール) と条件待機 (`Condition` モジュール) によるスレッド間の同期の仕方について扱います。これらのモジュールが持つ本質的なやっかいさを示す二つの完全なプログラムを含んでいます。

第三節では `Event` モジュールによって提供されているイベントを利用した通信手段について説明します。この通信手段によって新しい形態の相互作用を記述することができます。

第四節では、郵便局の窓口のための共有待ち行列の実装を示し、この章をまとめます。

並行プロセス

並行プロセスを組み合せてアプリケーションを作成した場合、逐次言語が持っていた決定性という便利な性質を失うことになります。メモリの同じ領域を共有しているプロセスの実行結果はプログラムの文面から一意に決めることはできません。

メインプログラム	
let *x* = *ref* 1;;	
プロセス P	プロセス Q
x := !*x* + 1;;	*x* := !*x* * 2;;

P と Q の実行の後で `x` の指す値は、プロセスの実行順序によって 2, 3, 4 の三通りがありえます。

非決定的なのはアプリケーションの終了についても当てはまります。メモリの状態が個々のプロセスの実行によって変わる状況では、ある特定の実行の時にアプリケーションが停止し、別の実行の時には動き続けることがありえます。実行を制御するためにプロセスの同期を利用することができます。

それぞれのプロセスが独立したメモリ領域を使用している場合ではアプリケーションの実行結果は通信の形態に影響を受けます。以下の例では、値を送る `send` と値を受け取る `receive` という二つのプリミティブを使っています。P と Q はお互いに通信する二つのプロセスです。

プロセス P	プロセス Q
let *x* = *ref* 1;;	**let** *y* = *ref* 1;;
send(*Q*,!*x*);	*y* := !*y* + 3;
x := !*x* * 2;	*y* := !*y* + *receive*(*P*);
send(*Q*,!*x*);	*send*(*P*,!*y*);
x := !*x* + *receive*(*Q*);	*y* := !*y* + *receive*(*P*);

通信媒体がバッファを持っていない時はプロセス Q は P からのメッセージを受け取れない場合がありえます。したがってこのシステム全体の動作は非決定的です。

通信媒体がメッセージを順番に保存する機能があり、非同期的である時はメッセージの受信でのみブロックします。プロセス Q が P からの二つのメッセージを受信する前であってもプロセス P が `receive` でブロックする場合は起こり得ます。しかし、そうなってもメッセージは通信媒体に保存されているのでプロセス Q は正しい値を受け取ります。この場合はシステム全体の動作は決定的です。

スレッドを使ったプログラムのコンパイル

Objective Caml スレッドライブラリは 5 つのモジュールから構成されています。その内 4 つのモジュールでは抽象データ型を定義しています。

- `Thread` モジュール: スレッド (*Thread.t* 型) の生成と実行。
- `Mutex` モジュール: mutex (*Mutex.t* 型) の生成、ロック、解放。
- `Condition` モジュール: 条件 (シグナル、*Condition.t* 型) の生成、待機、覚醒。
- `Event` モジュール: 通信チャネル (*'a Event.channel* 型) の生成。伝達される値 (*'a Event.event* 型)。通信を行う関数群。
- `ThreadUnix` モジュール: `Unix` モジュールの I/O 関数をブロックしないように再定義したもの。

このライブラリは Objective Caml の標準実行ライブラリには含まれていません。したがってこのライブラリを使ったプログラムをコンパイル時や新しいトップレベルを定義するためには特別な指定を行う必要があります。

```
$ ocamlc -thread -custom threads.cma  files.ml -cclib -lthreads
$ ocamlmktop -tread -custom -o threadtop thread.cma -cclib -lthreads
```

スレッドライブラリはプラットフォームが POSIX 1003[1]準拠のスレッドを実装していない限り、ネイティブコンパイラから使うことはできません。コンパイルする時は `unix.a` ライブラリと `pthread.a` ライブラリを加える必要があります。

```
$ ocamlc -thread -custom threads.cma files.ml -cclib -lthreads \
```

1. この場合 Objective Caml コンパイラは付属のスレッドライブラリではなくプラットフォームが提供しているものを使うように構成されていなければなりません。

```
  -cclib -lunix -cclib -lpthread
$ ocamltop -thread -custom threads.cma files.ml -cclib -lthreads \
  -cclib -lunix -cclib -lpthread
$ ocamlcopt -thread threads.cmxa files.ml -cclib -lthreads \
  -cclib -lunix -cclib -lpthread
```

Thread モジュール

Objective Caml の Thread モジュールはスレッドを生成、管理するための関数を提供しています。この章ではそれらのすべてを説明しませんが、ファイル I/O などのいくつかの操作は前章で紹介されています。

スレッドは次の関数を使って生成されます。

```
# Thread.create ;;
Characters 1-14:
  Thread.create ;;
  ^^^^^^^^^^^^^
Unbound value Thread.create
```

'a -> 'b 型の第一番目の引数には生成されたプロセスが実行する関数を指定します。'a 型の第二番目の引数には最初の関数に渡す引数を指定します。この関数は生成されたプロセスに結びつけられた記述子を返します。引数として渡した関数の実行が終了するとプロセスは自動的に終了します。

プロセスの実行を開始し、終了を待つには記述子を引数として join を呼び出します。

```
# let f_proc1 () = for i=0 to 10 do Printf.printf "(%d)" i; flush stdout done;
                   print_newline() ;;
val f_proc1 : unit -> unit = <fun>
# let t1 = Thread.create f_proc1 () ;;
Characters 9-22:
  let t1 = Thread.create f_proc1 () ;;
           ^^^^^^^^^^^^^
Unbound value Thread.create
# Thread.join t1 ;;
Characters 0-11:
  Thread.join t1 ;;
  ^^^^^^^^^^^
Unbound value Thread.join
```

警告 プロセスの実行結果を親プロセスが直接知る方法はありません。

kill 関数によってプロセスの実行を中止させることができます。以下にプロセスを生成後、すぐにそのプロセスの実行を中止させるプログラムを示します。

```
# let n = ref 0 ;;
val n : int ref = {contents = 0}
# let f_proc1 () = while true do incr n done ;;
val f_proc1 : unit -> unit = <fun>
# let go () = n := 0 ;
              let t1 = Thread.create f_proc1 ()
```

```
              in Thread.kill t1 ;
                 Printf.printf "n = %d\n" !n ;;
Characters 42-55:
              let t1 = Thread.create f_proc1 ()
                       ^^^^^^^^^^^^^
Unbound value Thread.create
# go () ;;
Characters 0-2:
  go () ;;
  ^^
Unbound value go
```

プロセスは自分自身の実行を終了させることができます。

```
# Thread.exit ;;
Characters 1-12:
  Thread.exit ;;
  ^^^^^^^^^^^
Unbound value Thread.exit
```

プロセスは指定された時間だけ実行を中断させることができます。

```
# Thread.delay ;;
Characters 1-13:
  Thread.delay ;;
  ^^^^^^^^^^^^
Unbound value Thread.delay
```

delay の引数は中断している秒数を意味しています。

直前の例を利用して、f_proc1 を新しいプロセスとして生成後、d 秒後に実行を終了させる関数 f_proc2 を定義してみます。f_proc2 の終了後、n の値を表示します。

```
# let f_proc2 d =
    n := 0 ;
    let t2 = Thread.create f_proc1 ()
    in Thread.delay d ;
       Thread.kill t2 ;;
Characters 41-54:
    let t2 = Thread.create f_proc1 ()
             ^^^^^^^^^^^^^
Unbound value Thread.create
# let t1 = Thread.create f_proc2 0.25
   in Thread.join t1 ; Printf.printf "n = %d\n" !n ;;
Characters 9-22:
  let t1 = Thread.create f_proc2 0.25
           ^^^^^^^^^^^^^
Unbound value Thread.create
```

プロセスの同期

メモリの同じ領域を共有しているプロセスがあると「並行性」という言葉の意味が重くなっています。それぞれのプロセスがメモリという資源[2]へのアクセスに関して競合状態になっています。このような状態ではメモリの更新やプロセスの使用時間に関してきちんと管理するのが難しい状況になっています。

複数のプロセスを管理するシステムでは、計算中の演算の任意の部分でプロセスの実行が中断される可能性があります。したがっていくつかのプロセスが一つの資源を共有している時、共有されているデータの整合性をきちんと保証することができなければなりません。このためにはプロセスがある一定の演算 (例えば周辺機器から一続きのデータを受け取る) を終了するまではその資源を占有する仕組みが必要になります。一つのプロセスがあるデータに対して排他的にアクセスできることを保証する仕組みを**相互排除**と呼びます。

クリティカルセクションと相互排除

相互排除の機構は **mutex** と呼ばれる特別なデータ構造を利用して実装されています。**mutex** に対して行える演算は生成、獲得、解放のみです。mutex は並行プロセスによって共有される最小のデータです。mutex への操作は常に排他的です。mutex を獲得できるのは常に一つのプロセスのみであり、もし他のプロセスが既に獲得された mutex を獲得しようとすればその操作は mutex が解放されるまでブロックすることになります。

`Mutex` モジュール

`Mutex` モジュールは相互排除に関連するプロセスのためにメモリ上に mutex を生成します。mutex の使い方を並行性の古典的な例題を使って具体的に説明します。

mutex の生成、ロックの獲得、ロックの解放を行う関数はそれぞれ次のようになります。

```
# Mutex.create ;;
Characters 1-13:
  Mutex.create ;;
  ^^^^^^^^^^^^
Unbound value Mutex.create
# Mutex.lock ;;
Characters 0-10:
  Mutex.lock ;;
  ^^^^^^^^^^
Unbound value Mutex.lock
# Mutex.unlock ;;
Characters 0-12:
  Mutex.unlock ;;
  ^^^^^^^^^^^^
```

2. 一般的にはメモリに限らず I/O 機器などの資源に対しても競合は発生する。

```
Unbound value Mutex.unlock
```

ブロックせずにロックを獲得する関数もあります。

```
# Mutex.try_lock;;
Characters 1-15:
  Mutex.try_lock;;
  ^^^^^^^^^^^^^^
Unbound value Mutex.try_lock
```

既に mutex がロックされている状態であればこの関数は **false** を返します。そうでなければ mutex はロックされ関数は **true** を返します。

食事をする哲学者

ダイクストラ *Dijkstra* によって考え出された「食事をする哲学者の問題」は並行計算での資源管理の問題を純粋に捉えています。

> 5 人の東洋の哲学者は思索に耽るか食事をするかどちらかしかしない。そして思索と食事は同時にできない。彼らは丸いテーブルに座っており、テーブルの中央にはは並々と皿に盛られた料理がある。哲学者の目の前には取り皿が 1 枚と箸が 1 本ずつ両脇に置かれている。部屋には 5 人の哲学者、6 枚の皿、5 本の箸がある。

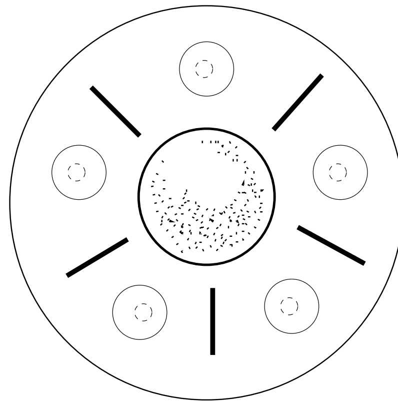


図 19.1: 食事をする哲学者のテーブル

図 19.1 から分かるように哲学者は食事をする時に自分の両脇にある 2 本の箸を取ろうとします。食事が終れば箸を置き、思索に入ります。ある哲学者が食事をしているとその両隣の哲学者は食べることができません。両隣の哲学者は箸が 2 本確保できるまで待たなければなりません。ここでは箸が資源に対応しています。

単純化して哲学者は常にテーブルの同じ席に付くことにします。5 本の箸を配列 `b` に格納された mutex によってモデル化します。

```
# let b =
```

```
   let b0 = Array.create 5 (Mutex.create()) in
    for i=1 to 4 do b0.(i) <- Mutex.create() done;
    b0 ;;
Characters 35-47:
   let b0 = Array.create 5 (Mutex.create()) in
                            ^^^^^^^^^^^^
Unbound value Mutex.create
```

食事と思索は単純にプロセスの中断によって表現します。

```
# let meditation = Thread.delay
  and eating = Thread.delay ;;
Characters 18-30:
  let meditation = Thread.delay
                   ^^^^^^^^^^^^
Unbound value Thread.delay
```

哲学者はダイクストラの話に出て来る行為を無限に繰り返す関数によってモデル化します。箸を取る行為は mutex の獲得として表します。したがって哲学者は同時に 1 本の箸しか確保できません。箸を取る行為と置く行為には短い時間がかかることにします。哲学者が行った行為は画面に出力することにします。

```
# let philosopher i =
    let ii = (i+1) mod 5
    in while true do
         meditation 3. ;
         Mutex.lock b.(i);
         Printf.printf "Philosopher (%d) takes his left-hand chopstick" i ;
         Printf.printf " and meditates a little while more\n";
         meditation 0.2;
         Mutex.lock b.(ii);
         Printf.printf "Philosopher (%d) takes his right-hand chopstick\n" i;
         eating 0.5;
         Mutex.unlock b.(i);
         Printf.printf "Philosopher (%d) puts down his left-hand chopstick" i;
         Printf.printf " and goes back to meditating\n";
         meditation 0.15;
         Mutex.unlock b.(ii);
         Printf.printf "Philosopher (%d) puts down his right-hand chopstick\n" i
       done ;;
Characters 70-80:
         meditation 3. ;
         ^^^^^^^^^^
Unbound value meditation
```

次のコードによってシミュレーションを行うことができます。

```
for i=0 to 4 do ignore (Thread.create philosopher i) done ;
while true do Thread.delay 5. done ;;
```

哲学者を表すプロセスの実行時間を多く確保するためにメインループ (**while**) は定期的に中断しています。シミュレーションが規則性を持たないように乱数で決まる時間だけ中断するようにしてもよいでしょう。

単純なアルゴリズムの問題点　我々の哲学者には恐ろしい状況が起きてしまいます。哲学者全員が左手に箸を持ち、そのまま何もできなくなってしまいます。このような状況を**デッドロック**と呼びます。哲学者はものを食べられなくなってしまうので**スタベーション** (飢餓状態) とも言います。

これを避けるためにもし 2 本目の箸をつかめなかった時は最初の箸を置くようにすることもできます。これは非常によい解決法に見えますが已然として両隣の哲学者が常に箸を取り、特定の哲学者が食事にありつけない状況は起こり得ます。この問題には数多くの解決法がありますが、そのうちの一つが 622 ページにしめしたオブジェクトによる実装です。

生産者消費者 *1*

生産者消費者 *producers-consumers* の組み合せは並行プログラミングの古典的な例です。生産者と呼ばれるプロセスは待ち行列にデータを格納し続けます。一方消費者と呼ばれるプロセスは待ち行列からデータを取り出して処理をし続けます。

このスキームを実装してみましょう。システムを正しく動作させるために、生産者と消費者に共有される待ち行列に対する操作は排他的でなければなりません。

f が共有される待ち行列であり m が mutex です。

```
# let f = Queue.create ()  and  m = Mutex.create () ;;
Characters 35-47:
  let f = Queue.create ()  and  m = Mutex.create () ;;
                                    ^^^^^^^^^^^^
Unbound value Mutex.create
```

生産者の活動を商品の生産 (produce 関数) と商品の格納 (store 関数) の二つに分割します。格納の操作に対してのみ mutex を必要とします。

```
# let produce i p d =
    incr p ;
    Thread.delay d ;
    Printf.printf "Producer (%d) has produced %d\n" i !p ;
    flush stdout ;;
Characters 35-47:
    Thread.delay d ;
    ^^^^^^^^^^^^
Unbound value Thread.delay

# let store i p =
    Mutex.lock m ;
    Queue.add (i,!p) f ;
    Printf.printf "Producer (%d) has added its %dth product\n" i !p ;
    flush stdout ;
    Mutex.unlock m ;;
```

```
Characters 19-29:
    Mutex.lock m ;
    ^^^^^^^^^^
Unbound value Mutex.lock
```

生産者のコードは生産と格納の無限の繰り返しです。シミュレーションが規則性を持たないように乱数時間だけ待ち時間を作ります。

```
# let producer i =
    let p = ref 0 and d = Random.float 2.
    in while true do
         produce i p d ;
         store i p ;
         Thread.delay (Random.float 2.5)
       done ;;
Characters 85-92:
         produce i p d ;
         ^^^^^^^
Unbound value produce
```

消費者側は待ち行列から要素を取り出すだけです。ただし商品の内容を確認します。

```
# let consumer i =
    while true do
      Mutex.lock m ;
      ( try
          let ip, p = Queue.take f
          in Printf.printf "The consumer(%d) " i ;
             Printf.printf "has taken product (%d,%d)\n" ip p ;
             flush stdout ;
        with
          Queue.Empty →
             Printf.printf "The consumer(%d) " i ;
             print_string "has returned empty-handed\n" ) ;
      Mutex.unlock m ;
      Thread.delay (Random.float 2.5)
    done ;;
Characters 38-48:
      Mutex.lock m ;
      ^^^^^^^^^^
Unbound value Mutex.lock
```

以下のテストプログラムでは 4 人の生産者と 4 人の消費者のシステムを作ります。

```
for i = 0 to 3 do
 ignore (Thread.create producer i);
 ignore (Thread.create consumer i)
done ;
while true do Thread.delay 5. done ;;
```

待機と同期

相互排除の関係はプロセスの同期を実現する手法としてはあまり良いものではありません。あるプロセスの動作が終り、何らかの条件が成立し、別のプロセスの実行へと引き継がれることは稀ではありません。したがってプロセスの待機条件が変化した時にその事実を待機中のプロセスへと伝達する機構があれば望ましいでしょう。

前の例では消費者は待ち行列が空であるときは生産者が製品を格納するまで待機することもできました。条件付きの mutex を**セマフォア** *semaphore* と呼びます。

セマフォア　セマフォアとは非負整数の値を取る変数 s として表現されます。s が初期化された後は s に対して $wait(s)$ または $signal(s)$ のどちらかの操作しか行うことができません。これらの操作はそれぞれ $P(s)$ または $V(s)$ と書くこともあります。s の値は利用できる資源の数に対応しています。

- $wait(s)$: もし $s > 0$ ならば s の値から 1 を引きます。そうでない時は $wait(s)$ を呼んだプロセスは中断する。
- $signal(s)$: もし $wait(s)$ を呼んで中断しているプロセスがあればそのプロセスを再開させる。そうでない時は s の値に 1 を加えます。

0 と 1 の値しか取らないセマフォアを特別にバイナリセマフォア *binary semaphore* と呼びます。

Condition モジュール

Condition モジュールの関数群はプロセスをシグナルに応じて停止、再開させる機能を実装しています。ここでのシグナルとはプロセスに共有される抽象データ型であり、以下の関数によって操作されます。

create : *unit -> Condition.t* シグナルを生成する。

signal : *Condition.t -> unit* シグナルで待っているプロセスを再開させる。

broadcast : *Condition.t -> unit* シグナルで待っているすべてのプロセスを再開させる。

wait : *Condition.t -> Mutex.t -> unit* 第 1 引数で渡されたシグナルでプロセスを待機させる。第 2 引数にはシグナルに関する操作を保護する mutex を渡す。この mutex はプロセスが待機状態になると解放され、待機が解けた時に確保されます。

生産者消費者 2

生産者消費者問題を条件変数を使って待ち行列が空の時は消費者が待機するように書き換えてみましょう。

まず条件変数を宣言します。

```
# let c = Condition.create () ;;
```

```
Characters 9-25:
  let c = Condition.create () ;;
          ^^^^^^^^^^^^^^^^
Unbound value Condition.create
```

生産者の格納関数にシグナルを送る部分を付け加えます。

```
# let store2 i p =
     Mutex.lock m ;
     Queue.add (i,!p) f ;
     Printf.printf "Producer (%d) has added its %dth product\n" i !p ;
     flush stdout ;
     Condition.signal c ;
     Mutex.unlock m ;;
Characters 20-30:
    Mutex.lock m ;
    ^^^^^^^^^^
Unbound value Mutex.lock
# let producer2 i =
     let p = ref 0 in
     let d = Random.float 2.
     in while true do
          produce i p d;
          store2 i p;
          Thread.delay (Random.float 2.5)
        done ;;
Characters 90-97:
         produce i p d;
         ^^^^^^^
Unbound value produce
```

消費者の行動は 2 段階になります。商品が待ち行列に加えられるのを待ち、そのあと取り出します。待ち時間が終了した時に mutex を確保し、商品を取り出した後に mutex を解放します。プロセスは条件変数 c に対して待機します。

```
# let wait2 i =
     Mutex.lock m ;
     while Queue.length f = 0 do
       Printf.printf "Consumer (%d) is waiting\n" i ;
       Condition.wait c m
     done ;;
Characters 17-27:
    Mutex.lock m ;
    ^^^^^^^^^^
Unbound value Mutex.lock
# let take2 i =
     let ip, p = Queue.take f in
       Printf.printf "Consumer (%d) " i ;
       Printf.printf "takes product (%d, %d)\n" ip p ;
       flush stdout ;
       Mutex.unlock m ;;
Characters 39-40:
```

```
    let ip, p = Queue.take f in
                              ^
Unbound value f
# let consumer2 i =
     while true do
       wait2 i;
       take2 i;
       Thread.delay (Random.float 2.5)
    done ;;
Characters 38-43:
      wait2 i;
      ^^^^^
Unbound value wait2
```

実は消費者が待ち行列に商品があるかどうかをチェックする部分は不要です。なぜなら待機状態の最後で mutex のロックを確保するため他の消費者が商品を取り出すことはありえないからです。

読み手と書き手

並行プロセスの古典的問題で共有データに対してプロセスが対称的でない例を見てみましょう。

一人の書き手と複数の読み手が同じ共有データを扱っています。買い手の動作はデータを変更するかもしれません。しかし読み手側はデータを単に読むだけです。したがって読み手は同時にアクセスしてもデータに矛盾は生じません。これを実現する一つの方法はアクセス中の読み手の数を表すカウンタを用意し、このカウンタが 0 の時のみ書き手がデータを変更するのを許すというものです。

データは極限まで単純化して `data` という整数変数で表します。この変数は `0` または `1` の値を取り、`0` の時は読み出し可能であることを表しています。

```
# let data = ref 0 ;;
val data : int ref = {contents = 0}
```

カウンタ `n` への操作は mutex `m` によって保護します。

```
# let n = ref 0 ;;
val n : int ref = {contents = 0}
# let m = Mutex.create () ;;
Characters 8-20:
  let m = Mutex.create () ;;
          ^^^^^^^^^^^^
Unbound value Mutex.create
# let cpt_incr () = Mutex.lock m ; incr n ; Mutex.unlock m ;;
Characters 18-28:
  let cpt_incr () = Mutex.lock m ; incr n ; Mutex.unlock m ;;
                    ^^^^^^^^^^
Unbound value Mutex.lock
# let cpt_decr () = Mutex.lock m ; decr n ; Mutex.unlock m ;;
Characters 18-28:
```

```
  let cpt_decr () = Mutex.lock m ; decr n ; Mutex.unlock m ;;
                    ^^^^^^^^^^
Unbound value Mutex.lock
# let cpt_signal () = Mutex.lock m ;
                      if !n=0 then Condition.signal c ;
                      Mutex.unlock m ;;
Characters 20-30:
  let cpt_signal () = Mutex.lock m ;
                      ^^^^^^^^^^
Unbound value Mutex.lock
```

読み手はカウンタを更新し、もし読み手がいなくなったら最後の読み手が c にシグナルを送ります。これにより書き手が動作を開始します。

```
# let c = Condition.create () ;;
Characters 9-25:
  let c = Condition.create () ;;
          ^^^^^^^^^^^^^^^^
Unbound value Condition.create
# let read i =
    cpt_incr () ;
    Printf.printf "Reader (%d) read (data=%d)\n" i !data ;
    Thread.delay (Random.float 1.5) ;
    Printf.printf "Reader (%d) has finished reading\n" i ;
    cpt_decr () ;
    cpt_signal () ;;
Characters 16-24:
    cpt_incr () ;
    ^^^^^^^^
Unbound value cpt_incr

# let reader i = while true do read i ; Thread.delay (Random.float 1.5) done ;;
Characters 30-34:
  let reader i = while true do read i ; Thread.delay (Random.float 1.5) done ;;
                               ^^^^
Unbound value read
```

書き手は読み手が共有データにアクセスしないようにロックを取る必要がありますがそれはカウンタが 0 の場合だけです。それ以外の時は条件が成立するまで待機します。

```
# let write () =
    Mutex.lock m ;
    while !n<>0 do Condition.wait c m done ;
    print_string "The writer is writing\n" ; flush stdout ;
    data := 1 ; Thread.delay (Random.float 1.) ; data := 0 ;
    Mutex.unlock m ;;
Characters 19-29:
    Mutex.lock m ;
    ^^^^^^^^^^
Unbound value Mutex.lock
```

```
# let writer () =
     while true do write () ; Thread.delay (Random.float 1.5) done ;;
Characters 34-39:
    while true do write () ; Thread.delay (Random.float 1.5) done ;;
                  ^^^^^
Unbound value write
```

1 人の書き手と 6 人の読み手で実験をしてみましょう。

```
ignore (Thread.create writer ());
for i=0 to 5 do ignore(Thread.create reader i) done;
while true do Thread.delay 5. done ;;
```

このアルゴリズムでは書き手と読み手がデータに同時にアクセスしないことは保証されています。しかし書き手がデータを書き込むことは保証されていません。ここでまたスタベーションの問題に直面しています。

同期通信

スレッドライブラリの Event モジュールは二つのプロセスの間で特定の「通信チャネル」を利用した値の転送を実装しています。値を転送する時に送信と受信が同期しています。

イベントを使った同期通信のモデルは通常の値に加えて関数クロージャ、オブジェクトそしてイベント自身も扱っています。

なお同期通信については文献 [Rep99] にも詳しく書かれています。

通信イベントを使った同期

通信イベントのプリミティブとして以下のものがあります。

- `send c v` チャネル c に値 v を送ります。
- `receive c` チャネル c から値を受け取ります。

実装上の都合から値を受け渡す時に送信と受信は同期していなければなりません。このためには同期という操作 (sync) を必要とします。値の受け渡しは送信プロセスと受信プロセスの両方が明示的に同期を行わなければなりません。片方のプロセスのみが同期を行った場合そのプロセスは他方のプロセスが同期を行うまでブロックします。つまり送信側 (*sync (send c v)*) は受信側が (*sync (receive c)*) を実行するまでブロックします。

転送される値

値を交換するのに使う通信チャネルは型が付いています。別の型の値を送るには新しい通信チャネルを作るのがよいでしょう。Objective Caml のスレッドの間で通信する限り、

任意の値を転送することができます。これは関数クロージャやオブジェクト、イベント自身も含んでいます。イベントを同期通信で送ることは間接的な同期要求とも考えられます。

Event モジュール

値を受け渡すのに使う通信チャネルには *'a channel* という抽象データ型が割り当てられています。通信チャネルは次の関数により生成されます。

```
# Event.new_channel ;;
Characters 1-18:
  Event.new_channel ;;
  ^^^^^^^^^^^^^^^^^
Unbound value Event.new_channel
```

送信イベントと受信イベントは次の関数によって生成されます。

```
# Event.send ;;
Characters 1-11:
  Event.send ;;
  ^^^^^^^^^^
Unbound value Event.send
# Event.receive ;;
Characters 0-13:
  Event.receive ;;
  ^^^^^^^^^^^^^
Unbound value Event.receive
```

この send と receive という関数は抽象型 *'a event* の構築子であるとも見ることができます。send によって作られたイベントは送られる値の型の情報を保存していません (*unit Event.event*) が receive によって作られたイベントは同期の後に受け取る値の型情報を持っています。どちらの関数ともブロックしません。値が実際に転送されるのはこれらの関数を呼び出した時ではなく、イベントの値を同期した時だからです。

```
# Event.sync ;;
Characters 1-11:
  Event.sync ;;
  ^^^^^^^^^^
Unbound value Event.sync
```

この関数は送り手と受け手ともブロックする可能性があります。

ブロックしないバージョンの関数もあります。

```
# Event.poll ;;
Characters 1-11:
  Event.poll ;;
  ^^^^^^^^^^
Unbound value Event.poll
```

この関数は通信の相手が同期のために待機しているかどうか確認します。

もし待機している時は通信を行い Some v という値を返します。ただし v が通信イベントに関連している値です。相手が待機状態ではない時は None を返します。sync によって取り出される値はより複雑な計算の結果でも構いませんし、別のメッセージ通信の結果であっても構いません。

同期の例　この例では3つのスレッドを使います。最初のスレッド t1 はチャネル c に対して文字列を送ります。チャネル c はすべてのプロセスで共有されています。スレッド t1 は関数 g のコードを実行します。他のスレッド t2 と t3 は同じチャネルで値が来るのを待っています。

```
# let c = Event.new_channel ();;
Characters 9-26:
  let c = Event.new_channel ();;
          ^^^^^^^^^^^^^^^^^
Unbound value Event.new_channel
# let f () =
     let ids = string_of_int (Thread.id (Thread.self ()))
     in print_string ("-------- before  -------" ^ ids) ; print_newline() ;
        let e = Event.receive c
        in print_string ("-------- during  -------" ^ ids) ; print_newline() ;
           let v = Event.sync e
           in print_string (v ^ " " ^ ids ^ " ") ;
              print_string ("-------- after  -------" ^ ids) ; print_newline() ;;
Characters 38-47:
    let ids = string_of_int (Thread.id (Thread.self ()))
                             ^^^^^^^^^
Unbound value Thread.id
# let g () =
     let ids = string_of_int (Thread.id (Thread.self ()))
     in print_string ("Start of " ^ ids ^ "\n");
        let e2 = Event.send c "hello"
        in Event.sync e2 ;
           print_string ("End of " ^ ids) ;
           print_newline () ;;
Characters 38-47:
    let ids = string_of_int (Thread.id (Thread.self ()))
                             ^^^^^^^^^
Unbound value Thread.id
```

3つのプロセスは次のように生成され、実行されます。

```
# let t1,t2,t3 = Thread.create g (), Thread.create f (), Thread.create f ();;
Characters 16-29:
  let t1,t2,t3 = Thread.create g (), Thread.create f (), Thread.create f ();;
                 ^^^^^^^^^^^^^
Unbound value Thread.create
# Thread.delay 1.0;;
Characters 0-12:
  Thread.delay 1.0;;
```

```
  ^^^^^^^^^^^^
Unbound value Thread.delay
```

通信はブロックする可能性があります。t1 の履歴は t2 と t3 の同期の後に表示されています。t2 と t3 はどちらか一方のみが通信に成功し、終了します。

```
# Thread.kill t2;;
Characters 1-12:
  Thread.kill t2;;
  ^^^^^^^^^^^
Unbound value Thread.kill
# Thread.kill t3;;
Characters 0-11:
  Thread.kill t3;;
  ^^^^^^^^^^^
Unbound value Thread.kill
```

例題: 郵便局

この章の終りに向けて並行プログラムの少し複雑な例を見て行きましょう。この例題は郵便局のカウンタの待ち行列をモデル化しています。

他のプログラムの例題と同じように問題を数学的に単純化して捉え、純粋に計算の問題として考えていきます。ここでは次のようなサービスのモデル化を行います。客は郵便局に付くと整理券を受け取り、待ちます。職員は接客が終ると次の客の番号を呼びます。自分の番号が呼ばれた客は対応するカウンタへと向かいます。

問題の定式化　まず資源と主体を区別して考えます。資源とは「整理券発行器」「番号アナウンサ」「窓口」です。主体とは「客」と「職員」です。資源は自分を相互排除の機構によって管理するオブジェクトによって表します。主体はスレッドによって実行される関数によって表します。主体がオブジェクトの状態を変更したり、状態について知りたくなったときに直接 mutex を管理することはありません。これによって定式化が簡単になり、主体のコードは問題の本質を記述しやすくなります。

資源と主体の実装

整理券発行器　整理券発行器はカウンタと mutex の二つのフィールドを持っています。メソッドは整理券を引く操作を実装したものだけです。

```
# class dispenser () =
    object
      val mutable n = 0
      val m = Mutex.create()
      method take () = let r = Mutex.lock m ; n <- n+1 ; n
                       in Mutex.unlock m ; r
```

```
    end ;;
Characters 65-77:
      val m = Mutex.create()
              ^^^^^^^^^^^^
Unbound value Mutex.create
```

mutex は二人以上の客が同時に整理券を引くのを防ぎます。中間変数 (r) を使ってクリティカルセクションの中で計算された値がメソッドの帰り値として返されるようになっています。

番号アナウンサ　番号アナウンサは呼ばれている客の番号を表す整数と mutex と条件変数の 3 つのフィールドを持っています。メソッドは自分の番号まで待つメソッド (wait) と次の客を呼び出すメソッド (call) です。

```
# class announcer () =
    object
      val mutable nclient = 0
      val m = Mutex.create()
      val c = Condition.create()

      method wait n =
        Mutex.lock m;
        while n > nclient do Condition.wait c m done;
        Mutex.unlock m;

      method call () =
        let r = Mutex.lock m ;
                nclient <- nclient+1 ;
                nclient
        in Condition.broadcast c ;
           Mutex.unlock m ;
           r
     end ;;
```

条件変数は客のプロセスが待機するために使われています。call メソッドが呼ばれると待機中のすべての客のプロセスが起こされます。呼び出し番号の読み書きは mutex によって保護されています。

窓口　窓口は定数である窓口番号 (ncounter)、並んでいる客の列の長さ (nclient)、利用できるかどうかを表す真偽値 (available)、mutex と条件変数の 5 つのフィールドを持っています。

メソッドに関しては 8 つあり、そのうちの 2 つはプライベートメソッドです。単純なアクセスメソッドが 2 つ (methods get_ncounter と get_nclient)、職員の待ち時間を制御するメソッドが 3 つ (wait と await_arrival と await_departure)、客の動作をシミュレートするメソッドが 3 つ (set_available と arrive と depart) です。

```
# class counter (i:int) =
    object(self)
```

```
    val ncounter = i
    val mutable nclient = 0
    val mutable available = true
    val m = Mutex.create()
    val c = Condition.create()

    method get_ncounter = ncounter
    method get_nclient  = nclient

    method private wait f =
      Mutex.lock m ;
      while f () do Condition.wait c m done ;
      Mutex.unlock m

    method wait_arrival n = nclient <- n ; self#wait (fun () → available)
    method wait_departure () =  self#wait (fun () → not available)

    method private set_available b =
      Mutex.lock m ;
      available <- b ;
      Condition.signal c ;
      Mutex.unlock m
    method arrive () = self#set_available false
    method leave () = self#set_available true

  end ;;
```

郵便局 これらの 3 つの資源を一つのレコードに入れたものが郵便局を表しています。

```
# type office = { d : dispenser ; a : announcer ;  cs : counter array } ;;
```

客と職員

システム全体の動作は次に 3 種類のパラメータに依存しています。

```
# let service_delay = 1.7 ;;
# let arrival_delay = 1.7 ;;
# let counter_delay = 0.5 ;;
```

これらのパラメータは動作の最大時間を表しています。実際の動作時間はこれらの値を最大値とする範囲からランダムに決定されます。最初のパラメータは接客にかかる時間を表しています。2 番目のパラメータは客が郵便局にやってくる間隔を表しています。最後のパラメータは職員が客を処理した後、次の客を呼び出すまでの時間を表しています。

職員 職員の仕事は次の無限の繰り返しになっています。

1. 番号を呼ぶ。
2. その番号の客が窓口まで来るのを待つ。
3. 客が窓口から立ち去るのを待つ。

情報を表示する機能を付け加えると以下のような関数になります。

```
# let clerk ((a:announcer), (c:counter)) =
    while true do
      let n = a#call ()
      in Printf.printf "Counter %d calls %d\n" c#get_ncounter n ;
         c#wait_arrival n ;
         c#wait_departure () ;
         Thread.delay (Random.float counter_delay)
    done ;;
Characters 15-24:
  let clerk ((a:announcer), (c:counter)) =
                ^^^^^^^^^
Unbound type constructor announcer
```

客 客は次の動作を順番に行います。

1. 整理券を取る。
2. 自分の番号が呼ばれるまで待つ。
3. サービスを受けるために自分を呼び出した窓口へと行く。

少し複雑かもしれないのは自分を呼び出した窓口の番号を知る部分です。そのために補助関数を追加します。

```
# let find_counter n cs =
    let i = ref 0 in while cs.(!i)#get_ncounter <> n do incr i done ; !i ;;
val find_counter : 'a -> < get_ncounter : 'a; .. > array -> int = <fun>
```

情報を表示する機能を付け加えると客を表すコードの主要な部分は以下のようになります。

```
# let client o =
    let n = o.d#take()
    in Printf.printf "Arrival of client %d\n" n ; flush stdout ;
       o.a#wait n ;
       let ic = find_counter n o.cs
       in o.cs.(ic)#arrive () ;
          Printf.printf "Client %d occupies window %d\n" n ic ;
          flush stdout ;
          Thread.delay (Random.float service_delay) ;
          o.cs.(ic)#leave () ;
          Printf.printf "Client %d leaves\n" n ; flush stdout ;;
Characters 26-29:
    let n = o.d#take()
            ^^^
Unbound record field label d
```

システム全体

このアプリケーションのメインプログラムは郵便局、職員 (プロセスで表されます) を生成し、客の無限列 (客もまたプロセスで表されます) を生成するプロセスを走らせます。

```
# let main () =
    let o =
     { d = new dispenser();
       a = new announcer();
       cs = (let cs0 = Array.create 5 (new counter 0) in
              for i=0 to 4 do cs0.(i) <- new counter i done;
              cs0)
     }
    in for i=0 to 4 do ignore (Thread.create clerk (o.a, o.cs.(i))) done ;
       let create_clients o = while true do
                                 ignore (Thread.create client o) ;
                                 Thread.delay (Random.float arrival_delay)
                               done
       in ignore (Thread.create create_clients o) ;
          Thread.sleep () ;;
Characters 26-205:
  ..{ d = new dispenser();
      a = new announcer();
      cs = (let cs0 = Array.create 5 (new counter 0) in
             for i=0 to 4 do cs0.(i) <- new counter i done;
             cs0)
    }
Unbound record field label d
```

最後の命令によってメインプログラムを動かしているプロセスを待機状態にさせ、制御を直ちにアプリケーションを構成する他のプロセスに移します。

練習問題

食事にありつく哲学者

食事をする哲学者で起こりうるデッドロックを防ぐために同時にテーブルへと付く哲学者の人数を 4 人に制限する解決法があります。この解決法を実装しなさい。

郵便局の拡張

618 ページに記述された郵便局のシステムを次のように変更しなさい。せっかちな客は自分の番号が呼び出される前に帰ってしまいます。

1. クラス dispenser にメソッド wait (型は *int -> unit*) を加えなさい。このメソッドは最後に配布した整理券の番号がパラメータで指定された値以下である間、メソッドを呼び出したプロセスを待機状態にします。

2. クラス counter のメソッド await_arrival を変更して真偽値を返すようにしなさい。このメソッドは呼び出した客がやってきた時は true を返し、ある決められた時間が経過しても来なかった時は false を返します。
3. クラス announcer に整理券発行器を引数で渡すように書き換えなさい。
   (a) wait_until というメソッドを加えなさい。このメソッドはある指定された時間内に番号が呼ばれた時は true を返し、そうでない時は false を返します。
   (b) メソッド call を変更し、窓口をパラメータとして取るように書き換えなさい。渡された窓口の nclient を適切に書き換えるようにしなさい (クラス counter に更新メソッドを追加する必要があります)。
4. 関数 clerk が来ない客を待たないように書き換えなさい。
5. せっかちな客の行為をシミュレートする関数 impatient_client を書きなさい。

生産者消費者のオブジェクトバージョン

この練習問題では生産者消費者アルゴリズムを再び扱います。ただし商品は待ち行列にではなく、商品を有限個貯蔵できる店に蓄えられます。また郵便局の例と同じように資源をオブジェクトとして実装します。

1. 製品を表す、次の署名を持つクラス product を定義しなさい。

```
class product : string →
  object
    val name : string
    method name : string
  end
```

2. 商品を保管する、次の署名を持つクラス shop を定義しなさい。

```
class show : int →
  object
    val mutable buffer : product array
    val c : Condition.t
    val mutable ic : int
    val mutable ip : int
    val m : Mutex.t
    val mutable np : int
    val size : int
    method dispose : product → unit
    method acquire : unit → product
  end
```

   ic と ip はバッファの添字を表していて、それぞれ消費者と生産者によって制御されています。ic は最後に取り出され商品の添字を保存しています。ip は最後に納入された商品の添字を保存しています。np は保管されている商品の数を表しています。相互排除や生産者、消費者の待機状態の制御はこのクラスのメソッドで管理してください。
3. 消費者を実装する関数 consumer: *shop → string → unit* を定義してください。
4. 商品を生成する関数 create_product (型は *string -> product*) を定義しなさい。商品の名前は引数として与えられた文字列と商品ごとにユニークに与えられる

番号を結合したものとして定義されます。この関数を使って producer: *shop* → *string* → *unit* を定義しなさい。

まとめ

この章では多くのプロセスが共有メモリや同期通信などの手段によって相互に影響しあう並行プログラミングについて扱いました。まず始めに共有メモリを利用した同期や相互排除の機構を命令型のプログラミングスタイルの上で詳しく見ていきました。次に関数型プログラミングスタイル上での並行性のモデルを提供している同期通信について扱いました。特に関数クロージャや通信イベントを通信によって送る機能は複数のプロセスでの計算を合成する時に極めて有益です。

この章で扱ったプロセスはすべて Objective Caml の Thread モジュールが提供しているスレッドです。

もっと知りたい人へ

並行アルゴリズムはまず最初にシステムプログラミングの中で必要性が認識されました。システムプログラミングでは共有メモリ上での命令型スタイルが広く使われています。例えば mutex やセマフォアが共有資源を管理するために使われています。共有メモリにアクセスするプロセスの管理法には別の方法もあり、文献 [Ari90] が詳述しています。

特定の言語の中で並行アルゴリズムを記述できれば文献 [And91] に紹介されているような様々なアルゴリズムを考えるのに役に立ちます。そのようなアルゴリズムに含まれる様々な概念は問題を単純化するのに有益ですが、並行アルゴリズムを正確に実装するのは一般に非常に難しい作業になります。

Event モジュールで採用されている同期通信のモデルは CML で使われていたものですが、これは文献 [Rep99] に完全に記述されています。この文献は以下の URL 上でも入手できます。

リンク: http://cm.bell-labs.com/cm/cs/who/jhr/index.html

興味深いアプリケーションとしてスレッドを用いたグラフィクスライブラリである EXene があります。これは X-Windows 上で動作し CML を使って実装されています。上に示したページにはこのライブラリへのリンクも含まれています。

20

分散プログラミング

分散プログラミングという技法を用いれば、異なるマシンで稼働し、ネットワークを通じ協調して一つのタスクを成し遂げるようなアプリケーション群を構築することができます。ここで説明する計算モデルは、分散メモリを用いた並列計算です。ローカルプログラムとリモートプログラムとはネットワークプロトコルを用いて通信します。ネットワークプロトコルの中でも、最もよく知られていて、かつ最も広く使われているのが、IP（Internet Protocol）と、その上位層の TCP や UDP です。これらの低レベル層を始めとして、多くのサービスがクライアント - サーバモデルに基づき構築されています。クライアント - サーバモデルでは、サーバは複数の異なるクライアントからのリクエストを待ち受け、それを処理し、レスポンスを返します。例えば、HTTP というプロトコルはウェブブラウザとウェブサーバとの間の通信を可能としてくれます。クライアントとサーバとにタスクを振り分けるというやり方が適したソフトウェアアーキテクチャは数多くあります。

Objective Caml 言語は、`Unix` ライブラリを通じて、様々なプログラム間通信の方法を提供しています。ソケットにより、TCP/IP や UDP/IP といったプロトコルを用いた通信が可能となります。`Unix` ライブラリのソケットに関する部分は Windows にも移植されています。“ヘビー級” のプロセスは `Unix.fork` で、ライト級プロセスも `Thread.create` で生成できるので、多くの要求を一度に受け付けるサーバを作ることができます。最終的には、新しいサービスを考案する上で重要なポイントは、そのアプリケーションに適したプロトコルを定義するということになります。

この章の構成

この章では、分散アプリケーション（特にクライアント - サーバアプリケーション）構築のために、Internet とソケットの基本要素を紹介しつつ、通信プロトコルのデザインにあたって問題となる点を詳しく述べます。

最初の節では Internet とそのアドレッシングシステムや主なサービスについて簡単に説明します。

第 2 節では Objective Caml プロセス間のソケットを用いたローカル通信とリモート通信とについて解説します。

第 3 節ではクライアント - サーバモデルについて述べ、サーバプログラムと汎用クライアントとを紹介します。

第 4 節ではネットワークサービスを構築する上での通信プロトコルの重要性を示します。

この章を読むにあたっては、システムプログラミング（18 章）と並行プログラミング（19 章）に関する章を読んでおくとよいでしょう。

インターネット（*The Internet*）

Internet とは、ネットワークのネットワークです。ネットワーク間の相互接続は、**インタフェース（interfaces）**を通じて行なわれ、ドメイン、サブドメインといった階層構造を成しています。インタフェースとはコンピュータの中にあるハードウェアの一つ（典型的にはイーサネットカード）で、それを通じてコンピュータはネットワークに接続されます。一つのコンピュータが複数のインタフェースを持つ場合もあります。インタフェースはそれぞれ固有の IP アドレスを持っています。IP アドレスは通常、相互接続の階層構造を反映したものとなっています。メッセージのルーティングも、ドメインからドメイン、ドメインからサブドメイン、といった感じで、メッセージが宛先のインタフェースに到るまで、階層的に構成されています。コンピュータは、そのインタフェースのアドレスに加えて、大抵、名前を持っています。ドメインやサブドメインも名前を持ちます。ある種のマシンはネットワークの中で特別な役割を担っています。

ブリッジ ネットワーク間を繋ぎます。

ルータ Internet のトポロジに関する情報を利用して、データのルーティングを行ないます。

ネームサーバ マシン名とネットワークアドレスとの対応関係を記録します。

Internet プロトコルの目的（すなわち IP の目的）は、ネットワークが寄り集まってできたネットワークを、一つのものとして統一的に扱うということです。こういうわけで、英語だと Internet には “the” が付くのです。どんなマシンであろうと、Internet に接続されているマシン同士は通信することができます。Internet には多種多様なマシンやシステムが共存しています。それらは全て IP プロトコルを使い、そのほとんどが UDP や TCP 層を利用しています。

Internet で利用されるその他のプロトコルとサービスについては RFC （Request For Comments）に掲載されており、Jussieu ミラーサイト

リンク: ftp://ftp.lip6.fr/pub/rfc

から取得できます。

Internet プロトコルとサービス

IP プロトコルにおけるデータ転送の単位は、**データグラム（*datagram*）**あるいは**パケット（*packet*）**です。このプロトコルは信頼性が低く、転送されたパケットが正しい順序で安全に到着することや、重複がないことを保証してくれません。パケットの正しいルーティングと、パケットが宛先に到着できない場合のエラーシグナリング処理してくれるだけです。アドレスは、現在のバージョン IPv4 では 32 ビット値として符合化されます。それは 4 つのフィールドに分割され、それぞれが 0 から 255 までの値を含みます。フィールドは、例えば、`132.227.60.30` のようにピリオドで区切られます。

IP プロトコルは現在、重要な変化の真っ只中にあります。その変化は、Internet の拡大に伴うアドレス空間の枯渇、ルーティング問題の複雑化により必然となっています。IP プロトコルの新しいバージョンは IPv6 です。[Hui97] で述べられています。

IP の上位にある二つのプロトコル、UDP（User Datagram Protocol）、TCP（Transfer Control Protocol）により、より高度なデータ送信が可能となります。これらのプロトコルは、複数マシン間の通信に IP を利用し、さらに、それらのマシンで稼働するアプリケーション（プログラム）間の通信を可能とします。また、データの内容に依存しない、正しい情報転送のための処理が織り込まれています。アプリケーションは**ポート番号（port number）**で識別されます。

UDP はコネクションレス型の低信頼プロトコルです。IP がインタフェース間を繋ぐのに対し、アプリケーション間を繋ぎます。TCP はコネクション型の高信頼プロトコルです。パケットの確認応答、再送、順序付けを処理してくれます。さらに、ウィンドウ技法を使って送信を最適化できます。

標準的な Internet サービス（アプリケーション）は、全てクライアント - サーバモデルを踏襲していると言っても過言ではありません。サーバはクライアントからの要求を処理し、それぞれのクライアントに特定のサービスを提供します。クライアントとサーバとの関係は非対称的です。標準的なサービスでは、注意深くコンテンツを転送する高度なプロトコルを確立しています。標準的なサービスの中でも、特に下のものを挙げておきます。

- FTP（File Transfer Protocol）
- TELNET（Terminal Protocol）
- SMTP（Simple Mail Transfer Protocol）
- HTTP（Hypertext Transfer Protocol）

他にも、クライアント - サーバモデルを用いたサービスには、次のようなものがあります。

- NFS（Network File System）
- X-Windows[1]
- rlogin、rwho などの Unix サービス

1. 訳注：正確には “X Window System”。

アプリケーション間の通信はソケットを経て行なわれます。ソケットは（一般には異なるマシン上の）プロセス間の通信を実現します。それぞれのプロセスはソケットに対して読み書きを行なうことができます。

Unix モジュールと IP アドレッシング

Unix ライブラリでは、Internet アドレスを表現する抽象型 *inet_addr* が定義されています。アドレスの内部表現と文字列との変換を行なう二つの関数も定義されています。

```
# Unix.inet_addr_of_string ;;
- : string -> Unix.inet_addr = <fun>
# Unix.string_of_inet_addr ;;
- : Unix.inet_addr -> string = <fun>
```

アプリケーションの内部では、サービス（あるいはサービス番号）に対する Internet アドレスとポート番号は、大抵の場合、名前で置き換えられます。名前とアドレスや番号との対応はデータベースで管理されます。Unix ライブラリは、データベースに対してデータリクエストを行なう関数や、取得した情報を保存するためのデータ型を提供しています。以下、それらの関数について簡単に説明します。

アドレステーブル アドレステーブル（*hosts database*）は、マシン名とインタフェースアドレスとの間の対応関係を含んでいます。アドレステーブルのエントリの構造は下のように表現されます。

```
# type host_entry =
    { h_name : string;
      h_aliases : string array;
      h_addrtype : socket_domain;
      h_addr_list : inet_addr array } ;;
```

最初の二つのフィールドはマシン名とその別名を含み、三つ目のフィールドはアドレスの型（629 ページ参照）、最後のフィールドはマシンアドレスのリストを含みます。

次の関数を利用してマシン名を知ることができます。

```
# Unix.gethostname ;;
- : unit -> string = <fun>
# let my_name = Unix.gethostname() ;;
val my_name : string = "dynabook.is.s.u-tokyo.ac.jp"
```

アドレステーブルに対する問い合わせには、名前かマシンアドレスが必要となります。

```
# Unix.gethostbyname ;;
- : string -> Unix.host_entry = <fun>
# Unix.gethostbyaddr ;;
- : Unix.inet_addr -> Unix.host_entry = <fun>
# let my_entry_byname = Unix.gethostbyname my_name ;;
val my_entry_byname : Unix.host_entry =
  {Unix.h_name = "dynabook.is.s.u-tokyo.ac.jp";
   Unix.h_aliases = [|"dynabook"; "localhost"|];
   Unix.h_addrtype = Unix.PF_INET; Unix.h_addr_list = [|<abstr>|]}
# let my_addr = my_entry_byname.Unix.h_addr_list.(0) ;;
```

```
val my_addr : Unix.inet_addr = <abstr>

# let my_entry_byaddr = Unix.gethostbyaddr my_addr ;;
val my_entry_byaddr : Unix.host_entry =
  {Unix.h_name = "dynabook.is.s.u-tokyo.ac.jp";
   Unix.h_aliases = [|"dynabook"; "localhost"|];
   Unix.h_addrtype = Unix.PF_INET; Unix.h_addr_list = [|<abstr>|]}

# let my_full_name = my_entry_byaddr.Unix.h_name ;;
val my_full_name : string = "dynabook.is.s.u-tokyo.ac.jp"
```

これらの関数は、要求の処理が失敗した場合、例外 Not_found を発生させます。

サービステーブル　サービステーブルはサービス名とポート番号との対応関係を保持しています。大半の Internet サービスは規格化されています。テーブル中のエントリの構造は、次の通りです。

```
# type service_entry =
    { s_name : string;
      s_aliases : string array;
      s_port : int;
      s_proto : string } ;;
```

最初の二つのフィールドはサービス名とその最終的な別名で、三番目のフィールドはポート番号、最後のフィールドは使用されるプロトコル名を含みます。

実際のところ、サービスは、ポート番号と使用されるプロトコルとで特徴付けられます。問い合わせ関数は次の通りです。

```
# Unix.getservbyname ;;
- : string -> string -> Unix.service_entry = <fun>
# Unix.getservbyport ;;
- : int -> string -> Unix.service_entry = <fun>
# Unix.getservbyport 80 "tcp" ;;
- : Unix.service_entry =
{Unix.s_name = "www"; Unix.s_aliases = [|"http"|]; Unix.s_port = 80;
 Unix.s_proto = "tcp"}
# Unix.getservbyname "ftp" "tcp" ;;
- : Unix.service_entry =
{Unix.s_name = "ftp"; Unix.s_aliases = [||]; Unix.s_port = 21;
 Unix.s_proto = "tcp"}
```

これらの関数は、要求されたサービスが見つからない場合、例外 Not_found を発生させます。

ソケット

18 章、19 章では、プロセス間で通信を行なうための二つの方法を見てきました。すなわち、パイプとチャネルです。これらの方法では、並行性という論理的なモデルに基づいています。一般に、パイプやチャネルといった方法は、通信するプロセスがリソース（特にプロセッサ）を共有する場合にしか適用できません。三つ目の可能性として、この節では通信のためにソケットを利用します。この方法は、Unix の世界に端を発します。

ソケットを利用すれば、同一マシン、あるいは異なるマシン上にあるプロセス間の通信が可能となります。

記述と生成

ソケットは、最終的に情報転送を実現するうえで、他のソケットとの通信の確立に対して責任を持ちます。TCP/IP ソケットで利用される関数やデータ型に加えて、実際よくありそうな状況を挙げます。ソケットは、昔から、よく電話機に例えられます。

- 機能するには、ネットワークに接続されなくてはなりません。(`socket`)
- 通話を受けるためには、*sock_addr* という電話番号を持つ必要があります。(`bind`)
- 設定されていれば、通話中に他の通話を受けることができます。(`listen`)
- 一度接続が確立されたら、相手を呼び出すのに自身の電話番号は不要です。(`connect`)

ドメイン　ソケットは、内部での通信に使うか、外部への通信に使うかに応じて、異なる**ドメイン（domains）**に属します。`Unix` ライブラリでは、型コンストラクタに対応して二つのドメインが定義されています。

```
# type socket_domain = PF_UNIX | PF_INET;;
```

最初のドメインはローカルな通信に対応しています。二つ目のドメインは、Internet 上の通信に対応しています。これらが *sockets* に対する主要なドメインとなります。

以降、Internet ドメインのソケットのみを扱います。

型とプロトコル　ドメインとは別に、ソケットは通信に関するプロパティ（信頼性、順序付けなど）を設定します。プロパティは型コンストラクタで表現されます。

```
# type socket_type = SOCK_STREAM | SOCK_DGRAM | SOCK_SEQPACKET | SOCK_RAW ;;
```

使われるソケットの型に応じて、利用する通信プロトコルの性質が限定されます。通信のそれぞれの型には、デフォルトのプロトコルが割付けられています。

実際に、最初の通信形態 — `SOCK_STREAM` — は、デフォルトのプロトコル TCP としか使いません。これにより、信頼性、順序、が保証され、メッセージの重複はなくなり、接続済モードで動作します。

より詳しい情報については、例えば [Ste92] などの Unix に関する文献を参考文献として挙げておきます。

生成　ソケットを生成は、次の関数で行ないます。

```
# Unix.socket ;;
- : Unix.socket_domain -> Unix.socket_type -> int -> Unix.file_descr = <fun>
```

三つ目の引数で、行なおうとする通信で利用するプロトコルを指定できます。値として 0 を指定すると、「デフォルトプロトコル」と解釈されます。これは、ソケット生成に使われた引数 (ドメイン、ソケットのタイプ) から決められます。この関数の返り値はファイルデスクリプタなので、`Unix` ライブラリにある標準的な入出力関数で利用することができます。TCP/IP ソケットは次のように生成できます。

```
# let s_descr = Unix.socket Unix.PF_INET Unix.SOCK_STREAM 0 ;;
val s_descr : Unix.file_descr = <abstr>
```

警告

socket 関数が *file_descr* 型の値を返した場合でも、システムはファイル用のデスクリプタとソケット用のデスクリプタとを区別します。Unix ライブラリにあるファイル操作関数に、ソケット用のデスクリプタを渡すことはできますが、逆に、ソケット用のデスクリプタを受け取るよう作られた関数に、通常ファイル用のデスクリプタを渡すと例外が発生します。

クローズ　ファイルデスクリプタ同様、ソケットも次の関数でクローズされます。

```
# Unix.close ;;
- : Unix.file_descr -> unit = <fun>
```

プロセスが exit の呼び出しで終了する場合、オープンされているファイルのデスクリプタは、全て自動的にクローズされます。

アドレスと接続

ソケットは生成時にはアドレスを持ちません。二つのソケット間の接続を確立するためには、呼び出し側が受信側のアドレスを知らねばなりません。ソケット（TCP/IP）のアドレスは IP アドレスとポート番号から構成されます。Unix ドメインのソケットの場合は単にファイル名となります。

```
# type sockaddr =
    ADDR_UNIX of string | ADDR_INET of inet_addr * int ;;
```

ソケットをアドレスにバインドする　ソケット生成の後、呼び出しを受け付けるためには、まずソケットをアドレスに **bind** しなくてはなりません。これは次の関数の役目です。

```
# Unix.bind ;;
- : Unix.file_descr -> Unix.sockaddr -> unit = <fun>
```

実際に、ソケットデスクリプタを得たとしても、その生成時に割り付けられたアドレスは、ほとんど使いものになりません。それは次の例で分かります。

```
# let (addr_in, p_num) =
    match Unix.getsockname s_descr with
      Unix.ADDR_INET (a,n) → (a,n)
    |  _ → failwith "not INET" ;;
val addr_in : Unix.inet_addr = <abstr>
val p_num : int = 0
# Unix.string_of_inet_addr addr_in ;;
- : string = "0.0.0.0"
```

ちゃんと使えるアドレスを生成し、ソケットに割り付ける必要があります。629 ページで使用したローカルアドレス my_addr を再利用し、通常空いている 12345 ポートを選んでみます。

```
# Unix.bind s_descr (Unix.ADDR_INET(my_addr, 12345)) ;;
- : unit = ()
```

リッスンと接続のアクセプト　ソケットが完璧に呼び出しを受け付けることができるようになるまで、二つの操作が必要となります。リッスンの容量を決めることと、接続をアクセプトできるようにすることです。それらは、それぞれ次の二つの関数の役目となります。

```
# Unix.listen ;;
- : Unix.file_descr -> int -> unit = <fun>
# Unix.accept ;;
- : Unix.file_descr -> Unix.file_descr * Unix.sockaddr = <fun>
```

listen 関数の二番目の引数には、接続数の上限値を与えます。accept 関数の呼出は接続要求を待ち受けます。accept は終了時にソケット、いわゆる **service socket** のデスクリプタを返します。このサービスソケットは自動的にあるアドレスに割り付けられます。accept 関数は、listen を呼び出したソケット、つまり、接続リクエストのキューをセットアップしたソケットにのみ適用されます。

接続要求　accept を呼び出した側の通信相手が呼び出す関数は、次の関数です。

```
# Unix.connect ;;
- : Unix.file_descr -> Unix.sockaddr -> unit = <fun>
```

Unix.connect s_descr s_addr の呼出は、ローカルソケット *s_descr*（自動的にバインドされる）とアドレス *s_addr*（存在しなくてはならない）を持つソケットとの間の通信を確立します。

通信　二つのソケット間の接続が確立された瞬間から、それを持つプロセス同士は双方向通信ができるようになります。入出力関数は、Unix モジュールの中にあります。18 章に記述があります。

クライアント - サーバ

TCP/IP ソケットを通じた、同一あるいは異なるマシン上のプロセス間通信は、ポイント - トゥ - ポイントの非同期通信となります。そのような通信の信頼性は TCP プロトコルにより保証されます。ポイント - トゥ - ポイントの通信を全ての受信プロセスに対して行なうことで、プロセスのグループへのブロードキャストをシミュレートすることは可能です。

一つのアプリケーションの構成要素として通信する複数のプロセスの役割は、非対称的なのが一般的です。それはクライアント - サーバアーキテクチャにも当てはまります。サーバは、一つ（あるいは複数）のプロセスで、リクエストを受け付け、それに対しレスポンスを返そうとします。クライアントはそれ自体が一つのプロセスでリクエストをサーバに送り、レスポンスを期待します。

クライアント - サーバの動作モデル

サーバは、ある決められたポートで **service** を提供します。そこでクライアントからの接続を待ちます。図 20.1 に、サーバとクライアントの主なタスクを示します。

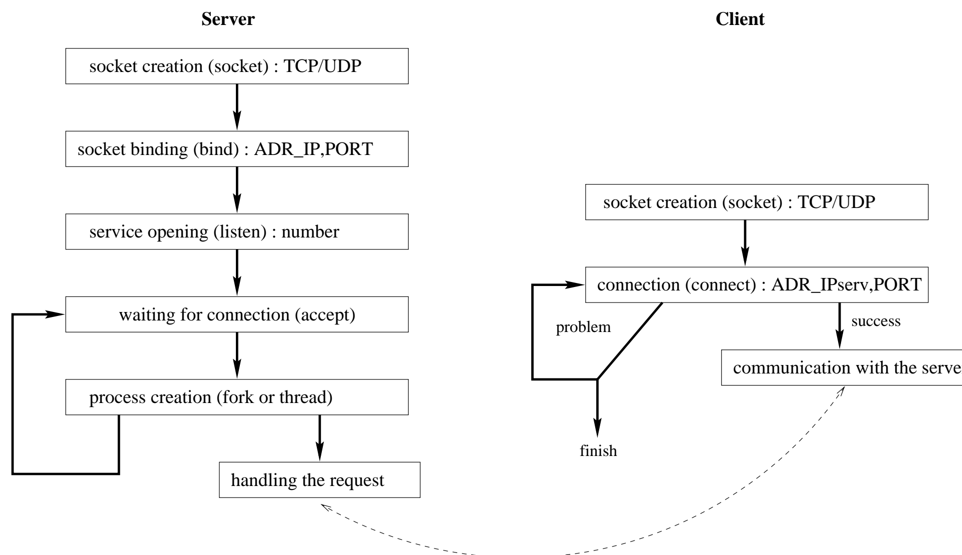


図 20.1: サーバとクライアントのモデル

クライアントは、サーバが接続をアクセプト（`accept`）する準備が整えば、ただちにサービスへと接続することができます。接続のためには、クライアントはサーバマシンの IP 番号とサービスのポート番号とを知らなくてはなりません。もし、IP 番号が分からない場合は、名前／番号の解決を `gethostbyname` で行なう必要があります。接続がサーバにアクセプトされれば、双方のプログラムは、生成されたソケットに対する入出力チャネルを通じて通信ができます。

クライアント - サーバプログラミング

クライアント - サーバプログラミングのメカニズムは、図 20.1 に示されたモデルに沿っています。これらのタスクは全て必要なものです。そこで、これらのタスクをまとめて行なうような、汎用的な関数を書きます。この関数は、関数パラメタを与えることで、それぞれのサーバ用に特化して使います。プログラム例として、一つサーバを書いてみます。そのサーバは、クライアントからの接続をアクセプトし、テクストが一行分得ら

れるまでソケットから文字を読み出し、そのテクストを大文字に変換し、それをクライアントに返します。

図 20.2 は、サービスと複数クライアント[2]間の通信の様子を示しています。

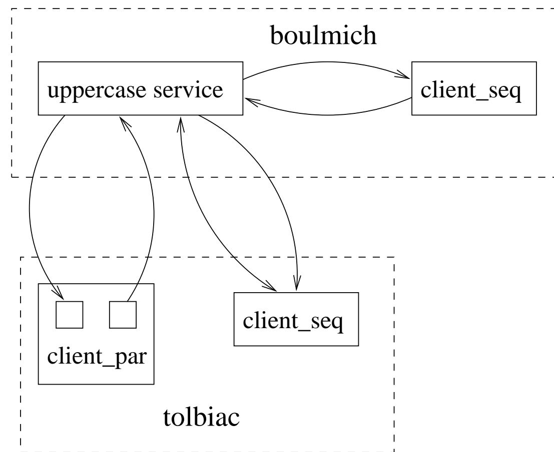


図 20.2: 大文字化サービスとそのクライアント

サーバと同じマシン上で実行されるタスクもあれば、リモートマシン上で実行されるタスクもあります。次のようなことを見てゆきます。

1. “汎用サーバ” コードの書き方と、それを大文字化サービスに特化する方法。
2. `telnet` プログラムを利用して、クライアントを書かずにサーバをテストする方法。
3. 次の二つのタイプのクライアントを作成する方法。
   - 逐次クライアント。リクエストを送った後、レスポンスを待ちます。
   - 並列クライアント。送信タスクと受信タスクとを分離します。つまり、この種のクライアントには、プロセスが二つとなります。

サーバのコード

サーバは二つの部分に分けることができます。接続を待つ部分と、それに続き接続を処理する部分です。

汎用サーバ

下に記した、汎用サーバ関数 `establish_server` は、第一引数にリクエストを処理するサービス関数（`server_fun`）をとり、第二引数にリクエストをリッスンする Internet ド

2. 英訳注：“boulmich” とは “Boulevard Saint-Michel” の略で話言葉で使います。パリのカルチェラタンの主要な大通りの一つです。

メインのソケットアドレスをとります。この関数は、ソケットアドレスからドメインを取得する補助関数 domain_of を使います。

実際には、establish_server 関数は、Unix ライブラリの高度な関数から作られています。この関数は、サーバへの接続をセットアップします。

```
# let establish_server server_fun sockaddr =
    let domain = domain_of sockaddr in
    let sock = Unix.socket domain Unix.SOCK_STREAM 0
    in Unix.bind sock sockaddr ;
       Unix.listen sock 3;
       while true do
         let (s, caller) = Unix.accept sock
         in match Unix.fork() with
                0 → if Unix.fork() <> 0 then exit 0 ;
                     let inchan = Unix.in_channel_of_descr s
                     and outchan = Unix.out_channel_of_descr s
                     in server_fun inchan outchan ;
                        close_in inchan ;
                        close_out outchan ;
                        exit 0
              | id → Unix.close s; ignore(Unix.waitpid [] id)
       done ;;
val establish_server :
  (in_channel -> out_channel -> 'a) -> Unix.sockaddr -> unit = <fun>
```

サーバをポート番号をパラメタとして受け取る、単独で実行可能なものに仕上げるために、サービスを示すパラメタをとる main_server 関数を書きます。この関数はコマンドラインパラメタをポート番号として利用します。補助関数 get_my_addr はローカルマシンのアドレスを返します。

```
# let get_my_addr () =
    (Unix.gethostbyname(Unix.gethostname())).Unix.h_addr_list.(0) ;;
val get_my_addr : unit -> Unix.inet_addr = <fun>

# let main_server  serv_fun =
    if Array.length Sys.argv < 2 then Printf.eprintf "usage : serv_up port\n"
    else try
           let port =  int_of_string Sys.argv.(1) in
           let my_address = get_my_addr()
           in establish_server serv_fun  (Unix.ADDR_INET(my_address, port))
         with
           Failure("int_of_string") →
             Printf.eprintf "serv_up : bad port number\n" ;;
val main_server : (in_channel -> out_channel -> 'a) -> unit = <fun>
```

サービス記述コード

以上で汎用的なメカニズムは用意できました。それがどのように動作するかを説明するために、対象となるサービスを定義する必要があります。ここでのサービスは、文字列を大文字に変換するというものです。入力チャネルを通じて一行のテクストを読み出し、変換し、出力チャネルに書き出し、出力バッファをフラッシュします。

```
# let uppercase_service ic oc =
    try while true do
          let s = input_line ic in
          let r = String.uppercase s
          in output_string oc (r^"\n") ; flush oc
        done
    with _ → Printf.printf "End of text\n" ; flush stdout ; exit 0 ;;
val uppercase_service : in_channel -> out_channel -> unit = <fun>
```

Unix ライブラリで発生した例外から正しく復帰するために、サービスの最初の呼出を Unix ライブラリにある関数を使って特別にラップします。

```
# let go_uppercase_service () =
    Unix.handle_unix_error main_server uppercase_service ;;
val go_uppercase_service : unit -> unit = <fun>
```

サービスのコンパイルとテスト

実際の go_uppercase_service の呼び出し部分を加えて、関数群をファイル serv_up.ml にまとめます。このファイルを Unix ライブラリがリンクされることを指定してコンパイルします。

```
ocamlc -i -custom -o serv_up.exe unix.cma serv_up.ml -cclib -lunix
```

この（-i オプションを使った）コンパイルの結果は次のようになります。

```
val establish_server :
  (in_channel -> out_channel -> 'a) -> Unix.sockaddr -> unit
val main_server : (in_channel -> out_channel -> 'a) -> unit
val uppercase_service : in_channel -> out_channel -> unit
val go_uppercase_service : unit -> unit
```

サーバの立ち上げには次のように書きます。

```
serv_up.exe 1400
```

選んだポート番号は 1400 です。これで、サーバが上がっているマシンはそのポートへの接続をアクセプトするようになります。

telnet を利用したテスト

ここまでくれば、テクストの送受信に既存のクライアントを使って、サーバのテストを始められます。telnet ユーティリティは、本来 23 番ポートの telnetd サービスのためのクライアントであり、リモート接続を制御するのに使われるのですが、マシン名と別のポート番号を与えることで流用できます。このユーティリティは、幾つかの OS には標準装備です。Unix でテストする場合は次のように入力します。

```
$ telnet boulmich 1400
Trying 132.227.89.6...
Connected to boulmich.ufr-info-p6.jussieu.fr.
Escape character is '^]'.
```

boulmich の IP アドレスは 132.227.89.6 で、ドメイン名を含んだ完全な名前は、boulmich.ufr-info-p6.jussieu です。telnet により表示されたテクストは、サーバへの接続が成功したことを示しています。クライアントはキーボードからの入力を待ち、文字を boulmich の 1400 番ポートで起動したサーバに送ります。サーバからのリプライを待ち、次のような表示をします。

```
The little cat is dead.
THE LITTLE CAT IS DEAD.
We obtained the expected result.
WE OBTAINED THE EXPECTED result.
```

ユーザが入力したフレーズは小文字で、サーバが送ってきたフレーズは大文字となっています。この変換は、まさにこのサービスの役割です。

クライアントを終了するには、kill コマンドを使ってそれが実行されたウィンドウを閉じる必要があります。このコマンドは、クライアントのソケットをクローズし、それによってサーバ側のソケットもクローズされます。サーバが “End of text” というメッセージを表示すると、サービスのプロセスは終了します。

クライアントコード

サーバが並列的なのは当然（一定の限界までリクエストを処理しつつ別のリクエストもアクセプトしたい。）なのですが、クライアントの場合はアプリケーションの性質次第となります。下では二つのバージョンのクライアントを挙げます。まず、それらのクライアントを書くのに便利な、二つの関数を紹介します。

Unix ライブラリにある open_connection 関数によりソケットに対する一組の入出力チャネルを得ることができます。

次のコードは、言語システムの配布物に含まれています。

```
# let open_connection sockaddr =
    let domain = domain_of sockaddr in
    let sock = Unix.socket domain Unix.SOCK_STREAM 0
    in try Unix.connect sock sockaddr ;
           (Unix.in_channel_of_descr sock , Unix.out_channel_of_descr sock)
```

```
    with exn → Unix.close sock ; raise exn ;;
val open_connection : Unix.sockaddr -> in_channel * out_channel = <fun>
```

同様に、shutdown_connection 関数はソケットをクローズします。

```
# let shutdown_connection inchan =
    Unix.shutdown (Unix.descr_of_in_channel inchan) Unix.SHUTDOWN_SEND ;;
val shutdown_connection : in_channel -> unit = <fun>
```

逐次的クライアント

これらの関数を使って、逐次的なクライアントのメイン関数を書くことができます。このクライアントは引数として、リクエストを送信しレスポンスを受信する関数をとります。メイン関数はコマンドライン引数を解析し、実際の処理の前に接続パラメタを得ます。

```
# let main_client client_fun  =
    if Array.length Sys.argv < 3
    then Printf.printf "usage :  client server port\n"
    else let server = Sys.argv.(1) in
         let server_addr =
           try  Unix.inet_addr_of_string server
           with Failure("inet_addr_of_string") →
                  try  (Unix.gethostbyname server).Unix.h_addr_list.(0)
                  with Not_found →
                         Printf.eprintf "%s : Unknown server\n" server ;
                         exit 2
         in try
              let port = int_of_string (Sys.argv.(2)) in
              let sockaddr = Unix.ADDR_INET(server_addr,port) in
              let ic,oc = open_connection sockaddr
              in client_fun ic oc ;
                 shutdown_connection ic
            with Failure("int_of_string") → Printf.eprintf "bad port number";
                                            exit 2 ;;
val main_client : (in_channel -> out_channel -> 'a) -> unit = <fun>
```

後はクライアント処理の関数を書くのみです。

```
# let client_fun ic oc =
    try
      while true do
        print_string  "Request : " ;
        flush stdout ;
        output_string oc ((input_line stdin)^"\n") ;
        flush oc ;
        let r = input_line ic
        in Printf.printf "Response : %s\n\n" r;
           if r = "END" then ( shutdown_connection ic ; raise Exit) ;
      done
    with
```

```
        Exit → exit 0
      | exn → shutdown_connection ic ; raise exn  ;;
val client_fun : in_channel -> out_channel -> unit = <fun>
```

client_fun 関数は無限ループに入ります。そのループで、キーボードから入力を読み込み、サーバに文字列を送り、変換された大文字の文字列を受けとり、それを表示します。もし、入力文字列が "END"であれば、例外 Exit が発生し、ループを終了させます。他の例外が発生した場合は、典型的にはサーバが落ちた場合ですが、処理を中止します。

以上からクライアントプログラムは次のようになります。

```
# let go_client () = main_client client_fun ;;
val go_client : unit -> unit = <fun>
```

fork を利用した並列的クライアント

先程少し触れた並列クライアントは、タスクを二つのプロセスで分割します。一つのプロセスは送信用に、もう一つは受信用に利用します。プロセスは一つのソケットを共有します。各プロセスに対応した関数は、それぞれのプロセスにパラメタとして与えられます。

修正したプログラムは次のようになります。

```
# let main_client client_parent_fun client_child_fun  =
    if Array.length Sys.argv < 3
    then Printf.printf "usage :  client server port\n"
    else
      let server = Sys.argv.(1) in
      let server_addr =
        try  Unix.inet_addr_of_string server
        with Failure("inet_addr_of_string")
               → try  (Unix.gethostbyname server).Unix.h_addr_list.(0)
                  with Not_found →
                         Printf.eprintf "%s : unknown server\n" server ;
                         exit 2
      in try
           let port = int_of_string (Sys.argv.(2)) in
           let sockaddr = Unix.ADDR_INET(server_addr,port) in
           let ic,oc = open_connection sockaddr
           in match Unix.fork () with
                  0 → if Unix.fork() = 0 then client_child_fun oc ;
                        exit 0
                | id → client_parent_fun ic ;
                        shutdown_connection ic ;
                        ignore (Unix.waitpid [] id)
          with
             Failure("int_of_string") → Printf.eprintf "bad port number" ;
                                        exit 2 ;;
val main_client : (in_channel -> 'a) -> (out_channel -> unit) -> unit = <fun>
```

子（孫）プロセスが要求を送信し、親が応答を受信するというのがパラメタとして与えられる関数に期待された挙動です。

このアーキテクチャには、次のような効果があります。つまり、子プロセスが幾つか複数のリクエストを送る必要がある場合、そのリクエスト処理と並行して、親プロセスがレスポンスを受け付けます。再び文字列大文字化の例で考えます。クライアント側プログラムを修正します。クライアントはあるファイルからテキストを読み込み、レスポンスを別のファイルに書き込みます。このために、小さなプロトコル（つまり、文字列"END"を認識するということ）を守りつつ、チャネル ic から別のチャネル oc へのコピーを行なうような関数が必要となります。

```
# let copy_channels ic oc =
    try while true do
          let s = input_line ic
          in if s = "END" then raise End_of_file
             else (output_string oc (s^"\n"); flush oc)
        done
    with End_of_file → () ;;
val copy_channels : in_channel -> out_channel -> unit = <fun>
```

並列クライアントモデルに従って、親プロセス用と子プロセス用に、二つの関数を書きます。

```
# let child_fun in_file out_sock =
     copy_channels in_file out_sock ;
     output_string out_sock ("END\n") ;
     flush out_sock ;;
val child_fun : in_channel -> out_channel -> unit = <fun>
# let parent_fun out_file in_sock = copy_channels in_sock out_file ;;
val parent_fun : out_channel -> in_channel -> unit = <fun>
```

これで、メインクライアント関数を書くことができます。コマンドラインから、入力ファイル名、出力ファイル名というパラメタがさらに必要となります。

```
# let go_client () =
   if Array.length Sys.argv < 5
     then Printf.eprintf "usage : client_par server port filein fileout\n"
   else let in_file = open_in Sys.argv.(3)
        and out_file = open_out Sys.argv.(4)
        in main_client (parent_fun out_file) (child_fun in_file) ;
           close_in in_file ;
           close_out out_file ;;
val go_client : unit -> unit = <fun>
```

全ての材料を一つのファイル client_par.ml にまとめて、（go_client の呼出を加えることを確認して）コンパイルします。変換するテキストを含むファイル toto.txt を作ります。

```
The little cat is dead.
We obtained the expected result.
```

クライアントをテストするために、次のように入力します。

```
client_par.exe boulmich 1400 toto.txt result.txt
```

ファイル result.txt は次のテクストを含みます。

```
$ more result.txt
THE LITTLE CAT IS DEAD.
WE OBTAINED THE EXPECTED RESULT.
```

クライアントが終了すると、サーバは必ず"End of text"というメッセージを表示します。

ライト級プロセスを用いたクライアント - サーバプログラミング

今までの、汎用サーバのコード表現と並列的クライアントでは、Unix ライブラリの fork プリミティブを使ってプロセスを生成しました。これは、Unix ではうまく動作します。多くの Unix サービスがこの技法で実装されています。ところが残念なことに、同じことは Windows には当てはまりません。移植性のために、19 章のように、クライアント - サーバコードをライト級プロセスを使って書く方がより好ましいでしょう。この場合、サーバプロセス間のやりとりについて調べる必要が出てきます。

スレッドと Unix ライブラリ

ライト級プロセスと Unix ライブラリとを同時に使用すると、全てのアクティブなスレッドが、システムコールがすぐにリターンしない場合に、ブロックしてしまいます。特に、socket や他のものによって作られたファイルテスクリプタからの読み込みはブロックしてしまいます。

この問題を回避するため、ThreadUnix ライブラリは、Unix ライブラリの中の入出力関数の大半を再実装しています。このライブラリで定義されている関数は、まさにシステムコールを呼び出しているスレッドのみをブロックします。その結果として、入出力は、ThreadUnix ライブラリ中の低レベルな関数、read と write とを使うことになります。

例えば、文字列を読む標準関数 input_line は、一行を読む間、他のスレッドをブロックしないように再定義されます。

```
# let my_input_line  fd =
    let s = " "  and  r = ref ""
    in while (ThreadUnix.read fd s 0 1 > 0) && s.[0] <> '\n' do r := !r ^s done ;
       !r ;;
Characters 70-85:
    in while (ThreadUnix.read fd s 0 1 > 0) && s.[0] <> '\n' do r := !r ^s done ;
             ^^^^^^^^^^^^^^^
Unbound value ThreadUnix.read
```

スレッドを使用したサーバのクラス

ここで、大文字化サービスの例を、今回はライトウェイトプロセス利用バージョンとして再利用します。我々の小さなアプリケーションでは、プロセスが独立的に動作する[3]ので、スレッドへの移行はサーバ側、クライアント側ともに問題なく行なえます。

これまでに、サービス関数でパラメタ化された汎用サーバを構築してきました。このような抽象化を達成することができたのは Objective Caml 言語が関数型言語の側面を持っていたおかげです。ここでは、Objective Caml 言語のオブジェクト指向拡張を利用し、オブジェクトによって同等の抽象化がどのように実現されるかを示します。

サーバは二つのクラスで構成されます。serv_socket と connection です。最初のクラスはサービスのスタートアップを処理します。二つ目のクラスはサービスそのものを処理します。サービスの主な段階を追跡するために、プリント文をいくつか挿入しました。

serv_socket クラス は、二つのインスタンス変数を持ちます。port は、サービスのためのポート番号であり、socket は、リッスンのためのソケットです。このオブジェクトを生成するときは、初期化関数がサービスをオープン（IP アドレスとポートを確定）し、ソケットアドレスに生成されたソケットをバインドします。run メソッドは接続をアクセプトし、リクエスト処理のために新しく connection オブジェクトを生成します。*serv_socket* は、次の段落で述べられている *connection* クラスを使用します。大抵の場合、このクラスは *serv_socket* クラスに先立って定義されなくてはなりません。

```
# class serv_socket p =
    object (self)
      val port = p
      val mutable sock = ThreadUnix.socket Unix.PF_INET Unix.SOCK_STREAM 0

      initializer
        let my_address = get_my_addr ()
        in Unix.bind sock (Unix.ADDR_INET(my_address,port)) ;
           Unix.listen sock 3

      method private client_addr = function
          Unix.ADDR_INET(host,_) → Unix.string_of_inet_addr host
        | _ → "Unexpected client"

      method run () =
        while(true) do
          let (sd,sa) = ThreadUnix.accept sock in
          let connection = new connection(sd,sa)
          in Printf.printf "TRACE.serv: new connection from %s\n\n"
                             (self#client_addr sa) ;
          ignore (connection#start ())
        done
    end ;;
```

3. 訳注：メモリの同じ番地へのアクセスがない。もしあれば意味が変わってしまう場合がある。

```
Characters 82-99:
      val mutable sock = ThreadUnix.socket Unix.PF_INET Unix.SOCK_STREAM 0
                         ^^^^^^^^^^^^^^^^^
Unbound value ThreadUnix.socket
```

このクラスを継承し、run メソッドを再定義すれば、サーバを改良することができます。

connection クラス　このクラスのインスタンス変数、s_descr と s_addr は、accept により生成されたソケットのアドレスとデスクリプタに初期化されます。メソッドは、start、run、stop です。start は他の二つのメソッドを呼び出すスレッドを生成し、そのスレッドの識別子を返します。その識別子は、このメソッドを呼び出している serv_socket のインスタンスが使うことができます。run メソッドは、サービスの核となる機能を含みます。サービスの終了条件は少々変更されました。空文字列の受信で終了します。stop はサービス用のソケットのデスクリプタをクローズするだけです。

新しい接続はそれぞれ、インスタンスが生成されるときに補助関数 gen_num を呼び出して得られる番号を割り当てられます。

```
# let gen_num = let c = ref 0 in (fun () -> incr c; !c) ;;
val gen_num : unit -> int = <fun>
# exception Done ;;
exception Done
# class connection (sd,sa) =
    object (self)
      val s_descr = sd
      val s_addr = sa
      val mutable number = 0
      initializer
        number <- gen_num();
        Printf.printf "TRACE.connection : object %d created\n" number ;
        print_newline()

      method start () =  Thread.create (fun x -> self#run x ; self#stop x) ()

      method stop() =
        Printf.printf "TRACE.connection : object finished %d\n" number ;
        print_newline () ;
        Unix.close s_descr

     method run () =
       try
         while true do
           let line =  my_input_line s_descr
           in if (line = "") or (line = "\013") then raise Done ;
              let result = (String.uppercase line)^"\n"
              in ignore (ThreadUnix.write s_descr result 0 (String.length result))
         done
       with
```

```
          Done → ()
        | exn  → print_string (Printexc.to_string exn) ; print_newline()
    end ;;
Characters 273-286:
      method start () =  Thread.create (fun x -> self#run x ; self#stop x) ()
                         ^^^^^^^^^^^^^
Unbound value Thread.create
```

ここでも、継承と run メソッドの再定義により、新しいサービスを定義することができます。

この新しいバージョンのサーバを、protect_serv 関数を実行してテストすることができます。

```
# let go_serv () = let s = new serv_socket 1400 in s#run () ;;
# let protect_serv () = Unix.handle_unix_error go_serv () ;;
```

多段クライアント - サーバプログラミング

クライアント - サーバの関係は非対称的なのですが、サーバが他のサービスのクライアントになっても一向に構いません。このように通信の階層を作っていきます。典型的なクライアント - サーバアプリケーションとして次のようなものが考えられます。

- メールクライアントは使い易いユーザインタフェースを提供し、
- ワープロソフトが起動され、ユーザがインタラクトし、
- ワープロソフトがデータベースにアクセスします。

クライアント - サーバアプリケーションの目標の一つは、中心的なマシンの処理の負担を軽減することです。図 20.3 は二つの三段クライアント - サーバアーキテクチャを示しています。

各段は異なるマシンで実行されるかも知れません。ユーザインタフェースは、ユーザメールアプリケーションが走っているマシンで動作します。処理部分は何人かのユーザで共有されるマシンで実行され、さらにそれがリモートデータベースサーバへリクエストを送ります。アプリケーションによっては、処理部分がユーザメールアプリケーション、あるいはデータベースサーバの上でも動作するかも知れません。[4]

クライアント - サーバプログラムに関するコメント

これまでの節では、大文字化サービスのためのサーバをいくつか構築してきました。それぞれのサーバは異なるアプローチで実装されました。最初のサーバは Unix の fork 機構を利用しました。サーバを構築後、Unix、Windows、MacOS で提供されている telnet

4. 訳注：英訳が変なのでオリジナルの方を読んでみました。自信なし。

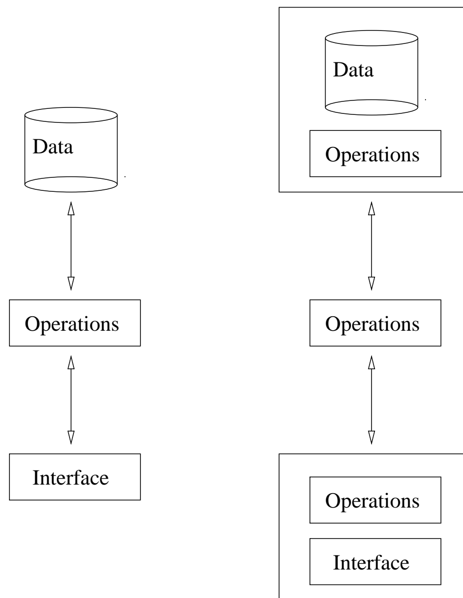


図 20.3: 二つのクライアント - サーバアーキテクチャ

クライアントで、構築したサーバをテストすることができました。その次に、最初の簡単なクライアントを作りました。この段階でクライアントとサーバのテストが可能になりました。クライアントは、複数の通信を同時に扱う場合もありえます。この例として、`client_par.exe` クライアントを作りました。これは fork を使って読み込みと書き込みとを分離したものでした。そして、新しい種類のサーバをスレッドを使って実装してみました。これは、サーバとクライアントが比較的独立していることをはっきりと示し、スレッドを使ったやり方に必要な、入出力に関する注意事項[5]を実践してみせるためでした。このサーバは二つの再利用可能なクラスとして構成されました。関数型プログラミングとオブジェクト指向プログラミングとの両方の要素のおかげで “定型的な” 再利用可能コードと、特化された処理コードとを分離することができるということは、注目に値します。

通信プロトコル

前節で述べられていた様々なクライアント - サーバ間の通信は、キャリッジリターンで終る文字列を送り、他の文字列を受け取るということで成り立っていました。どんなに単純でも、この通信パターンがプロトコルを定義するのです。より複雑な値、例えば、浮動小数点数や浮動小数点数の行列、算術式の木構造、関数クロージャ、オブジェクトなどをやりとりしたい場合は、それらの値をエンコードをするという問題が出てきます。実装言語やマシンアーキテクチャ、場合によっては OS などによって特徴付けられる通

5. 訳注：ちゃんと `ThreadUnix` ライブラリを使うということ。

信プログラムの性質に依存して多くの解法があります。整数は、マシンアーキテクチャによって、様々な方法で表現されます。（MSB が右側だったり左側だったり、タグビットを使ったり、マシンワードと同じサイズだったり。）異なるプログラム間で値をやりとりするためには、値の共通表現が必要です。これは**外部（external）**表現と呼ばれます[6]。レコードの様な、より構造的な値も、整数値のように外部表現がなくてはなりません。それでも、例えば、ある言語に他の言語にはないコンストラクト、例えば C 言語のビットフィールドのようなコンストラクトがあると、問題が出てきます。コードを含む関数オブジェクトや、オブジェクトをやりとりするには、また別の困難が生じます。コードの送信側と受信側におけるバイト単位での互換性、コードを動的にロードする機構の有無などです。一般則として、コードがもともと両側に存在することにすれば問題が簡単になります。送信されるのはコード自体ではなく、それを読み出すための情報となります。オブジェクトについては、インスタンス変数がオブジェクトの型と共にやりとりされます。これによりオブジェクトのメソッドを参照できます。関数クロージャについては、環境がコードのアドレスとともに送られます。これは、通信し合うプログラムが、全く同じ実行コードでなくてはならないことを意味します。

複雑に絡み合うデータのやりとりと、多くのプログラムの通信を同期する必要性から、別の困難も生じます。

まず、テクストプロトコルから始めて、確認応答と、リクエストとレスポンスとの間の時間制限について議論します。また、内部値をやりとりする困難についても言及します。これは、特に異なる言語で記述されたプログラム間の相互運用可能性と関連するからです。

テクストプロトコル

テクストプロトコルとは、ASCII 形式による通信のことです。実装が一番簡単で、移植性が非常に高いことから、最もよく使われます。プロトコルが複雑になってくると、実装が難しくなってきます。テクストプロトコルでは、通信フォーマットを記述する文法を定義します。この文法には、非常に多くの文法要素が盛り込まれることもあるのですが、それは送受信されるテクスト文字列の符合化や解釈の処理を行なう通信プログラム次第で決まります。

一般的に、ネットワークアプリケーションは、使用しているプロトコルとは別のレイヤのプロトコルを隠蔽します。これは、ブラウザを利用してウェブサイトとの通信を可能とする HTTP プロトコルが典型です。

HTTP プロトコル

“HTTP” という用語は広告でも良く見受けられます。それはウェブアプリケーションで使われる通信プロトコルのことです。このプロトコルの全ては W3 協会のページ、

リンク: http://www.w3.org

に掲載されています。このプロトコルは、ブラウザ（Communicator、Internet Explorer、Opera、など）からリクエストを送ったり、リクエストされたページの内容を返送するの

6. 例えば、C プログラムのためにデザインされた XDR 表現（eXternal Data Representation）

に使われます。ブラウザから送られるリクエストにはプロトコル名（HTTP）、マシン名（www.ufr-info-p6.jussieu.fr）リクエストされたページのパス（/Public/Localisation-/index.html）が含まれています。これらが一つになって、

```
http://www.ufr-info-p6.jussieu.fr/Public/Localisation/index.html
```

という URL（Uniform Resource Locator）を定義しています。このような URL がブラウザからリクエストされると、ブラウザと、指定されたサーバマシンの 80 番ポート（特に指定のない場合）で走っているサーバとの間の、ソケットを通じた接続が確立されます。そしてブラウザは HTTP フォーマットのリクエストを次のように送ります。

```
GET /index.html HTTP/1.0
```

サーバは、次のようなヘッダを付けて HTTP フォーマットでレスポンスします。

```
HTTP/1.1 200 OK
Date: Wed, 14 Jul 1999 22:07:48 GMT
Server: Apache/1.3.4 (Unix) PHP/3.0.6 AuthMySQL/2.20
Last-Modified: Thu, 10 Jun 1999 12:53:46 GMT

Accept-Ranges: bytes
Content-Length: 3663
Connection: close
Content-Type: text/html
```

このヘッダは、要求がアクセプトされた（code 200 OK）ということと、サーバの種類、ページの修正日時、ページのサイズ、続くコンテンツの型を示しています。プロトコル（HTTP/1.0）の GET コマンドを使うと、HTML ページのみが転送されます。次の telnet による接続で、実際に何が送られるのかを確認できます。

```
$ telnet www.ufr-info-p6.jussieu.fr 80
Trying 132.227.68.44...
Connected to triton.ufr-info-p6.jussieu.fr.
Escape character is '^]'.
GET


<!-- index.html -->
<HTML>
<HEAD>
<TITLE>Serveur de l'UFR d'Informatique de Pierre et Marie Curie</TITLE>
</HEAD>
<BODY>

<IMG SRC="/Icons/upmc.gif" ALT="logo-P6" ALIGN=LEFT HSPACE=30>
Unit&eacute; de Formation et de Recherche 922 - Informatique<BR>
```

```
Universit&eacute; Pierre et Marie Curie<BR>
4, place Jussieu<BR>
75252 PARIS Cedex 05, France<BR><P>
....
</BODY>
</HTML>
<!-- index.html -->

Connection closed by foreign host.
```

ページがコピーされたら、接続はクローズされます。基礎となるプロトコルは、解釈される言語ともどもテクストモードです。画像はページとともに送られるわけではないことに注意して下さい。HTML の文法を解析して、アンカーや画像を見つける（転送されてきたページの IMG タグを見る）のはブラウザの仕事です。この時点で、ブラウザは、HTML ソース中に画像が見つかるたびに、新しいリクエストを送ります。各画像毎に、新しい接続が張られます。画像は受信された時点で描画されるので、並列的に表示されてゆきます。

HTTP は十分に単純なのですが、情報は HTML で記述して送られます。この言語はちょっと複雑です。

確認応答と時間制限のあるプロトコル

プロトコルが複雑になってくると、メッセージの受信側が送信側にメッセージの無事到着とそれが文法的に正しいかどうかを伝えるということが有益となってきます。クライアントは、タスクを処理する前に、レスポンスを待ってブロックします。このリクエストを処理するサーバの部分が、メッセージの解釈で引っかかっている場合は単にリクエストを無視して終りではなく、クライアントにその状況を教えてあげなくてはなりません。HTTP プロトコルにはエラーコードの体系があります。正しいリクエストにはコード 200 が返ってきます。フォーマットから逸脱したリクエストや、閲覧権限のないページのリクエストに対してはエラーの種類によって、4xx や 5xx といったコードが返ってきます。これらのエラーコードによって、クライアントは何をすべきかが分かり、サーバ側も、出来事の詳細をログファイルに記録しておくことができます。

サーバは、整合性の無い状態に陥った場合でも、クライアントからの接続をアクセプトすることはできます。ただ、ソケットを通じてレスポンスを全く返さないかも知れないというリスクがあります。ブロック待ちを避けるために、レスポンスの送信に時間制限を設定するのが有効です。その時間が過ぎると、クライアントはサーバがレスポンスを返さないとみなします。そして、クライアントは他の仕事のため接続をクローズすることができます。このように WWW ブラウザは動作します。リクエストに対してある一定時間経ってもレスポンスが無い場合、ブラウザはその旨をユーザに伝えます。Objective Caml には、時間制限付の入出力機能があります。`Thread` ライブラリにある `wait_time_read` 関数と `wait_time_write` 関数は、ある一定時間内で一文字読み書きが可能になるまで、実行をサスペンドします。これらの関数は、入力としてファイルデスクリプタと時間制限を秒単位でとります。*`Unix.file_descr -> float -> bool`*. 制限時間を経過したら、`false` を返し、さもなくば I/O が処理されます.

内部表現のままの値の転送

内部値をそのまま転送するということが興味深いのは、それによりプロトコルを簡略化することができるからです。もはやデータをエンコードしたりテクストフォーマットにデコードする必要はなくなります。内部表現された値をそのまま送受信することの困難は、永続値を扱う困難と同様のものです。(228 ページ、Marshal ライブラリ参照。) 実際には、ファイルに対する値の読み書きは、同じ値をソケット上で送受信するのと変わりません。

関数値

二つの Objective Caml プログラム間で関数クロージャを転送する場合、関数クロージャ内のコードは送られず、環境とコードポインタのみが送られます。(335 ページ、図 12.9 参照。) この方針がうまくいくためには、サーバが同じコードを同じメモリ番地に配置している必要があります。つまり、全く同じプログラムが、サーバ、クライアントの両方で実行されることになります。しかし、その二つのプログラムが同時に異なる部分を実行していても、何ら問題はありません。行列計算サービスを、計算に必要なデータを含む環境をもった関数クロージャを送る、という風に作り変えることができます。関数クロージャを受信すると、サーバはそのクロージャを () に適用し計算が開始されます。

異言語間相互運用

テクストプロトコルの良いところは、クライアントやサーバの実装言語に依存しないというところです。実際のところ、ASCII コードはどんなプログラミング言語でも扱えます。よって、送られた文字列の文法的な解釈はクライアントやサーバの責任ということになります。そのようなオープンなプロトコルの例の一つに、RoboCup と呼ばれるサッカープレイヤのシミュレーションがあります。

サッカーロボット

サッカーチームが対戦します。チームの各メンバはレフリーサーバのクライアントとなります。同じチームのプレイヤ間では直接的な通信ができません。情報はサーバを通じて送らなくてはなりません。サーバは対話を再送します。サーバはプレイヤの位置に応じてフィールドの一部分を示します。これらの通信は全てテクストプロトコルで行なわれます。次のウェブページに、プロトコル、サーバ、幾つかのクライアントが掲載されています。

リンク: http://www.robocup.org/

サーバは C で書かれています。クライアントは、C、C++、SmallTalk、Objective Caml、など、様々な言語で書かれています。一つのチームでフィールドプレイヤが別々の言語で書かれていても構いません。

このプロトコルは、異なる実装言語で書かれたプログラム間の相互運用のニーズに応えています。比較的シンプルなのですが、各言語ファミリーごとに個別の文法解析器が必要となります。

演習問題

ここに挙げる演習問題では、様々な種類の分散アプリケーションを試してみることができます。最初の演習では、クライアントマシンの時刻をセットするネットワークサービスを新しく提供します。二つ目の演習では、ある計算を分散させるために、異なるマシン上のリソースをいかに利用するかを示します。

サービス： 時計

この演習では、クライアントに時刻情報を提供する “時計” サービスを実装します。ネットワーク上の様々なマシンの時刻をセットするための参照マシンを用意するというのがアイデアです。

1. 日、月、時間、秒を含む日付情報を転送するためのプロトコルを定義する。
2. この章で紹介した汎用サーバのどれか一つを利用して、サービス関数、あるいはサービスクラスを書く。サービスは日付情報を、アクセプトした接続のそれぞれに送り、ソケットをクローズする。
3. 一時間毎に時計をセットするクライアントを書く。
4. リクエストが送られるときに時間誤差を記録する。

ネットワーク コーヒー自動販売機

飲料水の自動販売機をシミュレートする、ちょっとしたサービスを構築します。クライアントとサービスとの間のプロトコルの概要は次のようになります。

- クライアントは接続すると、購入できる飲物のリストを受けとります。
- 次に、どの飲物を選んだかをサーバに送ります。
- サーバはその飲物の料金をレスポンスとして返します。
- クライアントはその料金を送るか、あるいはいくらか適当な額を送ります。
- サーバは選ばれた飲物の名前を返送し、支払われた額を示します。

サーバは、リクエストが理解できなかった場合や、料金が不足している場合などには、エラーメッセージを返すのもよいでしょう。クライアントのリクエストには必ず情報が一つだけ含まれます。

クライアントとサーバとのやりとりは、文字列の形で行なわれます。メッセージの要素は二つのピリオドで区切られ、全ての文字列は`:$\n`で終ります。

サービス関数は、コマンド渡すためのキューと、飲物や値段を取り出すためのハッシュ表を使ってコーヒー自動販売機と通信します。

この演習では、ソケット、ライト級プロセス（ちょっとした並行性のため)、オブジェクトを使うことになるでしょう。

1. `ThreadUnix` のプリミティブを使って `establish_server` 関数を書き直します。

2. 二つの関数 get_request と send_answer を書きます。一つ目の関数がリクエストを読み込み、それをエンコードして、二つ目の関数がフォーマットし文字列のリストで始まるレスポンスを送ります。
3. 未処理のコマンドを処理する cmd_fifo クラスを書きます。コマンドはそれぞれユニークな番号を割り当てられます。このため、num_cmd_gen クラスを実装します。
4. 自動販売機に飲物をストックするための ready_table クラスを書きます。
5. 自動販売機をモデル化する machine クラスを書きます。 このクラスは run メソッドを持ちます。このメソッドは、コマンドを待ちそれを実行する、というループを、飲物が続く限り続けます。*drink_descr* 型を定義します。各飲物に対して、名前、ストック量、未処理コマンドを処理した場合の残量、値段を示します。パラメタとして渡された条件を満たす表中最初の要素のインデックスを返す補助関数 array_index も使えます。
6. サービス関数 waiter を書きます。
7. メイン関数 main を書きます。この関数は、サービスのポート番号をコマンドラインから得て、いくつか初期化処理を行ないます。特に、コーヒー自動販売機は一つのプロセスで実行されます。

まとめ

この章では、分散プログラミングがもたらす新しい可能性について紹介してきました。プログラム間の通信は、低レベルのインターネットプロトコルで使われる、ソケットという基本メカニズムで実現されます。クライアントとサーバの動作モデルは非対称的です。クライアントとサーバ間の通信は、ほとんどの場合はプレインテクストを使ったなんらかのプロトコルを使用します。関数型プログラミングとオブジェクト指向プログラミングにより、分散アプリケーションを容易に構築することができます。クライアント - サーバモデルは、二、三段に多段化して、その段へのタスクの分散度合を調整すれば、様々なソフトウェアアーキテクチャに適合します。

さらに進んだ話題

異なるマシン上の Objective Caml プログラム間の通信を、よりリッチにすることができます。構文解析ユーティリティにより、テクストプロトコルの利用は大いに促進されました。(11 章参照) Marshal ライブラリによる永続化メカニズム (8 章参照) により、複雑なデータ (通信しているプログラムが同じなら関数値も含む) を内部フォーマットのまま送ることができます。このメカニズムの主な欠点は、クラスのインスタンスを送れないことです。この問題に対する一つの解決策は、オブジェクトの転送やリモートメソッドの実行に ORB (Object Request Broker) を利用することです。このアーキテクチャは既に多くのオブジェクト指向言語に CORBA (Common ORB Architecture) 規格という形で備わっています。この 1990 年に初公開された、OMG (Object Management Group) 発の規格では、リモートオブジェクトの使用が許されており、それはクラスの実装言語に依存しません。

リンク: http://www.omg.org

CORBA の主要な機能は、リモートプログラムにオブジェクトを送る機能や、特に、オブジェクトのインスタンス変数を更新できるメソッドを、ネットワークの様々な場所からを呼び出せる機能です。さらに、この規格では、リモートオブジェクトの実装言語を問いません。このため、ORB は、IDL（Interface Definition Language）と呼ばれるインタフェース記述言語を備えています。それは、Objective Caml と C のインタフェースのための `CAMLIDL` 風です。さしあたって、Objective Caml で動作する ORB はありませんが、IDL がクラスを持つオブジェクト指向言語の抽象化となっているので、作ることは可能です。簡素化のため CORBA は、ソフトウェアバス（IIOP）を備えています。これにより、リモートデータの転送やアドレッシングが可能となります。

ネットワークの様々な場所で同一オブジェクトを参照する機能は、分散共有メモリをシミュレートします。これは、自動ガーベージコレクションの問題と無縁ではありません。

リモートオブジェクトを参照する能力があるからといって、コードが転送されるわけではありません。サーバに存在するあるクラスのインスタンスのコピーを受け取るだけです。クライアント - サーバアプリケーションによっては、コードのダイナミックロード（Java アプレットのように）が必要かもしれません。リモードコードのダイナミックロードの興味深い例の一つに、François Rouaix が Objective Caml で書いた MMM ブラウザがあります。

リンク: http://caml.inria.fr/~rouaix/mmm/

このブラウザは普通に Web ページの閲覧に使えますし、さらに Objective Caml アプレットをサーバからロードし、グラフィックウィンドウの中でで実行できます。

21

アプリケーション

The first application is really a toolbox to facilitate the construction of client-server applications which transmit Objective Caml values. To build an application using the toolbox, one need only implement serialization functions for the values to be transmitted, then apply a functor to obtain an abstract class for the server, then add the application's processing function by means of inheritance.

The second application revisits the robot simulation, presented on page 554, and adapts it to the client-server model. The server represents the world in which the robot clients move around. We thus simulate distributed memory shared by a group of clients possibly located on various machines on the network.

The third application is an implementation of some small HTTP servers (called **servlets**). A server knows how to respond to an HTTP request such as a request to retrieve an HTML page. Moreover, it is possible to pass values in these requests using the CGI format of HTTP servers. We will use this functionality right away to construct a server for requests on the association database, described on page 148. As a client, we will use a Web browser to which we will send an initial page containing the query form.

Client-server Toolbox

We present a collection of modules to enable client-server interactions among Objective Caml programs. This toolbox will be used in the two applications that follow.

A client-server application differs from others in the protocol that it uses and in the processing that it associates with the protocol. Otherwise, all such applications use very similar mechanisms: waiting for a connection, starting a separate process to handle the connection, and reading and writing sockets.

Taking advantage of Objective Caml's ability to combine modular genericity and extension of objects, we will create a collection of functors which take as argument a

communications protocol and produce generic classes implementing the mechanisms of clients and of servers. We can then subclass these to obtain the particular processing we need.

Protocols

A communications protocol is a type of data that can be translated into a sequence of characters and transmitted from one machine to another via a socket. This can be described using a signature.

```
# module type PROTOCOL =
    sig
      type t
      val to_string : t → string
      val of_string : string → t
    end ;;
```

The signature requires that the data type be monomorphic; yet we can choose a data type as complex as we wish, as long as we can translate it to a sequence of characters and back. In particular, nothing prevents us from using objects as our data.

```
# module Integer =
    struct
      class integer x =
        object
          val v = x
          method x = v
          method str = string_of_int v
        end
      type t = integer
      let to_string o = o#str
      let of_string s = new integer (int_of_string s)
    end ;;
```

By making some restrictions on the types of data to be manipulated, we can use the module `Marshal`, described on page 229, to define the translation functions.

```
# module Make_Protocol = functor ( T : sig type t end ) →
    struct
      type t = T.t
      let to_string (x:t) = Marshal.to_string x [Marshal.Closures]
      let of_string s = (Marshal.from_string s 0 : t)
    end ;;
```

Communication

Since a protocol is a type of value that can be translated into a sequence of characters, we can make these values persistent and store them in a file.

The only difficulty in reading such a value from a file when we do not know its type is that *a priori* we do not know the size of the data in question. And since the file in question is in fact a socket, we cannot simply check an end of file marker. To solve this problem, we will write the size of the data, as a number of characters, before the data itself. The first twelve characters will contain the size, padded with spaces.

The functor `Com` takes as its parameter a module with signature `PROTOCOL` and defines the functions for transmitting and receiving values encoded using the protocol.

```
# module Com = functor (P : PROTOCOL) →
    struct
      let send fd m =
        let mes = P.to_string m in
        let l = (string_of_int (String.length mes)) in
        let buffer = String.make 12 ' ' in
          for i=0 to (String.length l)-1 do buffer.[i] <- l.[i] done ;
          ignore (ThreadUnix.write fd buffer 0 12) ;
          ignore (ThreadUnix.write fd mes 0 (String.length mes))

      let receive fd =
        let buffer = String.make 12 ' '
        in
          ignore (ThreadUnix.read fd buffer 0 12) ;
          let l = let i = ref 0
          in while (buffer.[!i]<>' ') do incr i done ;
             int_of_string (String.sub buffer 0 !i)
        in
          let buffer = String.create l
          in ignore (ThreadUnix.read fd buffer 0 l) ;
             P.of_string buffer
    end ;;
Characters 283-299:
          ignore (ThreadUnix.write fd buffer 0 12) ;
                  ^^^^^^^^^^^^^^^^
Unbound value ThreadUnix.write
```

Note that we use the functions `read` and `write` from module `ThreadUnix` and not those from module `Unix`; this will permit us to use our functions in a thread without blocking the execution of other processes.

Server

A server is built as an abstract class parameterized by the type of data in the protocol. Its constructor takes as arguments a port number and the maximum number of simultaneous connections allowed. The method for processing a request is abstract; it must be implemented in a subclass of `server` to obtain a concrete class.

```
# module Server = functor (P : PROTOCOL) →
    struct
```

```
  module Com = Com (P)

  class virtual ['a] server p np =
    object (s)
      constraint 'a = P.t
      val port_num = p
      val nb_pending = np
      val sock = ThreadUnix.socket Unix.PF_INET Unix.SOCK_STREAM 0

      method start =
        let host = Unix.gethostbyname (Unix.gethostname()) in
        let h_addr = host.Unix.h_addr_list.(0) in
        let sock_addr = Unix.ADDR_INET(h_addr, port_num) in
          Unix.bind sock sock_addr ;
          Unix.listen sock nb_pending ;
          while true do
            let (service_sock, client_sock_addr) = ThreadUnix.accept sock
            in ignore (Thread.create s#process service_sock)
          done
      method send = Com.send
      method receive = Com.receive
      method virtual process : Unix.file_descr → unit
    end
end ;;
```

In order to show these ideas in use, let us revisit the CAPITAL service, adding the capability of sending lists of strings.

```
# type message = Str of string | LStr of string list ;;
# module Cap_Protocol = Make_Protocol (struct type t=message end) ;;
# module Cap_Server = Server (Cap_Protocol) ;;

# class cap_server p np =
    object (self)
      inherit [message] Cap_Server.server p np
      method process fd =
        match self#receive fd with
            Str s  → self#send fd (Str (String.uppercase s)) ;
                     Unix.close fd
          | LStr l → self#send fd (LStr (List.map String.uppercase l)) ;
                     Unix.close fd
    end ;;
Characters 54-81:
      inherit [message] Cap_Server.server p np
              ^^^^^^^^^^^^^^^^^^^^^^^^^^
Unbound class
Cap_Server.server
```

The processing consists of receiving a request, examining it, processing it and sending the result. The functor allows us to concentrate on this processing while constructing the server; the rest is generic. However, if we wanted a different mechanism, such as

for example using acknowledgements, nothing would prevent us from redefining the inherited methods for communication.

Client

To construct clients using a given protocol, we define three general-purpose functions:

- `connect`: establishes a connection with a server; it takes the address (IP address and port number) and returns a file descriptor corresponding to a socket connected to the server.
- `emit_simple`: opens a connection, sends a message and closes the connection.
- `emit_answer`: same as `emit_simple`, but waits for the server's response before closing the connection.

```
# module Client = functor (P : PROTOCOL) →
    struct
     module Com = Com (P)

     let connect addr port =
       let sock = ThreadUnix.socket Unix.PF_INET Unix.SOCK_STREAM 0
       and in_addr = (Unix.gethostbyname addr).Unix.h_addr_list.(0)
       in ThreadUnix.connect sock (Unix.ADDR_INET(in_addr, port)) ;
          sock

     let emit_simple addr port mes =
       let sock = connect addr port
       in Com.send sock mes ;  Unix.close sock

     let emit_answer addr port mes =
       let sock = connect addr port
       in Com.send sock mes ;
          let res = Com.receive sock
          in Unix.close sock ; res
    end ;;
Characters 70-73:
      module Com = Com (P)
                   ^^^
Unbound module Com
```

The last two functions are of a higher level than the first: the mechanism linking the client and the server does not appear. The caller of `emit_answer` does not even need to know that the computation it is requesting is carried out by a remote machine. As far as the caller is concerned, it invokes a function that is represented by an address and port, with an argument which is the message to be sent, and a value is returned to it. The distributed aspect can seem entirely hypothetical.

A client of the CAPITAL service is extremely easy to construct. Assume that the `boulmich` machine provides the service on port number 12345; then the function `list_uppercase` can be defined by means of a call to the service.

```
# let list_uppercase l =
    let module Cap_client = Client (Cap_Protocol)
    in match Cap_client.emit_answer "boulmich" 12345 (LStr l)
       with Str x → [x]
         | LStr x → x ;;
Characters 51-57:
    let module Cap_client = Client (Cap_Protocol)
                            ^^^^^^
Unbound module Client
```

To Learn More

The first improvement to be made to our toolbox is some error handling, which has been totally absent so far. Recovery from exceptions which arise from a broken connection, and a mechanism for retrying, would be most welcome.

In the same vein, the client and the server would benefit from a timeout mechanism which would make it possible to limit the time to wait for a response.

Because we have constructed the generic server as a class, which moreover is parameterized by the type of data to be transmitted over the network, it is easy to extend it to augment or modify its behavior in order to implement any desired improvements.

Another approach is to enrich the communication protocols. One can for example add requests for acknowledgement to the protocol, or accompany each request by a checksum allowing verification that the network has not corrupted the data.

The Robots of Dawn

As we promised in the last application of the third part (page 554), we will now revisit the problem of robots in order to treat it in a distributed framework where the world is a server and where each robot is an independent process capable of being executed on a remote machine.

This application is a good summary of the possibilities of the Objective Caml language because we will utilize and combine the majority of its features. In addition to the distributed model which is imposed on us by the exercise, we will make use of concurrency to construct a server in which multiple connections will be handled independently while all sharing a single memory representation of the "world". All access to and modification of the state of affairs of the world will therefore have to be protected by critical sections.

In order to reuse as much as possible the code that we have already built for robots in one section, and the client-server architecture of another section, we will use functors and inheritance of classes at the same time.

This application is quite minimal, but we will see that its architecture lends itself particularly well to extensions in multiple directions.

World-Server

We take a representation of the world similar to that which we developed in Part III. The world is a grid of finite size, and each cell of the grid can be occupied by only one robot. A robot is identified by its name and by its position; the world is determined by its size and by the robots that live in it. This information is represented by the following types:

```
# type position = { x:int ; y:int } ;;

# type robot_info = { name : string ; mutable pos : position }
  type world_info = { length : int ; width  : int ;
                      mutable robots : robot_info list } ;;
```

The world will have to serve two sorts of clients:

- passive clients which simply observe the positions of various robots. They will allow us to build the clients in charge of displays. We will call them **spies**.
- active clients, able to ask the server to move robots and thus modify its state.

These two categories of clients and their behavior will determine the collection of messages exchanged by the server and clients.

When a client connects, it declares itself passive (`Spy`) or active (`Enter`). A spy receives as response to its connection the global state of the world. Then, it is kept informed of all changes. However, it cannot submit any requests. A robot which connects must supply its characteristics (its name and its initial position); the world then confirms its arrival. Then, it can request information: its own position (`GetPos`) or the list of robots that surround it (`Look`). It can also instruct the world to move it. The protocol of requests to the world from distributed robots is represented by the following type:

```
# type query =
    | Spy                    (* initial declaration requests *)
    | Enter of robot_info

    | Move of position       (* robot requests *)
    | GetPos
    | Look of int

    | World of world_info    (* messages delivered by the world *)
    | Pos of robot_info
    | Exit of robot_info ;;
```

From this protocol, using the functors from the "distributed toolbox" of the previous chapter, we immediately derive the generic server.

```
# module Pquery = Make_Protocol (struct type t = query end ) ;;
```

```
# module Squery = Server (Pquery) ;;
```

Now we need only specify the behavior of the server by implementing the method `process` to handle both the data that represent the world and the data for managing connections.

More precisely, the server contains a variable `world` (of type *world_info*) which is protected by the lock `sem` (of type *Mutex.t*). It also contains a variable `spies` which is a list of queues of messages to send to observers, with one queue per spy. To activate the processes in charge of sending these messages, the server also maintains a signal (of type *Condition.t*).

We provide an auxiliary function `dist` to calculate the distance between two positions:

```
# let dist p q = max (abs (p.x-q.x)) (abs (p.y-q.y)) ;;
val dist : position -> position -> int = <fun>
```

The function `critical` encapsulates the calculation of a value within a critical section:

```
# let critical m f a =
    Mutex.lock m ; let r = f a in Mutex.unlock m ; r ;;
Characters 25-35:
    Mutex.lock m ; let r = f a in Mutex.unlock m ; r ;;
    ^^^^^^^^^^
Unbound value Mutex.lock
```

Here is the definition of the class `server` implementing the world-server. It is long, but we will follow it up with a step-by-step explanation.

```
# class  server l w n np =
    object (self)
      inherit [query] Squery.server n np
      val world = { length=l ; width=w ; robots=[] }
      val sem = Mutex.create ()
      val mutable spies = []
      val signal = Condition.create ()

      method lock = Mutex.lock sem
      method unlock = Mutex.unlock sem

      method legal_pos p = p.x>=0 && p.x<l && p.y>=0 && p.y<w

      method free_pos p =
        let is_not_here r = r.pos.x<>p.x || r.pos.y<>p.y
        in critical sem (List.for_all is_not_here) world.robots

      method legal_move r p =
        let dist1 p = (dist r.pos p) <= 1
        in (critical sem dist1 p) && self#legal_pos p && self#free_pos p
```

```
method queue_message mes =
  List.iter (Queue.add mes) spies ;
  Condition.broadcast signal

method trace_loop s q =
  let foo = Mutex.create () in
  let f () =
    try
      spies <- q :: spies ;
      self#send s (World world) ;
      while true do
        while Queue.length q = 0 do Condition.wait signal foo done ;
        self#send s (Queue.take q)
      done
    with _ → spies <- List.filter ((!=) q) spies ;
             Unix.close s
  in ignore (Thread.create f ())

method remove_robot r =
  self#lock ;
  world.robots <- List.filter ((<>) r) world.robots ;
  self#queue_message (Exit {r with name=r.name}) ;
  self#unlock

method try_move_robot r p =
  if self#legal_move r p
  then begin
         self#lock ;
         r.pos <- p ;
         self#queue_message (Pos {r with name=r.name}) ;
         self#unlock
       end

method process_robot s r =
  let f () =
    try
      world.robots <- r :: world.robots ;
      self#send s (Pos r) ;
      self#queue_message (Pos r) ;
      while true do
        Thread.delay 0.5 ;
        match self#receive s with
            Move p → self#try_move_robot r p
          | GetPos → self#send s (Pos r)
          | Look d →
              self#lock ;
              let dist p = max (abs (p.x-r.pos.x)) (abs (p.y-r.pos.y)) in
              let l = List.filter (fun x → (dist x.pos)<=d) world.robots
              in self#send s (World { world with robots = l }) ;
                 self#unlock
```

```
                | _ → ()
              done
            with  _ → self#unlock ;
                      self#remove_robot r ;
                      Unix.close s
          in ignore (Thread.create f ())

        method process s =
          match self#receive s with
              Spy → self#trace_loop s (Queue.create ())
            | Enter r →
               ( if not (self#legal_pos r.pos && self#free_pos r.pos) then
                  let i = ref 0 and j = ref 0 in
                  ( try
                      for x=0 to l do
                        for y=0 to w do
                          let p = { x=x ; y=y }
                          in if self#legal_pos p && self#free_pos p
                             then ( i:=x ; j:=y; failwith "process" )
                        done done ;
                      Unix.close s
                    with Failure "process" → r.pos <- { x= !i ; y= !j }  )) ;
                    self#process_robot s r
            | _ → Unix.close s

      end ;;
Characters 55-76:
Unbound class
Squery.server
```

The method `process` starts out by distinguishing between the two types of client. Depending on whether the client is active or passive, it invokes a processing method called: `trace_loop` for an observer, `process_robot` for a robot. In the second case, it checks that the initial position proposed by the client is compatible with the state of the world; if not, it finds a valid initial position. The remainder of the code can be divided into four categories:

1. **General methods**: these are methods which we developed in Part III for general worlds. Mainly, it is a matter of verifying that a displacement is legal for a given robot.
2. **Management of observers**: each observer is associated with a socket through which it is sent data, with a queue containing all the messages which have not yet been sent to it, and with a process. The method `trace_loop` is an infinite loop that empties the queue of messages by sending them; it goes to sleep when the queue is empty. The queues are filled, all at the same time, by the method `queue_message`. Note that after appending a message, the activation signal is sent to all processes.
3. **Management of robots**: here again, each robot is associated with a dedicated process. The method `process_robot` is an infinite loop: it waits for a request,

processes it, and responds if necessary. Then it resumes waiting for the next request. Note that it is these robot-management methods which issue calls to the method `queue_message` when the state of the world has been modified. If the connection with a robot is lost—that is, if an exception is raised while waiting for a request—the robot is considered to have terminated and its departure is signaled to the observers.

4. **Inherited methods**: these are the methods of the generic server obtained by application of the functor `Server` to the protocol of our application.

Observer-client

The functor `Client` gives us generic functions for connecting with a server according to the particular protocol that concerns us here.

```
# module Cquery = Client (Pquery) ;;
Characters 17-23:
  module Cquery = Client (Pquery) ;;
                  ^^^^^^
Unbound module Client
```

The behavior of a spy is simple: it connects to the server and displays the information that the server sends it. The spy includes three display functions which we provide below:

```
# let display_robot r =
     Printf.printf "The robot %s is located at (%d,%d)\n" r.name r.pos.x r.pos.y ;
     flush stdout ;;
val display_robot : robot_info -> unit = <fun>

# let display_exit r =   Printf.printf "The robot %s has terminated\n" r.name ;
     flush stdout ;;
val display_exit : robot_info -> unit = <fun>

# let display_world w =
     Printf.printf "The world is a grid of size %d by %d \n" w.length w.width ;
     List.iter display_robot w.robots ;
     flush stdout ;;
val display_world : world_info -> unit = <fun>
```

The primary function of the spy-client is:

```
# let trace_client name port =
     let sock = Cquery.connect name port
     in Cquery.Com.send sock Spy ;
        ( match Cquery.Com.receive sock with
              World w → display_world w
            | _ → failwith "the server did not follow the protocol" ) ;
        while true do
          match Cquery.Com.receive sock with
              Pos r → display_robot r
            | Exit r → display_exit r
```

```
        |_ → failwith "the server did not follow the protocol"
      done ;;
Characters 43-57:
    let sock = Cquery.connect name port
               ^^^^^^^^^^^^^^
Unbound value Cquery.connect
```

There are two ways of constructing a graphical display. The first is simple but not very efficient: since the server sends the complete set of information when a connection is established, one can simply open a new connection at regular intervals, display the world in its entirety, and close the connection. The other approach involves using the information sent by the server to maintain a copy of the state of the world. It is then easy to display only the modifications to the state upon reception of messages. It is this second solution which we have implemented.

Robot-Client

As we defined them in the previous chapter (cf. page 554), the robots conform to the following signature.

```
# module type ROBOT =
    sig
      class robot : int → int →
        object
          val mutable i : int
          val mutable j : int
          method get_pos : int * int
          method next_pos : unit → int * int
          method set_pos : int * int → unit
        end
      end ;;
```

The part that we wish to save from the various classes is that which necessarily varies from one type of robot to another and which defines its behavior: the method `next_pos`.

In addition, we need a method for connecting the robot to the world (`start`) and a loop that alternately calculates a new position and communicates with the server to submit the chosen position.

We define a functor which, when given a class implementing a virtual robot (that is, conforming to the signature `ROBOT`), creates, by inheritance, a new class containing the proper methods to make an autonomous client out of the robot.

```
# module RobotClient (R : ROBOT) =
    struct
      class robot robname x y hostname port =
```

```
        object (self)
          inherit R.robot x y as super
          val mutable socket = Unix.stderr
          val mutable rob = { name=robname ; pos={x=x;y=y} }

          method private adjust_pos r =
            rob.pos <- r.pos ; i <- r.pos.x ; j <- r.pos.y

          method get_pos =
            Cquery.Com.send socket GetPos ;
            match Cquery.Com.receive socket with
                Pos r → self#adjust_pos r ; super#get_pos
              | _ → failwith "the server did not follow the protocol"

          method set_pos =
            failwith "the method set_pos cannot be used"

          method start =
            socket <- Cquery.connect hostname port ;
            Cquery.Com.send socket (Enter rob) ;
            match Cquery.Com.receive socket with
                Pos r → self#adjust_pos r ; self#run
              | _ → failwith "the server did not follow the protocol"

          method run =
            while true do
              let (x,y) = self#next_pos ()
              in Cquery.Com.send socket (Move {x=x;y=y}) ;
                 ignore (self#get_pos)
            done
        end
    end ;;
Characters 380-395:
Unbound value Cquery.Com.send
```

Notice that the method `get_pos` has been redefined as a query to the server: the instance variables `i` and `j` are not reliable, because they can be modified without the consent of the world. For the same reason, the use of `set_pos` has been made invalid: calling it will always raise an exception. This policy may seem severe, but it's a good bet that if this method were used by `next_pos` then a discrepancy would appear between the real position (as known by the server) and the supposed position (as known by the client).

We use the functor `RobotClient` to create various classes corresponding to the various robots.

```
# module Fix = RobotClient (struct class robot = fix_robot end) ;;
# module Crazy = RobotClient (struct class robot = crazy_robot end) ;;
# module Obstinate = RobotClient (struct class robot = obstinate_robot end) ;;
```

The following small program provides a way to launch the server and the various clients from the command line. The argument passed to the program specifies which one to launch.

```
# let port = 1200 in
    if Array.length Sys.argv >=2 then
      match Sys.argv.(1) with
        "1" → let s = new server 25 30 port 10 in s#start
      | "2" → trace_client "localhost" port
      | "3" → let o = new Fix.robot "fix" 10 10 "localhost" port in o#start
      | "4" → let o = new Crazy.robot "crazy" 10 10 "localhost" port in o#start
      | "5" → let o = new Obstinate.robot "obstinate" 10 10 "localhost" port
            in o#start
      | _ → () ;;
```

To Learn More

The world of robots stimulates the imagination. With the elements already given here, one can easily create an "intelligent robot" which is both a robot and a spy. This allows the various inhabitants of the world to cooperate. One can then extend the application to obtain a small action game like "chickens-foxes-snakes" in which the foxes chase the chickens, the snakes chase the foxes and the chickens eat the snakes.

HTTP Servlets

A **servlet** is a "module" that can be integrated into a server application to respond to client requests. Although a servlet need not use a specific protocol, we will use the HTTP protocol for communication (see figure 21.1). In practice, the term servlet refers to an HTTP servlet.

The classic method of constructing dynamic HTML pages on a server is to use CGI (Common Gateway Interface) commands. These take as argument a URL which can contain data coming from an HTML form. The execution then produces a new HTML page which is sent to the client. The following links describe the HTTP and CGI protocols.

リンク: http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc1945.html

リンク: http://hoohoo.ncsa.uiuc.edu/docs/cgi/overview.html

It is a slightly heavyweight mechanism because it launches a new program for each request.

HTTP servlets are launched just once, and can can decode arguments in CGI format to execute a request. Servlets can take advantage of the Web browser's capabilities to construct a graphical interface for an application.

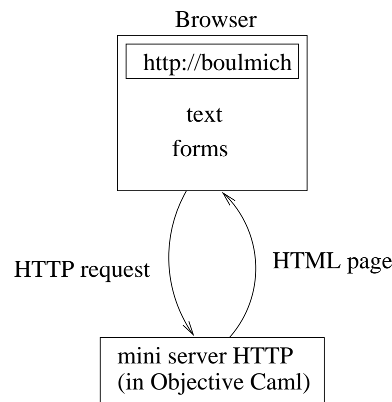


図 21.1: communication between a browser and an Objective Camlserver

In this section we will define a server for the HTTP protocol. We will not handle the entire specification of the protocol, but instead will limit ourselves to those functions necessary for the implementation of a server that mimics the behavior of a CGI application.

At an earlier time, we defined a generic server module `Gsd`. Now we will give the code to create an application of this generic server for processing part of the HTTP protocol.

HTTP and CGI Formats

We want to obtain a server that imitates the behavior of a CGI application. One of the first tasks is to decode the format of HTTP requests with CGI extensions for argument passing.

The clients of this server can be browsers such as Netscape or Internet Explorer.

Receiving Requests

Requests in the HTTP protocol have essentially three components: a method, a URL and some data. The data must follow a particular format.

In this section we will construct a collection of functions for reading, decomposing and decoding the components of a request. These functions can raise the exception:

```
# exception Http_error of string ;;
exception Http_error of string
```

Decoding The function `decode`, which uses the helper function `rep_xcode`, attempts to restore the characters which have been encoded by the HTTP client: spaces (which

have been replaced by +), and certain reserved characters which have been replaced by their hexadecimal code.

```
# let rec rep_xcode s i =
    let xs = "0x00" in
      String.blit s (i+1) xs 2 2;
      String.set s i (char_of_int (int_of_string xs));
      String.blit s (i+3) s (i+1) ((String.length s)-(i+3));
      String.set s ((String.length s)-2) '\000';
      Printf.printf"rep_xcode1(%s)\n" s ;;
val rep_xcode : string -> int -> unit = <fun>

# exception End_of_decode of string ;;
exception End_of_decode of string

# let decode s =
    try
      for i=0 to pred(String.length s) do
        match s.[i] with
          '+' → s.[i] <- ' '
        | '%' → rep_xcode s i
        | '\000' → raise (End_of_decode (String.sub s 0 i))
        | _ → ()
      done;
      s
    with
     End_of_decode s → s ;;
val decode : string -> string = <fun>
```

String manipulation functions The module `String_plus` contains some functions for taking apart character strings:

- `prefix` and `suffix`, which extract the substrings to either side of an index;
- `split`, which returns the list of substrings determined by a separator character;
- `unsplit`, which concatenates a list of strings, inserting separator characters between them.

```
# module String_plus =
   struct
    let prefix s n =
     try String.sub s 0 n
     with Invalid_argument("String.sub") → s

    let suffix s i =
     try String.sub s i ((String.length s)-i)
     with Invalid_argument("String.sub") → ""
```

```
  let rec split c s =
   try
    let i = String.index s c in
    let s1, s2 = prefix s i, suffix s (i+1) in
     s1 :: (split c s2)
   with
    Not_found → [s]

  let unsplit c ss =
   let f s1 s2 = match s2 with "" → s1 | _ → s1^(Char.escaped c)^s2 in
    List.fold_right f ss ""
 end ;;
```

Decomposing data from a form Requests typically arise from an HTML page containing a form. The contents of the form are transmitted as a character string containing the names and values associated with the fields of the form. The function `get_field_pair` transforms such a string into an association list.

```
# let get_field_pair s =
    match String_plus.split '=' s with
      [n;v] → n,v
    | _ → raise (Http_error ("Bad field format : "^s)) ;;
val get_field_pair : string -> string * string = <fun>

# let get_form_content s =
    let ss = String_plus.split '&' s in
      List.map get_field_pair ss ;;
val get_form_content : string -> (string * string) list = <fun>
```

Reading and decomposing The function `get_query` extracts the method and the URL from a request and stores them in an array of character strings. One can thus use a standard CGI application which retrieves its arguments from the array of command-line arguments. The function `get_query` uses the auxiliary function `get`. We arbitrarily limit requests to a maximum size of 2555 characters.

```
# let get =
    let buff_size = 2555 in
      let buff = String.create buff_size in
        (fun ic → String.sub buff 0 (input ic buff 0 buff_size)) ;;
val get : in_channel -> string = <fun>

# let query_string http_frame =
    try
      let i0 = String.index http_frame ' ' in
      let q0 = String_plus.prefix http_frame i0 in
        match q0 with
          "GET"
          → begin
               let i1 = succ i0 in
```

```
              let i2 = String.index_from http_frame i1 ' ' in
              let q = String.sub http_frame i1 (i2-i1) in
              try
                let i = String.index q '?' in
                let q1 = String_plus.prefix q i in
                let q = String_plus.suffix q (succ i) in
                  Array.of_list (q0::q1::(String_plus.split ' ' (decode q)))
              with
                Not_found → [|q0;q|]
             end
        | _ → raise (Http_error ("Unsupported method: "^q0))
    with e → raise (Http_error ("Unknown request: "^http_frame)) ;;
val query_string : string -> string array = <fun>

# let get_query_string ic =
    let http_frame = get ic in
      query_string http_frame;;
val get_query_string : in_channel -> string array = <fun>
```

The Server

To obtain a CGI pseudo-server, able to process only the `GET` method, we write the class `http_servlet`, whose argument `fun_serv` is a function for processing HTTP requests such as might have been written for a CGI application.

```
# module Text_Server = Server (struct type t = string
                                      let to_string x = x
                                      let of_string x = x
                               end);;

# module P_Text_Server (P : PROTOCOL) =
  struct
    module Internal_Server = Server (P)

    class http_servlet  n np fun_serv =
      object(self)
        inherit [P.t] Internal_Server.server n np

        method receive_h fd =
          let ic = Unix.in_channel_of_descr fd in
            input_line ic

        method process fd =
          let oc = Unix.out_channel_of_descr fd in (
            try
              let request = self#receive_h fd in
                let args = query_string request in
                  fun_serv oc args;
            with
              Http_error s → Printf.fprintf oc "HTTP error : %s <BR>" s
            | _ → Printf.fprintf oc "Unknown error <BR>" );
```

```
          flush oc;
          Unix.shutdown fd Unix.SHUTDOWN_ALL
    end
  end;;
```

As we do not expect the servlet to communicate using Objective Caml's special internal values, we choose the type *string* as the protocol type. The functions `of_string` and `to_string` do nothing.

```
# module Simple_http_server =
    P_Text_Server (struct type t = string
                          let of_string x = x
                          let to_string x = x
                   end);;
```

Finally, we write the primary function to launch the service and construct an instance of the class *http_servlet*.

```
# let cgi_like_server port_num fun_serv =
   let sv =  new Simple_http_server.http_servlet port_num 3  fun_serv
   in sv#start;;
Characters 54-89:
   let sv =   new Simple_http_server.http_servlet port_num 3  fun_serv
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Unbound class Simple_http_server.http_servlet
```

Testing the Servlet

It is always useful during development to be able to test the parts that are already built. For this purpose, we build a small HTTP server which sends the file specified in the HTTP request as is. The function `simple_serv` sends the file whose name follows the GET request (the second element of the argument array). The function also displays all of the arguments passed in the request.

```
# let send_file oc f =
      let ic = open_in_bin f in
      try
        while true do
          output_byte oc (input_byte ic)
        done
      with End_of_file → close_in ic;;
val send_file : out_channel -> string -> unit = <fun>

# let simple_serv oc args =
    try
      Array.iter (fun x → print_string (x^" ")) args;
      print_newline();
      send_file oc args.(1)
    with _ → Printf.printf "error\n";;
val simple_serv : out_channel -> string array -> unit = <fun>

# let run n = cgi_like_server n simple_serv;;
Characters 13-28:
```

```
  let run n = cgi_like_server n simple_serv;;
              ^^^^^^^^^^^^^^^
Unbound value cgi_like_server
```

The command *run* 4003 launches this servlet on port 4003. In addition, we launch a browser to issue a request to load the page baro.html on port 4003. The figure 21.2 shows the display of the contents of this page in the browser.

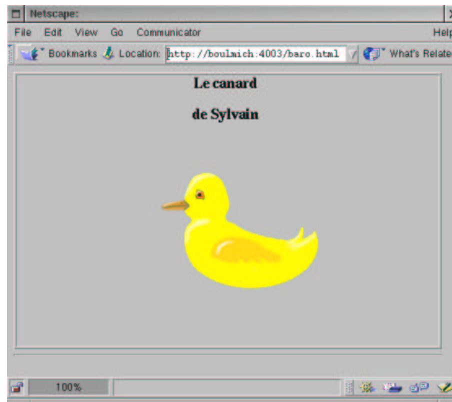


図 21.2: HTTP request to an Objective Caml servlet

The browser has sent the request GET /baro.html to load the page, and then the request GET /canard.gif to load the image.

HTML Servlet Interface

We will use a CGI-style server to build an HTML-based interface to the database of chapter 6 (see page 148).

The menu of the function main will now be displayed in a form on an HTML page, providing the same selections. The responses to requests are also HTML pages, generated dynamically by the servlet. The dynamic page construction makes use of the utilities defined below.

Application Protocol

Our application will use several elements from several protocols:

1. Requests are transmitted from a Web browser to our application server in the HTTP request format.
2. The data items within a request are encoded in the format used by CGI applications.
3. The response to the request is presented as an HTML page.

4. Finally, the nature of the request is specified in a format specific to the application.

We wish to respond to three kinds of request: queries for the list of mail addresses, queries for the list of email addresses, and queries for the state of received fees between two given dates. We give these query types respectively the names:
`mail_addr`, `email_addr` and `fees_state`. In the last case, we will also transmit two character strings containing the desired dates. These two dates correspond to the values of the fields `start` and `end` on an HTML form.

When a client first connects, the following page is sent. The names of the requests are encoded within it in the form of HTML anchors.

```
<HTML>
<TITLE> association </TITLE>
<BODY>
<HR>
<H1 ALIGN=CENTER>Association</H1>
<P>
<HR>
<UL>
<LI>List of
<A HREF="http://freres-gras.ufr-info-p6.jussieu.fr:12345/mail_addr">
mail addresses
</A>
<LI>List of
<A HREF="http://freres-gras.ufr-info-p6.jussieu.fr:12345/email_addr">
email addresses
</A>
<LI>State of received fees<BR>
<FORM
 method="GET"
 action="http://freres-gras.ufr-info-p6.jussieu.fr:12345/fees_state">
Start date : <INPUT type="text" name="start" value="">
End date : <INPUT type="text" name="end" value="">
<INPUT name="action" type="submit" value="Send">
</FORM>
</UL>
<HR>
</BODY>
</HTML>
```

We assume that this page is contained in the file `assoc.html`.

HTML Primitives

The HTML utility functions are grouped together into a single class called `print`. It has a field specifying the output channel. Thus, it can be used just as well in a CGI

application (where the output channel is the standard output) as in an application using the HTTP server defined in the previous section (where the output channel is a network socket).

The proposed methods essentially allow us to encapsulate text within HTML tags. This text is either passed directly as an argument to the method in the form of a character string, or produced by a function. For example, the principal method `page` takes as its first argument a string corresponding to the header of the page[1], and as its second argument a function that prints out the contents of the page. The method `page` produces the tags corresponding to the HTML protocol.

The names of the methods match the names of the corresponding HTML tags, with additional options added in some cases.

```
# class print (oc0:out_channel) =
    object(self)
      val oc = oc0
      method flush () = flush oc
      method str =
        Printf.fprintf oc "%s"
      method page header (body:unit → unit) =
        Printf.fprintf oc "<HTML><HEAD><TITLE>%s</TITLE></HEAD>\n<BODY>" header;
        body();
        Printf.fprintf oc "</BODY>\n</HTML>\n"
      method p () =
        Printf.fprintf oc "\n<P>\n"
      method br () =
        Printf.fprintf oc "<BR>\n"
      method hr () =
        Printf.fprintf oc "<HR>\n"
      method hr () =
        Printf.fprintf oc "\n<HR>\n"
      method h i s =
        Printf.fprintf oc "<H%d>%s</H%d>" i s i
      method h_center i s =
        Printf.fprintf oc "<H%d ALIGN=\"CENTER\">%s</H%d>" i s i
      method form url (form_content:unit → unit) =
        Printf.fprintf oc "<FORM method=\"post\" action=\"%s\">\n" url;
        form_content ();
        Printf.fprintf oc "</FORM>"
      method input_text =
        Printf.fprintf oc
          "<INPUT type=\"text\" name=\"%s\" size=\"%d\" value=\"%s\">\n"
      method input_hidden_text =
        Printf.fprintf oc "<INPUT type=\"hidden\" name=\"%s\" value=\"%s\">\n"
      method input_submit =
        Printf.fprintf oc "<INPUT name=\"%s\" type=\"submit\" value=\"%s\">"
      method input_radio =
        Printf.fprintf oc "<INPUT type=\"radio\" name=\"%s\" value=\"%s\">\n"
      method input_radio_checked =
```

1. This header is generally displayed in the title bar of the browser window.

```
      Printf.fprintf oc
        "<INPUT type=\"radio\" name=\"%s\" value=\"%s\" CHECKED>\n"
    method option =
      Printf.fprintf oc "<OPTION> %s\n"
    method option_selected opt =
      Printf.fprintf oc "<OPTION SELECTED> %s" opt
    method select name options selected =
      Printf.fprintf oc "<SELECT name=\"%s\">\n" name;
      List.iter
        (fun s → if s=selected then self#option_selected s else self#option s)
        options;
      Printf.fprintf oc "</SELECT>\n"
    method options selected =
      List.iter
        (fun s → if s=selected then self#option_selected s else self#option s)
  end ;;
```

We will assume that these utilities are provided by the module `Html_frame`.

Dynamic Pages for Managing the Association Database

For each of the three kinds of request, the application must construct a page in response. For this purpose we use the utility module `Html_frame` given above. This means that the pages are not really constructed, but that their various components are emitted sequentially on the output channel.
We provide an additional (virtual) page to be returned in response to a request that is invalid or not understood.

Error page The function `print_error` takes as arguments a function for emitting an HTML page (*i.e.*, an instance of the class `print`) and a character string containing the error message.

```
# let print_error (print:Html_frame.print) s =
    let print_body() =
     print#str s; print#br()
    in
     print#page "Error" print_body ;;
val print_error : Html_frame.print -> string -> unit = <fun>
```

All of our functions for emitting responses to requests will take as their first argument a function for emitting an HTML page.

List of mail addresses To obtain the page giving the response to a query for the list of mail addresses, we will format the list of character strings obtained by the function `mail_addresses`, which was defined as part of the database (see page 157). We will

assume that this function, and all others directly involving requests to the database, have been defined in a module named Assoc.

To emit this list, we use a function for outputting simple lines:

```
# let print_lines (print:Html_frame.print) ls =
    let print_line l =  print#str l; print#br() in
     List.iter print_line ls ;;
val print_lines : Html_frame.print -> string list -> unit = <fun>
```

The function for responding to a query for the list of mail addresses is:

```
# let print_mail_addresses print db =
     print#page "Mail addresses"
                (fun () → print_lines print (Assoc.mail_addresses db))
    ;;
val print_mail_addresses : Html_frame.print -> Assoc.data_base -> unit =
  <fun>
```

In addition to the parameter for emitting a page, the function print_mail_addresses takes the database as its second parameter.

List of email addresses This function is built on the same principles as that giving the list of mail addresses, except that it calls the function email_addresses from the module Assoc:

```
# let print_email_addresses print db =
     print#page "Email addresses"
                (fun () → print_lines print (Assoc.email_addresses db)) ;;
val print_email_addresses : Html_frame.print -> Assoc.data_base -> unit =
  <fun>
```

State of received fees The same principle also governs the definition of this function: retrieving the data corresponding to the request (which here is a pair), then emitting the corresponding character strings.

```
# let print_fees_state print db d1 d2 =
    let ls, t = Assoc.fees_state db d1 d2 in
    let page_body() =
     print_lines print ls;
     print#str ("Total : "^(string_of_float t));
     print#br()
    in
     print#page "State of received fees" page_body ;;
val print_fees_state :
  Html_frame.print -> Assoc.data_base -> string -> string -> unit = <fun>
```

Analysis of Requests and Response

We define two functions for producing responses based on an HTTP request. The first (`print_get_answer`) responds to a request presumed to be formulated using the GET method of the HTTP protocol. The second alters the production of the answer according to the actual method that the request used.

These two functions take as their second argument an array of character strings containing the elements of the HTTP request as analyzed by the function `get_query_string` (see page 669). The first element of the array contains the method, the second the name of the database request.
In the case of a query for the state of received fees, the start and end dates for the request are contained in the two fields of the form associated with the query. The data from the form are contained in the third field of the array, which must be decomposed by the function `get_form_content` (see page 669).

```
# let print_get_answer print q db =
    match q.(1) with
    | "/mail_addr" → print_mail_addresses print db
    | "/email_addr" → print_email_addresses print db
    | "/fees_state"
      → let nvs = get_form_content q.(2) in
        let d1 = List.assoc "start" nvs
        and d2 = List.assoc "end" nvs in
          print_fees_state print db d1 d2
    | _ → print_error print ("Unknown request: "^q.(1)) ;;
val print_get_answer :
  Html_frame.print -> string array -> Assoc.data_base -> unit = <fun>

# let print_answer print q db =
    try
      match q.(0) with
       "GET" → print_get_answer print q db
      | _ → print_error print ("Unsupported method: "^q.(0))
    with
      e
      → let s = Array.fold_right (^) q "" in
        print_error print ("Something wrong with request: "^s) ;;
val print_answer :
  Html_frame.print -> string array -> Assoc.data_base -> unit = <fun>
```

Main Entry Point and Application

The application is a standalone executable that takes the port number as a parameter. It reads in the database before launching the server. The main function is obtained from the function `print_answer` defined above and from the generic HTTP server function `cgi_like_server` defined in the previous section (see page 671). The latter function is located in the module `Servlet`.

```
# let get_port_num() =
    if (Array.length Sys.argv) < 2 then 12345
    else
      try int_of_string Sys.argv.(1)
      with _ → 12345 ;;
val get_port_num : unit -> int = <fun>

# let main() =
    let db = Assoc.read_base "assoc.dat" in
    let assoc_answer oc q = print_answer (new Html_frame.print oc) q db in
      Servlet.cgi_like_server (get_port_num()) assoc_answer ;;
Characters 138-161:
      Servlet.cgi_like_server (get_port_num()) assoc_answer ;;
      ^^^^^^^^^^^^^^^^^^^^^^^
Unbound value Servlet.cgi_like_server
```

To obtain a complete application, we combine the definitions of the display functions into a file `httpassoc.ml`. The file ends with a call to the function `main`:

```
main() ;;
```

We can then produce an executable named `assocd` using the compilation command:

```
ocamlc -thread -custom -o assocd unix.cma threads.cma \
       gsd.cmo servlet.cmo html_frame.cmo string_plus.cmo assoc.cmo \
       httpassoc.ml -cclib -lunix -cclib -lthreads
```

All that's left is to launch the server, load the HTML page[2] contained in the file `assoc.html` given at the beginning of this section (page 673), and click.

The figure 21.3 shows an example of the application in use. The browser establishes an initial connection with the servlet, which sends it the menu page. Once the entry fields are filled in, the user sends a new request which contains the data entered. The server decodes the request and calls on the association database to retrieve the desired information. The result is translated into HTML and sent to the client, which then displays this new page.

To Learn More

This application has numerous possible enhancements. First of all, the HTTP protocol used here is overly simple compared to the new versions, which add a header supplying the type and length of the page being sent. Likewise, the method `POST`, which allows modification of the server, is not supported.[3]

2. . . . taking care to update the URL according to your machine
3. Nothing prevents one from using `GET` for this, but that does not correspond to the standard.

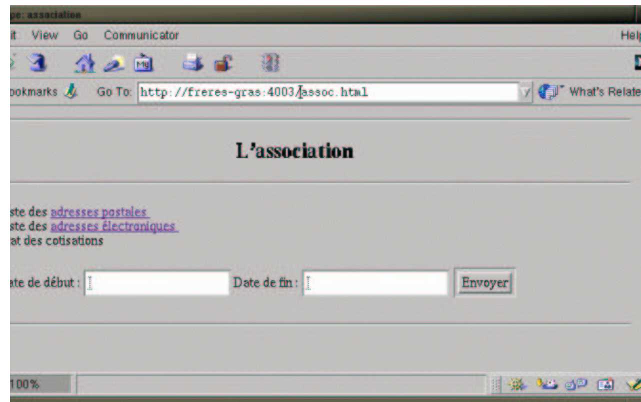


図 21.3: HTTP request to an Objective Caml servlet

To be able to describe the type of a page to be returned, the servlet would have to support the MIME convention, which is used for describing documents such as those attached to email messages.

The transmission of images, such as in figure 21.2, makes it possible to construct interfaces for 2-player games (see chapter 17), where one associates links with drawings of positions to be played. Since the server knows which moves are legal, only the valid positions are associated with links.

The MIME extension also allows defining new types of data. One can thus support a private protocol for Objective Caml values by defining a new MIME type. These values will be understandable only by an Objective Caml program using the same private protocol. In this way, a request by a client for a remote Objective Caml value can be issued via HTTP. One can even pass a serialized closure as an argument within an HTTP request. This, once reconstructed on the server side, can be executed to provide the desired result.

22

*Objective Caml*でのアプリケーション開発

ここまで読み進めて来た読者には Objective Caml が豊かな機能を持っていることに同意していただけるに違いありません。この言語は核となる、関数型と命令型の実行モデルに基づき、モジュールとオブジェクトという二種類の代表的なアプリケーション構成モデルを統合しています。ライブラリとして提供されていますが、スレッドはこの言語の魅力的な機能です。システムとのインターフェースも大部分はポータブルになっていて、分散プログラミングを柔軟に記述できる能力を Objective Caml は持っています。これらの様々なプログラミングパラダイムは推論を行う静的な型システムの中に統合されています。では、これらの機能はアプリケーション開発における Objective Caml の重要性を実証していると言えるでしょうか？あるいはもっと簡単に言うと「Objective Caml は良い言語でしょうか？」

しかし残念なことに、以下に挙げる古典的な評価法を Objective Caml に対して利用することはできません。

- (マーケティングの観点) 「多くのクライアントが買っています (だからいい言語です)。」
- (歴史的観点) 「今までに何万行ものプログラムが書かれています。」
- (システム開発者の観点) 「Unix や Windows がこの言語で書かれています。」
- (キラーアプリケーション)「例の有名なソフトウェアはこの言語で書かれています。」
- (政治的観点) 「ISO 標準で仕様が決まっています。」

この章では Objective Caml の特徴を最後にまた詳しく見ていきますが、今回はシステムの開発チームからの必要性という観点で考えていきます。評価項目となる基準は言語固有の品質、開発環境、コミュニティの発展度、今まで作成された主要なアプリケーションです。最後に違いをはっきりさせるために Objective Caml を同種の関数型言語やオブジェクト指向言語の Java と比較します。

評価項目

Objective Caml 配付パッケージには、バイトコードを生成するコンパイラと現代的なプロセッサの機械語コードを生成するコンパイラの二種類が含まれています。トップレベルは内部でバイトコードコンパイラを利用しています。配付パッケージではさらに、モジュールとして整理された数多くのライブラリ、モジュールの間の依存関係を計算するツールやプロファイリング、デバグのためのツールを提供しています。また C 言語とのインターフェースのおかげで Objective Caml のプログラムと C 言語のプログラムをリンクすることが可能です。C 言語との同じようなインターフェース (Java の JNI など) を提供している言語ともこの機能を使えばアクセスすることができます。

参照マニュアルは言語構文、開発ツール、ライブラリのシグネチャ等について解説しています。開発環境とは、これらの項目 (言語、ツール、ライブラリ、文書) を一体化させたものです。

言語

仕様と実装

新しい言語を学ぶには二種類の方法があるでしょう。一つは言語仕様を読んで大まかな認識を形成する方法です。もう一つは言語のユーザーズマニュアルを丹念に読み、一つ一つ例を学んで行く方法です。残念なことに Objective Caml にはそのどちらもないため自己学習の敷居がやや高いものと言わざるを得ません。(SML にあるような) 形式的な仕様も (ADA にあるような) 記述的な仕様のどちらもないためことがプログラムの動作の理解の障害となっているかもしれません。また仕様がないため、当然のことながら ISO, ANSI, IEEE から標準言語仕様を入手することもできません。そのため他の環境に対応した新しい実装を第三者が作ることは困難です。好運なことに INRIA が提供している実装の品質は極めて良く、またソースコードもダウンロードできるようになっています。

文法

Objective Caml の文法には奇妙に見える点がいくつもあり、それが言語習得の障害になっていることは否定できませんが、これは Objective Caml が関数型言語から発展して来たことに起因しています。この文法の特殊性については歴史的な逸話がたくさんあります。

関数適用の文法は単純な並置 (たとえば `f 1 2`) となっています。括弧がないことで初学者だけではなく C 言語プログラマや熟練した Lisp プログラマも混乱するかもしれません。しかしこれは括弧を使うのを嫌うプログラマが書いたコードを読む場合に困難となるのであって初学者が明示的に括弧を書くのを妨げている訳ではありません。

関数型以外の部分でも Objective Caml では所々でプログラミングの慣習からやや離れた文法を採用しています。例えば、配列の要素のアクセスはブラケットの適用ではなく `t.(i)` という記法になっています。メソッド適用はドットではなく井桁記号 (#) を採用しています。このような特異性は Objective Caml を把握する障害となっています。

最後に Objective Caml とその祖先の Caml の構文は最初の実装以来、無数の変更を受けて来ています。アプリケーションの開発は一般に長く続き、またソースコードは開発終了後も長く残りますから、頻繁な構文の変更は決して望まれるものではありません。

ただ最後に良い面も書いておくと、ML 系列の言語から譲り受けているパターンマッチの構文は関数定義の構文と単純な方法で気持良く統合されています。

静的型付け

Objective Caml 言語の根本的な性格は式と宣言の静的型付けによって決まっています。静的型付けによってプログラム実行中に型エラーに見舞われることはありません。静的型推論は ML 系列の関数型言語を思い起こさせますが、Objective Caml では命令型、オブジェクト指向拡張に対する型規則の大部分をも管理することができます。しかしオブジェクト指向拡張の場合には、プログラマが明示的に型制約を書いて型推論を補助しなければならない時があります。もちろんその場合でも式や定義の静的型付けは維持されていて、動的な型付けを行うオブジェクト指向言語のように「メソッドが見付かりません」という例外を発生しないことが保証されています。Objective Caml が卓越した安全性の高さを誇っているのはこのためです。

Objective Caml のパラメータ型多相性は総称的なアルゴリズムの実装を可能にしています。総称性はオブジェクト指向層にも適用されており、型パラメータを持つクラスを使って総称的なコードを記述できます。パラメータ型多相性とは、他の言語のテンプレートのようにコードを展開したりしないので、ソースコード上、他のコードと容易に混在できます。パラメータ型多相性はコード再利用性を高める重要な機能となっています。

オブジェクト指向拡張では、オブジェクト間の部分型によって指定される包含的多相性の観念を型の概念に追加しています。この観念のおかげで、クラス間の継承関係によって得られるコード再利用性と静的型付けの安全性を両立できています。

ライブラリとツール

配付パッケージに含まれているライブラリはプログラミングの大部分をカバーしています。スタック、待ち行列、ハッシュ表、AVL 木などの標準的なデータ構造やそれを操作する関数群がライブラリには含まれています。このライブラリは言語のバージョンアップにつれて次第に充実して来ています。

`Unix` ライブラリは I/O やプロセス管理などの低水準プログラミングを記述するのに使います。このライブラリはプラットフォームによって機能に差がありますが、そのために多少プログラミングに支障が出るかもしれません。

正確な数値計算ライブラリや正規表現ライブラリはある種のアプリケーションの記述になくてはならないものでしょう。

残念なことですが、汎用的なグラフィクスライブラリは機能が少ないため GUI の作成には限界があります。

C 言語のためのライブラリを Objective Caml から簡単に使うことができます。

提供されているツールの中でも字句解析と構文解析のためのツールは、複雑なテキストデータを扱うには不可欠であり、特に注目に値します。それらのツールは古典的な `lex` と `yacc` を元に直和型を使って Objective Caml の機能に完全に統合し、より使いやすいものになっています。

最後に言語がどんなにすばらしいものであっても**実際の開発現場では**デバッギングの過程を無視する訳にはいきません。Objective Caml 2.04 配付パッケージには統合開発環境 (IDE) は含まれていません。確かにトップレベルを使えばコンパイルと関数単位の動作テストを素早く行えます (このこと自体は否定できない利点です) が、そのやり方はプラットフォームとプログラマによって大きく変わります。X-Windows ではカットアンドペースト主体になりますし、`emacs` では内部でシェルを呼び出すことになりますし、Windows ではバッファを評価させることになります。Objective Caml の次のバージョン (付録 B 参照) では初めてモジュールとインターフェースのブラウザとトップレベルと統合された構造エディタを提供します。付加されているデバッガにはやや使いづらい点 (関数型固有の問題とパラメータ型多相のため) があり、また Unix システムに限定されています。

文書

配付パッケージにはドキュメントとして印刷可能な (PostScript) マニュアルとオンラインで見るための (HTML) マニュアルが含まれています。このマニュアルは言語について知りたい初学者のためのものではなく、逆にライブラリの内容を調べるため、またはツールへのパラメータの与え方などを探すのに不可欠なものです。Inria の Caml に関するページには教育目的のものが少しありますが、多くは Caml-Light 言語に関するものです。Objective Caml の完全なチュートリアルは存在しませんが、この本がその代用となってくれることを望みます。

他の開発ツール

ここまでは Objective Caml 配付パッケージに限定して話を進めてきました。しかし、この言語を使っている開発者のコミュニティの活動はメーリングリスト (`caml-list@inria.fr`) の流量が証明しているようにとても活発で、開発を支援する数多くのツール、ライブラリ、拡張が提供されてきました。以下では編集、文法拡張、他言語インターフェース、並列プログラミングのためのツールの使い方について説明します。加えて、多くのグラフィクスライブラリについても触れます。このような貢献の多くは “Caml Hump” サイトで見付けることができます。

リンク: http://caml.inria.fr/humps/index.html

編集ツール

Objective Caml の文法を認識できる `emacs` エディタのためのモードは複数提供されています。それらのモードでは自動的にインデントを行い、読みやすく整形します。`emacs` は Windows 環境でも動作し、ウィンドウの中で Objective Caml のトップレベルを立ち上げることができるので Windows 環境では統合環境の一つの代替として使えます。

文法拡張

配付パッケージに含まれている字句解析、構文解析ツールはそれだけで完結していますが、文法拡張の機能はありません。`camlp4` ツール (315 ページのリンク参照) を Objective Caml の構文解析器の替わりに使うことができます。このツールは、ユーザが Objective Caml の文法を拡張または変更できる機能を提供しています。また生成された構文木を綺麗に表示する (pretty printing) 機能もあります。このツールを利用すると Objective Caml の文法を拡張した言語のトップレベルを簡単に記述できます。それどころか別の言語を Objective Caml の上に実装することすら可能です。

他言語インターフェース

第 12 章では Objective Caml 言語から C 言語を呼び出す方法について解説しました。多言語アプリケーションは同じメモリ領域に共有しながらそれぞれの言語の長所を活かせます。しかし C の関数を Objective Caml から呼び出せるようにするには非常に面倒な準備が必要になります。それを緩和するために `IDL` で記述されたインターフェースの仕様からインタフェースの実装を自動的に生成する機能を持つ `camlIDL` ツール (350 ページのリンク参照) があります。このツールは COM (Windows) のコンポーネントを Objective Caml から利用できる機能もあります。

グラフィックインターフェース

`Graphics` ライブラリは描画とユーザとの簡単な対話が可能ですが、とてもその名にふさわしい機能を提供しているとは言えません。第 13 章でこのライブラリを使って簡単な対話を行うコンポーネントを作る方法を示しました。`Graphics` ライブラリだけを基礎として使っている限り、異なるプラットフォーム (X-Windows, Windows, MacOS) 上での互換性が保証されますが、同時に低水準なイベントと描画に機能が限定されます。

この問題を克服されるためにいくつかのプロジェクトがライブラリを提供していますが、残念ながら機能の完全性、移植性、文書の充実、簡単な使用法のすべてを実現したものはありません。主要なプロジェクト (“Caml Hump” から抜きだしたものです) をここに挙げてみます。

- OlibRt: よい語感を持っていますが、その名に恥じないものです。X-Windows で動作しますが、文書がありません。配付パッケージには完全なソースコードと数多くのゲームが含まれています。
  リンク: http://cristal.inria.fr/~ddr/
- camlTk: このライブラリは Tk ツールキットの完全で良く文書化されたインターフェースを提供しています。ただし Tcl/Tk の特定のバージョンに依存しており、インストールに難があるかもしれません。`mmm` [Rou96] という Objective Caml で書かれたウェブブラウザを実装するのに使われました。
  リンク: http://caml.inria.fr/~rouaix/camltk-readme.html
- Objective Caml のために書き換えられた `Xlib` ライブラリがあります。それを使った `Efuns` という `emacs` 風エディタがあります。`Xlib` は最も低水準の動作を記述するものであり、X-Windows 以外の環境では動作しません。

- mlGtk は Gtk の上に作られたインターフェースです。現在開発中であり、文書はありません。Gtk 自体に互換性があるため Unix と Windows のどちらでも動作します。また Tk よりも簡単に利用できます。ただし Objective Caml のオブジェクト指向層を利用しており、その点で何か問題があるかもしれません。
- LabTk は Unix 上で動作する、Tcl/Tk のインターフェースで、Objective Caml の次期バージョンで統合される拡張機能 (付録 B 章参照) を利用しています。Tcl/Tk 自体を含んでいるのでインストールは簡単です。

コミュニティの努力はあるものの、互換性のあるグラフィックインターフェースを構築するツールはまだ提供されていません。将来的に LabTk が他のプラットフォーム上でも動作すれば良いと思います。

並列分散プログラミング

並行分散のための基本的機構としてスレッドとソケットは既に提供されています。C 言語のライブラリを呼び出すことで、古典的な並列プログラミングを行うことができます。様々な種類の並列モデルを実装したライブラリ、言語拡張を利用できます。ただし CORBA の遠隔オブジェクトに対するメソッド呼び出しはまだサポートされていません。

ライブラリ

二つの主要な並列プログラミングライブラリ `MPI` (Message Passing Interface) と `PVM` (Parallel Virtual Machine) のインターフェースが提供されています。文書、参考情報、ライブラリのソースなどは以下のサイトで入手できます。

リンク: http://www.netlib.org

"Caml Hump" には Objective Caml とのインターフェースが定義されているバージョンをダウンロードできる様々な HTTP アドレスが載っています。

言語拡張

Caml-Light や Objective Caml のための数多くの並列拡張が開発されています。

- Caml-Flight ([FC95]) とは SPMD (Simple Program Multiple Data) 型の並列モデルを Caml-Light 言語で実行できるようにした言語拡張です。同じコードが固定された数のプロセス上で実行されます。通信は明示的で、同期命令 `sync` を発行した時に一対の通信 `get` が行われます。
  Link:
  http://www.univ-orleans.fr/SCIENCES/LIFO/Members/ghains/caml-flight.html
- BSML [BLH00] は BSP モデルのための言語拡張です。言語は合成性 (compositionality) を持ち、固定されたプロセッサ数に対して正確な性能の予測ができます。
  Link:
  http://www.univ-orleans.fr/SCIENCES/LIFO/Members/loulergu/bsml.html
- OCAMLP3 [DDLP98] は P3L 言語のスケルトンモデルに基づいた並列プログラミング環境です。様々な並列実行のスケルトンがあらかじめ定義されており、それら

を組み合わせてプログラムを記述します。プログラムは逐次モードでも並列モードでも実行できます。そのため Objective Caml 自身のツールも再利用できます。

リンク: http://www.di.unipi.it/~susanna/projects.html

- JoCAML [CL99] は join calculus モデルに基づいています。Join calculus は分散オブジェクトや移動コードを記述できる高レベルな並行性、通信、同期操作を支援しています。

  リンク: http://pauillac.inria.fr/jocaml/

- Lucid Synchrone ([CP95]) は reactive system を実装するための言語です。この言語は Objective Caml とデータフロー同期型言語を組み合わせたものです。

  リンク: http://www-spi.lip6.fr/~pouzet/lucid-synchrone/

Objective Caml で開発されたアプリケーション

いくつかのアプリケーションが既に Objective Caml を使って開発されています。ただしここで対象にしているのはあくまでも「公開されている」アプリケーションだけです。すなわち誰でも無料または有料で入手できるものでなければなりません。

他の関数型言語と同様に Objective Caml はコンパイラ実装言語として優れています。`ocaml` コンパイラのブートストラップ[1]はその一つの例です。同様に数多くの言語拡張が記述されています。既に示したように並列プログラミングへの拡張や O'Labl (この言語は Objective Caml と統合中の言語で詳しくは付録第 B 章参照) などの言語が Objective Caml で記述されています。これらのアプリケーションへのリンクは “Caml Hump” に記載されています。

次に Objective Caml 言語が向いている対象領域は定理証明支援です。その代表的なアプリケーションは Caml とほぼ同時に発展してきた Coq というシステムです。歴史的に ML は独立したシステムとして認知されるまでの間 LCF (Logic for Computable Functions) システムのメタ言語として知られていました。したがって定理証明支援システムの実装言語に向いているのは至極自然なことです。

他に向いているアプリケーション領域として並列性と通信 (686 ページ参照) です。その良い例として `Ensemble` システムがあります。

リンク: http://www.cs.cornell.edu/Info/Projects/Ensemble/

Objective Caml で開発された重要なアプリケーションは Inria の Caml サイトに記載されています。ただしこのリストはすべてのアプリケーションを網羅していません。

リンク: http://caml.inria.fr/users_programs-eng.html

1. ブートストラップとはコンパイラのコンパイルを自分自身のコンパイラで行うことを意味します。浮動点に達する、すなわちコンパイラと生成された実行ファイルが同一になるかどうか調べることはコンパイラの正当性を確かめる良いテストになります。

特に紹介しておきたいのが LaTeX から HTML への変換器である hevea です。附属の CD-ROM に含まれているこの本の HTML バージョンを生成するのに hevea を使っています。

リンク: http://pauillac.inria.fr/~maranget/hevea/

いま挙げたアプリケーションは確かに重要なものですが、この章の最初で述べた「キラーアプリケーション」と呼べるものではありません。それどころか、これらのアプリケーションは Objective Caml の重要性を示す新しい領野を開拓した訳でもありません。しかし、この話題を果してコンピュータ科学界から提起するべきなのか疑問に感じます。新しいアプリケーションはむしろ産業界から言語標準化 (形式化) の議論と共に、または異なったプログラミングとプログラム記述法を統合しなければならないアプリケーションの必要性と共に提起されるものだと思います。

類似の関数型言語

関数型としての面、あるいは型付けの面で Objective Caml と類似の言語があります。Objective Caml は ML から派生した言語なので最も近い系列の言語は大西洋の向こうで開発されている SML (Standard ML) でしょう。Lisp 系列の言語、とりわけ Scheme 言語の ML との最大の違いは動的型付けを行う点です。遅延評価を行う Miranda と Haskell の二つの言語は ML の型システムを取り込み拡張しています。エリクソン社とクリオネットワークス社によってそれぞれ開発された Erlang と SCOL は通信機能に特化された関数型言語です。

ML 系列

ML 系列の言語には Caml (Categorical Abstract Machine Language) 派と SML (Standard ML) 派の主要な二つの勢力があります。Caml 派の言語にはその派生の Caml-Light と Objective Caml があります。SML 派の言語には直系の SML/NJ と mosml があります。Caml は 1986 年から 1990 年の間に INRIA の FORMEL プロジェクトとしてパリ第 7 大学とエコール・ノルマル・シュペリオールÉcole Normale Supérieure との共同研究の中で開発されました。実装は Le_Lisp のランタイムを基にしていました。Caml は言語の中に文法定義と整形表示器 pretty printer を含んでいて新しく定義した言語と Caml の間で値を交換することがってきました。更新可能な値への型システムは制限されていて多相型を持つことができませんでした。最初の派生言語である Caml-Light は CAM 機械ではなく Zinc を使って実装されていましたが、系統を表すため CAM の名前は残りました。この言語はより効率的な実装がなされ、関数クロージャの割り当てや GC が最適化されていました。Caml 系列の言語はその時代の PC 上で動作させることができました。Caml-Light は副作用に関する型付けが改良され、数多くのライブラリが作られていきました。次の派生言語の Caml Special Light (CSL) ではパラメータ付きモジュールと機械語コードコンパイラが実装されました。最後の派生言語である Objective Caml は主に CSL にオブジェクト指向拡張を追加したものです。Caml 系列の言語では元々一度も完全な仕様が作成されなかったため、このような仕様拡張は自由に行われて来ました。

一方 SML 派の開発方針はその逆でした。最初の実装が行われる前に形式的仕様 [MTH97] が作成されました。この形式的仕様は分かりづらい点があったためその後出た本で解説

[MT91] を行いました。仕様を決めてから実装を行うという開発手法は複数の実装を並行して行うことができます。最も良く知られている SML の実装は Lucent (元 AT&T) が開発した SML/NJ (Standard ML of New Jersey) です。SML には始めから統合されたパラメータ付きモジュールがありました。初期の型システムは副作用の扱いが Caml とは異なり、レベル付きの弱い型変数を使っていました。この二つの言語系列の違いは文献 [CKL96] に詳しく記載されています。この違いは次第に小さくなっていきました。この二つの系列の言語は関数型と命令型の核の部分は本質的に同じ型システムを持っています。現在の Objective Caml にはパラメータ付きモジュールがあり、SML もレコード型の変更など多くの変化を受けて結果として Objective Caml に近付いています。この二つの言語が統合されていないのは、単に別々に開発されてきたからにすぎません。SML には商用の開発環境である MLWorks が Harlequin から提供されています。

リンク: http://www.harlequin.com/products/

SML の実装である `mosml` は Caml-Light のランタイムを基に実装されています。

Scheme

Scheme 言語 (1975) は Lisp 言語 (1960) の方言の一つで既に標準化 (IEEE Std 1178-1990) されています。Scheme 言語は命令型の特徴があり正格 strict な評価を行い、動的に型付けされた言語です。文法は正則で、括弧を多用する特徴があります。基本的なデータ構造は異種リストを構築できる、点 dot により分割された組 (ML の組と同等) です。Scheme のトップレベルは、Scheme 言語を使って `(print (eval (read)))` と簡単に記述できます。`read` 関数は標準入力から読み Scheme の式を構築します。`eval` 関数は構築された式を評価し `print` 関数は結果を表示します。Scheme は便利なマクロ展開システムを持っていて、`eval` 関数と組み合わせて言語拡張を簡単に行えます。計算の割り込み (例外) 機能を持っているだけでなく、継続 continuation を使って計算の再開も行えます。継続とは計算の一時点を表し、特殊形式 `call_cc` によって現地点から実行を再開させる継続を生成します。この継続を呼び出すと `call_cc` を評価する時点に計算状態が移動します。Scheme の実装は多く、中にはマクロ言語として使われている画像変換ツール GIMP の例もあります。Scheme は継続などの特殊な機能のおかげで新しい逐次、並列言語の実装を実験する優れた環境でもあります。

遅延評価の機能を持つ言語

ML や Lisp とは対照的に遅延評価を行う言語では関数呼び出し時に引数を評価せず、実際に値が必要になった時点で評価します。Lazy ML という遅延評価を行う ML もありますが、この分野で主流なのは Miranda と Haskell の一族です。

Miranda

Miranda([Tur85]) は純関数型言語つまり副作用のない言語です。関数とデータ構造を定義する等式の列が Miranda のプログラムです。

Link:
http://www.engin.umd.umich.edu/CIS/course.des/cis400/miranda/miranda.html

例えばフィボナッチ関数はこのように定義されます。

```
 fib a = 1, a=0
       = 1, a=1
       = fib(a-1) + fib(a-2), a>1
```

同じ関数を定義する等式の中でガード (条件部) または以下の例にあるようなパターンに一致したものが選ばれます。

```
fib 0 = 1
fib 1 = 1
fib a = fib(a-1)+ fib(a-2)
```

なお、この二つの定義法は混在可能です。

関数は高階であり、また部分的に評価することもあります。遅延評価を行い部分式は値が必要になるまで評価されません。したがって Miranda のリストはそのままでストリームとなります。

Miranda は無限のデータ構造 (リスト、集合) を表す簡単な記法があり、例えばすべての正の整数は `[1..]` と表されます。フィボナッチ数列は簡単に

```
fibs = [a | (a,b) <- (1,1),(b,a+b)..]
```

と記述できます。値が必要になるまで評価されないので、この宣言をした時点では計算を行いません。

Miranda は Hindley-Milner 型の型システムを持つ、強く型付けされた言語です。型規則は本質的に ML のものと同等です。ユーザによるデータ型の定義が可能です。

Miranda は純関数型遅延評価言語の代表例です。

Haskell

The main Haskell language website contains reports of the definition of the language and its libraries, as well as its main implementations.

リンク: http://www.haskell.org

Haskell を使った関数型プログラミングについての本が数冊出版されており、最新のものは文献 [Tho99] です。

この言語は関数型言語の新しい概念のほぼすべてを含んでいます。この言語は純関数型 (副作用のない) で、遅延評価 (正格でない) を行い、ML のようなパラメータ型多相性に加えて非標準的多相性 (多重定義のため) を備えています。

非標準的多相性　これは今まで見てきた多相性とは違います。ML の多相関数は多相的引数の型が何であるかを考慮せずにすべての型の要素に対して一様に扱います。Haskell

では違います。多相関数は多相的引数の型によって異なった動作を行うことができます。この性質を利用すれば関数の多重定義を行えます。

基本的なアイデアは、多重定義された関数をまとめる機能を持つ型クラスに含まれています。クラス宣言は新しいクラスとそのクラスに対して許される操作を定義します。クラスのインスタンス宣言は、ある具体的な型が宣言されたクラスのインスタンスであることを指示します。インスタンス宣言では多重定義された操作の定義を提供します。

例えば the `Num` クラスを次のように宣言します。

```
class Num a where
  (+)    :: a -> a -> a
  negate :: a -> a
```

次に `Num` クラスのインスタンスである `Int` を次のように宣言します。

```
instance Num Int where
  x + y    =  addInt x y
  negate x = negateInt x
```

同様に `Num` クラスのインスタンスである `Float` を次のように宣言します。

```
instance Num Float where
  x + y    =  addFloat x y
  negate x = negateFloat x
```

`negate` `Num` は適用された引数が `Int` か `Float` のどちらであるかによって異なった動作をします。

クラスに対して継承を行うこともできます。下位クラスでは上位クラスの関数を再定義することができて、下位クラスのインスタンスの動作を修正できます。

他の特徴　Haskell には他にも次のような特徴があります。

- モナドを使った純関数的 I/O システム
- 必要に応じて (lazy に) 構築される配列
- 同じデータ型に対して別々の表現を与えるビュー機能

この言語には関数型言語の研究分野で生み出された先進的な機能のほぼすべてが含まれています。これはこの言語の長所であると同時に短所にもなり得ます。

通信記述言語

ERLANG

ERLANG は並行プログラミングのための動的な型付けを行う関数型言語です。この言語は Ericsson 社が電話通信システムの記述のために開発されました。現在はオープンソースであり、以下のサイトから入手できます。

リンク: http://www.erlang.org

この言語はプロセスの生成と通信を簡単に記述できるように設計されています。通信はメッセージの受渡しによって行い、メッセージの配送は非同期型つまり遅延時間があります。ポートに対してプロトコルを定義するのは簡単です。プロセスはそれぞれ自分自身の定義辞書を持っています。エラー管理は例外機構を使い、例外はプロセス間で配送できます。数多くの電話システムが Erlang によって行われてきており、明らかに開発時間の短縮に貢献しています。

SCOL

SCOL は 3D 世界を構築するためにクリオネットワークス社によって開発された通信記述言語です。

リンク: http://www.cryo-networks.com

この言語の核となる部分は Caml と似ています。関数型で静的な型付けを行い、パラメータ型多相性を持ち、型推論を行います。サウンド、2D、3D ライブラリ API のおかげでマルチメディアを扱う機能を持っています。3D エンジンはかなり効率的です。SCOL の独特な点はチャネルを使った、仮想機械同士の通信機能にあります。チャネルとは環境とネットワークリンクの組であり、ネットワークリンクとは TCP または UDP のソケットです。

SCOL はネットワークからダウンロードしたコードの信頼性の問題を独特の方法で解決しています。ネットワーク上でやりとりするのはソースコードに限定し、それを受け取った仮想機械は型推論をした後に実行します。正しく型付けできないコードは正式なコンパイラによって生成されたコードと見なされません。このような判定法をネットワーク上の通信時間を犠牲にせずに実装するために、簡潔にアルゴリズムを表現できる静的型付き関数型言語が選ばれました。

オブジェクト指向言語 — *Java* との比較

Objective Caml は関数型の世界から生まれてきたものですが、オブジェクト指向拡張の部分をオブジェクト指向言語の重要な代表例と比較してみる必要があります。ここでは Java を取り上げます。Java は実装の観点からはそれほど Objective Caml と違いはないもののオブジェクトモデルと型システムという点については大きく異なっています。

Java はサンマイクロシステムズ社によって開発されたオブジェクト指向言語です。この言語の公式サイトは以下の URL でアクセスできます。

リンク: http://java.sun.com

主な特徴

Java 言語はクラスという概念のある言語です。継承は多重継承ではなく単一継承であり、メソッドの再定義が行えます。またメソッドの多重定義が行えます。型付けは静的です。継承されたクラスは基になるクラスの部分型となります。

Java のクラスはパラメータ型多相を持っていません。Java で許されている多相性は多重定義による非標準的多相性と再定義による包含的多相性の 2 種類です。

マルチスレッドの機能があり、ソケットや遠隔オブジェクトのメソッド起動 (Remote Method Invocation) を使って分散システムの開発が行えます。

Java の実装方針は Objective Caml に似ています。Java プログラムは仮想機械語にコンパイルされ、動的にロードされます。生成されたコードは仮想機械によって解釈され、アーキテクチャ非依存です。基本的なデータ型はすべてのアーキテクチャ上で同じ表現となることが保証されています。ランタイムは GC を行います。

Java には有益なクラスライブラリがあります。JDK にはおよそ 600 のクラスがあり、独立の開発者によってその機能を補完するクラスライブラリが数多く提供されています。主要なライブラリは画像インターフェースライブラリやホスト間通信機能を統合した I/O ライブラリなどです。

Objective Caml との違い

Java と Objective Caml の主要な違いは型システムに起因するメソッドの再定義や多重定義の仕方にあります。継承されたメソッドの再定義では引数の型が完全に一致しなければなりません。メソッドの多重定義はメソッド呼び出しの引数の型によってメソッドの定義を切替えています。以下の例ではクラス `B` はクラス `A` を継承しています。クラス `B` はクラス `A` の `to_string` メソッドを再定義していますが、このメソッドはクラス `B` で多重定義されています。その上 `eq` メソッドを多重定義しています。なぜなら `eq` メソッドの引数の型がクラス `A` とクラス `B` で違っているからです。結局のところクラス `B` には `eq` メソッドが二つと `to_string` メソッドが二つあります。

```
class A {
  boolean eq (A o) {  return true;}
  String to_string (int n ) { }
}

class B extends A {
  boolean eq (B o) {  return true;}
  String to_string (int n ) { }
  String to_string (float x, float y)
}
```

束縛は実行時に決定されますが、多重定義の解消 (呼び出されるメソッドの決定) はコンパイル時に決定されます。

二番目の重要な違いは、C 言語にあるような型キャストの有無に起因しています。以下の例ではクラス A とクラス B のオブジェクトそれぞれ a と b が定義されています。そのほかに 3 つの変数 c, d, e が宣言されています。

```
{
  A a = new A ();
  B b = new B ();
  A c = (A) b; // ok
  B d = (B) c; // ok
  B e = (B) a; // ng
}
```

b の型は a の型の部分型なので変数 b から変数 c へのキャストは合法です。この場合は実行時型検査は省略されます。一方それ以外のキャスト式では変数 c と変数 a の値がクラス B のオブジェクトであるかどうか実行時に検査が必要になります。このプログラムでは変数 c についてはその通りですが変数 a はクラス B のオブジェクトを指していません。したがって最後のキャスト式は実行時に例外を発生します。Java は静的に型付けされた部分と動的に型付けされた部分を持つ言語と言えるでしょう。動的に型制約を検査する機能はとりわけ (継承を多用する) 画像インターフェースを記述する時には有益です。また Java にはパラメータ型多相がないため総称的なクラスを書くときにこの機能を頻繁に使わなければなりません。

*Objective Caml*開発の将来

新しい言語が重要なアプリケーション (C 言語にとっての Unix のような) なしに、あるいは商業的、工業的な支援 (Java 言語にとっての SUN のような) を伴わずに生き延びることは極めて困難です。言語自体の質の良さだけでは充分ではありません。この章でずっと解説してきたように Objective Caml には数多くの長所といくつかの短所があります。Objective Caml はこれまで INRIA によって維持されてきました。Objective Caml は CRISTAL 計画の中で考案され、実装されました。INRIA では学術研究の中から生み出された Objective Caml を新しいプログラミングパラダイムを試す実験環境と実装言語として使っています。またこの言語は様々な大学や進学コースの授業で幅広く教えられています。毎年数千人の学生がこの言語の概念を学び、実習しています。このようにして Objective Caml は学術世界の中で重要な地位を作ってきました。コンピュータ科学の教育においてフランスだけではなく米国でも理論と実践の両面で数多くのプログラマを生み出しています。

その一方で工業界での活動はそれほど活発ではありません。私達の知る限り Objective Caml で開発され、一般に売り出され、Objective Caml を宣伝に使っている商用ソフトウェアは一つもありません。最も条件に近い唯一の例はクリオネットワークス社から提供されている SCOL 言語でしょう。しかしこれはやや誇張があると言わざるを得ません。Objective Caml を使ったソフトウェア開発に対する資金提供の呼びかけは今後、出てくることになるでしょう。私達はそれが雪だるま式に拡大するとは思っていませんが、この

ようなプログラミング言語に対する需要があるということが重要です。Objective Camlを使ったソフトウェア開発がすぐに見返りをもたらす訳ではありませんが、今後のソフトウェア開発において Objective Caml を採用する可能性を考慮することは重要です。

ここで言語と開発環境との間の関連性を示すべきでしょう。言語の発展にしたがってある種の保証をすることは疑いもなく重要です。しかし、この種の保証を行うには Objective Caml が学術界の外へと乗り出す決断をしなければなりません。しかし Objective Camlが「世に出る」ためには以下の保証がなければなりません。

- 言語仕様の上位互換性を保持することで、アプリケーション開発の継続性を保証(オブジェクトなどの新しい言語要素の安定性が難点)
- 将来的な標準化につながる、外部の開発者と共同での言語仕様の策定 (同時並行の実装作業が可能になり、常に言語の実装が提供されている状況を保証できる)
- 汎用性のある画像インターフェース、CORBA bus のようなデータベースインターフェース、より使いやすいデバグ環境を含んだ開発環境の考案

ここで提起した論点のいくつか、とりわけ標準化については学術界の中でも扱うことができます。しかし残りのものは産業界でのみ意味を持っています。したがって今後のすべては学術界と産業界の協力の度合にかかってきます。「フリーで」提供された言語が商用目的で保全されるようになった先例として ADA 言語があります。ADA 言語の `gnat` コンパイラはまずフリーで提供され、後に ACT 社によって保全されるようになりました。

リンク: http://www.act-europe.fr

おわりに

コンピュータ科学は既に社会の中の一つの産業になってしまっていますが、そもそも (プログラミング) 言語の習得とはすぐれて人間の主観に関わる行為です。「心情は、理性の知らない、それ自身の根拠を持っている」[2] のだとしたら Objective Caml には心情に対して訴えかけるだけのものがあります。

Objective Caml は幅広いプログラミングパラダイムに対応する一方で堅実な理論的基礎を持っています。加えて、対話的に結果を確かめることのできるインタプリタはプログラミングの授業にとても適しています。

- 構造化された型と抽象型のおかげでメモリ表現と割り当てに煩わされることなく複雑なデータ構造を対象にしたアルゴリズムを記述することができます。
- 言語が基にしている関数型の理論モデルは、評価と型付けの概念を学ぶ良い素材になっています。それは「真のプログラマ」が独習する役に立つでしょう。
- 様々なプログラミングモデルを、独立に学ぶことができます。モジュール式やオブジェクト指向から低水準のシステムプログラミングまで Objective Caml が役に立たない分野はほとんどありません。
- 記号計算に極めて適しており、コンパイルや人工知能のような理論的教科を巧みに支援します。

このような資質を持っているので Objective Caml はコンピュータ科学のカリキュラムへの導入部から、発展的なプログラミングの授業まで幅広く利用されています。そのため言語の高水準の機能と実行との関係が良く理解されています。多くの教師はこのような教育上の利点に惹かれており、そのため多くの学生や研究者が Objective Caml を利用してプログラミングを習った経験を持っています。

Objective Caml を使ったプログラム開発は快適です。コンパイラは高速にプログラムを読み込み、静的な型推論は何も見逃しません。型エラーとならない場合でも別の静的解析がバグの貴重な兆候を教えてくれるでしょう。不完全なパターンマッチは警告されま

2. 訳註. パスカル『パンセ』277

す。逐次実行部に置かれている関数の部分適用は検出されます。このような種々の機能がプログラマに快適な開発を保証すると同時に、コンパイラが効率の良いコードを高速に生成することを可能にしています。

Objective Caml のコンパイル性能、関数型プログラミングの簡潔さ、ライブラリの質と多様さは「使い捨て、1 回限り」のソフトウェアを記述するのに向いています。しかし「使い捨て、1 回限り」ではないアプリケーションを記述するのに向いていない訳ではありません。全く同じ理由から Objective Caml は実験的なプロトタイプを記述するのに貴重なツールです。それどころかモジュールとオブジェクトを使った構造化プログラミングによって最終的なアプリケーションを作成できるだけの概念と機能を持っています。

最後に Objective Caml の開発者のコミュニティはプログラミング研究分野での技術革新にすばやく対応する環境を形成しています。ソースコードを無料で配付しているので新しいアイデアを実験できる環境としても利用できます。

他の言語に慣れているプログラマにとって Objective Caml を学ぶにはある種の努力が必要かもしれません。しかしこれは、学問の対象と同様に絶えまない発展の中にあります。我々はこの本が、種々の概念の複雑さを省略することなく、Objective Caml を学ぶ一助になって欲しいと願います。またアプリケーション開発者に対しては、この本のおかげで開発の成果が多少なりとも向上したとすればそれは望外の喜びです。

Part V

付録

A

Cyclic Types

Objective Caml's type system would be much simpler if the language were purely functional. Alas, language extensions entail extensions to the type language, and to the inference mechanism, of which we saw the illustration with the weak type variables (see page 73), made unavoidable by imperative extensions.

Object typing introduces the notion of **cyclic type**, associated with the keyword **as** (see page 458), which can be used independently of any concept of object oriented programming. The present appendix describes this extension of the type language, available through an option of the compiler.

Cyclic types

In Objective Caml, it is possible to declare recursive data structures: such a structure may contain a value with precisely the same structure.

```
# type sum_ex1 = Ctor of sum_ex1 ;;
type sum_ex1 = Ctor of sum_ex1

# type record_ex1 = { field : record_ex1 } ;;
type record_ex1 = { field : record_ex1; }
```

How to build values with such types is not obvious, since we need a value before building one! The recursive declaration of values allows to get out of this vicious circle.

```
# let rec sum_val = Ctor sum_val ;;
val sum_val : sum_ex1 = Ctor (Ctor (Ctor (Ctor (Ctor ...))))

# let rec val_record_1 = { field = val_record_2 }
```

```
  and     val_record_2 = { field = val_record_1 } ;;
val val_record_1 : record_ex1 =
  {field = {field = {field = {field = {field = ...}}}}}
val val_record_2 : record_ex1 =
  {field = {field = {field = {field = {field = ...}}}}}
```

Arbitrary planar trees can be represented by such a data structure.

```
# type 'a tree = Vertex of 'a * 'a tree list ;;
type 'a tree = Vertex of 'a * 'a tree list
# let height_1 = Vertex (0,[]) ;;
val height_1 : int tree = Vertex (0, [])
# let height_2 = Vertex (0,[ Vertex (1,[]); Vertex (2,[]); Vertex (3,[]) ]) ;;
val height_2 : int tree =
  Vertex (0, [Vertex (1, []); Vertex (2, []); Vertex (3, [])])
# let height_3 = Vertex (0,[ height_2; height_1 ]) ;;
val height_3 : int tree =
  Vertex (0,
   [Vertex (0, [Vertex (...); Vertex (...); Vertex (...)]); Vertex (0, [])])

(* same with a record *)
# type 'a tree_rec = { label:'a ; sons:'a tree_rec list } ;;
type 'a tree_rec = { label : 'a; sons : 'a tree_rec list; }
# let hgt_rec_1 = { label=0; sons=[] } ;;
val hgt_rec_1 : int tree_rec = {label = 0; sons = []}
# let hgt_rec_2 = { label=0; sons=[hgt_rec_1] } ;;
val hgt_rec_2 : int tree_rec = {label = 0; sons = [{label = 0; sons = []}]}
```

We might think that an enumerated type with only one constructor is not useful, but by default, Objective Caml does not accept recursive type abbreviations.

```
# type 'a tree = 'a * 'a tree list ;;
Characters 6-34:
  type 'a tree = 'a * 'a tree list ;;
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
The type abbreviation tree is cyclic
```

We can define values with such a structure, but they do not have the same type.

```
# let tree_1 = (0,[]) ;;
val tree_1 : int * 'a list = (0, [])
# let tree_2 = (0,[ (1,[]); (2,[]); (3,[]) ]) ;;
val tree_2 : int * (int * 'a list) list = (0, [(1, []); (2, []); (3, [])])
# let tree_3 = (0,[ tree_2; tree_1 ]) ;;
val tree_3 : int * (int * (int * 'a list) list) list =
  (0, [(0, [(...); (...); (...)]); (0, [])])
```

In the same way, Objective Caml is not able to infer a type for a function whose argument is a value of this form.

```
# let max_list = List.fold_left max 0 ;;
val max_list : int list -> int = <fun>

# let rec height = function
      Vertex (_,[]) → 1
    | Vertex (_,sons) → 1 + (max_list (List.map height sons)) ;;
val height : 'a tree -> int = <fun>
```

```
# let rec height2 = function
      (_,[]) → 1
    | (_,sons) → 1 + (max_list (List.map height2 sons)) ;;
Characters 95-99:
    | (_,sons) -> 1 + (max_list (List.map height2 sons)) ;;
                                                  ^^^^
This expression has type 'a list but is here used with type
  ('b * 'a list) list
```

The error message tells us that the function `height2` could be typed, if we had type equality between *'a* and *'b * 'a list*, and precisely this equality was denied to us in the declaration of the type abbreviation *tree*.

However, object typing allows to build values, whose type is **cyclic**. Let us consider the following function, and try to guess its type.

```
# let f x = x#copy = x ;;
```

The type of `x` is a class with method `copy`. The type of this method should be the same as that of `x`, since equality is tested between them. So, if *foo* is the type of `x`, it has the form: *< copy : foo ; .. >*. From what has been said above, the type of this function is cyclic, and it should be rejected; but it is not:

```
# let f x = x#copy = x ;;
val f : (< copy : 'a; .. > as 'a) -> bool = <fun>
```

Objective Caml does accept this function, and notes the type cyclicity using **as**, which identifies *'a* with a type containing *'a*.

In fact, the problems are the same, but by default, Objective Caml will not accept such types unless objects are concerned. The function `height` is typable if it gives a cyclicity on the type of an object.

```
# let rec height a = match a#sons with
      [] → 1
    |  l → 1 + (max_list (List.map height l)) ;;
val height : (< sons : 'a list; .. > as 'a) -> int = <fun>
```

Option -rectypes

With a compiler option, we can avoid this restriction to objects in cyclic types.

```
$ ocamlc -rectypes  ...
$ ocamlopt -rectypes  ...
$ ocaml -rectypes
```

If we take up the above examples in a toplevel started with this option, here is what we get.

```
# type 'a tree = 'a * 'a tree list ;;
type 'a tree = 'a * 'a tree list

# let rec height = function
      (_,[]) → 1
    | (_,sons) → 1 + (max_list (List.map height sons)) ;;
val height : ('b * 'a list as 'a) -> int = <fun>
```

The values `tree_1`, `tree_2` and `tree_3` previously defined don't have the same type, but they all have a type compatible with that of `height`.

```
# height tree_1 ;;
- : int = 1
# height tree_2 ;;
- : int = 2
# height tree_3 ;;
- : int = 3
```

The keyword **as** belongs to the type language, and as such, it can be used in a type declaration.

構文 : **type** *nom* **=** *typedef* **as '***var* **;;**

We can use this syntax to define type *tree*.

```
# type 'a tree = ( 'a * 'vertex list ) as 'vertex ;;
type 'a tree = 'a * 'b list as 'b
```

警告 If this mode may be useful in some cases, it tends to accept the typing of too many values, giving them types that are not easy to read.

Without the option `-rectypes`, the function below would have been rejected by the typing system.

```
# let inclus l1 l2 =
    let rec mem x = function
        [] → false
      | a::l → (l=x) || (mem x a) (* an error on purpose: a and l inverted *)
    in List.for_all (fun x → mem x l2) l1  ;;
val inclus : ('a list as 'a) list list -> 'a -> bool = <fun>
```

Although a quick examination of the type allows to conclude to an error, we no longer have an error message to help us locating this error.

B

Objective Caml 3.04

Independently of the development of Objective Caml, several extensions of the language appeared. One of these, named Olabl, was integrated with Objective Caml, starting with version 3.00.

This appendix describes briefly the new features offered in the current version of Objective Caml at the time of this writing, that is. Objective Caml 3.04. This version can be found on the CD-ROM accompanying this book. The new features include:

- **labels**;
- **optional arguments**;
- **polymorphic constructors**;
- the `ocamlbrowser` IDE;
- the `LablTk` library.

The reader is referred to the Objective Caml reference manual for a more detailed description of these features.

Language Extensions

Objective Caml 3.04 brings three language extensions to Objective Caml: labels, optional arguments, and polymorphic constructors. These extensions preserve backward compatibility with the original language: a program written for version 2.04 keeps the same semantics in version 3.04.

Labels

A label is an annotation for the arguments of a function in its declaration and its application. It is presented as a separate identifier of the function parameter (formal or actual), enclosed between an initial symbol '~' and a final symbol ':'.

Labels can appear in the declarations of functions:

構文 : `let f ~label:p = exp`

in the anonymous declarations with the keyword **fun** :

構文 : `fun ~label:p -> exp`

and in the actual parameter of a function:

構文 : `( f ~label:exp )`

Labels in types The labels given to arguments of a functional expression appear in its type and annotate the types of the arguments to which they refer. (The '~' symbol in front of the label is omitted in types.)

```
# let add ~op1:x ~op2:y = x + y ;;
val add : op1:int -> op2:int -> int = <fun>

# let mk_triplet ~arg1:x ~arg2:y ~arg3:z = (x,y,z) ;;
val mk_triplet : arg1:'a -> arg2:'b -> arg3:'c -> 'a * 'b * 'c = <fun>
```

If one wishes to give the same identifier to the label and the variable, as in `~x:x`, it is unnecessary to repeat the identifier; the shorter syntax `~x` can be used instead.

構文 : `fun ~p -> exp`

```
# let mk_triplet ~arg1 ~arg2 ~arg3 = (arg1,arg2,arg3) ;;
val mk_triplet : arg1:'a -> arg2:'b -> arg3:'c -> 'a * 'b * 'c = <fun>
```

It is not possible to define labels in a declaration of a function by pattern matching; consequently the keyword **function** cannot be used for a function with a label.

```
# let f = function ~arg:x → x ;;
Characters 18-23:
  let f = function ~arg:x -> x ;;
                   ^^^^^
Syntax error
# let f = fun ~arg:x → x ;;
val f : arg:'a -> 'a = <fun>
```

Labels in function applications When a function is defined with labeled parameters, applications of this function require that matching labels are provided on the function arguments.

```
# mk_triplet ~arg1:'1' ~arg2:2 ~arg3:3.0 ;;
- : char * int * float = ('1', 2, 3)
# mk_triplet '1' 2 3.0 ;;
- : char * int * float = ('1', 2, 3)
```

A consequence of this requirement is that the order of arguments having a label does not matter, since one can identify them by their label. Thus, labeled arguments to a function can be "commuted", that is, passed in an order different from the function definition.

```
# mk_triplet ~arg2:2  ~arg1:'1'  ~arg3:3.0 ;;
- : char * int * float = ('1', 2, 3)
```

This feature is particularly useful for making a partial application on an argument that is not the first in the declaration.

```
# let triplet_0_0 = mk_triplet ~arg2:0 ~arg3:0 ;;
val triplet_0_0 : arg1:'a -> 'a * int * int = <fun>
# triplet_0_0 ~arg1:2 ;;
- : int * int * int = (2, 0, 0)
```

Arguments that have no label, or that have the same label as another argument, do not commute. In such a case, the application uses the first argument that has the given label.

```
# let test ~arg1:_ ~arg2:_  _  ~arg2:_  _ = () ;;
val test : arg1:'a -> arg2:'b -> 'c -> arg2:'d -> 'e -> unit = <fun>

# test ~arg2:() ;;  (* the first arg2: in the declaration *)
- : arg1:'a -> 'b -> arg2:'c -> 'd -> unit = <fun>

# test () ;;       (* the first unlabeled argument in the declaration *)
- : arg1:'a -> arg2:'b -> arg2:'c -> 'd -> unit = <fun>
```

Legibility of code Besides allowing re-ordering of function arguments, labels are also very useful to make the function interface more explicit. Consider for instance the `String.sub` standard library function.

```
# String.sub ;;
- : string -> int -> int -> string = <fun>
```

In the type of this function, nothing indicates that the first integer argument is a character position, while the second is the length of the string to be extracted. Objective Caml 3.04 provides a "labelized" version of this function, where the purpose of the different function arguments have been made explicit using labels.

```
# StringLabels.sub ;;
- : string -> pos:int -> len:int -> string = <fun>
```

Clearly, the function `StringLabels.sub` takes as arguments a string, the position of the first character, and the length of the string to be extracted.

Objective Caml 3.04 provides "labelized" versions of many standard library functions in the modules `ArrayLabels`, `ListLabels`, `StringLabels`, `UnixLabels`, and `MoreLabels`. Table B.1 gives the labeling conventions that were used.

label	significance
pos:	a position in a string or array
len:	a length
buf:	a string used as buffer
src:	the source of an operation
dst:	the destination of an operation
init:	the initial value for an iterator
cmp:	a comparison function
mode:	an operation mode or a flag list

図 B.1: Conventions for labels

Optional arguments

Objective Caml 3.04 allows the definition of functions with labeled **optional** arguments. Such arguments are defined with a default value (the value given to the parameter if the application does not give any other explicitly).

構文 : **fun ?***name*: **(** *p* **=** exp_1**)** -> exp_2

As in the case of regular labels, the argument label can be omitted if it is identical to the argument identifier:

構文 : **fun ?(** *name* **=** exp_1**)** -> exp_2

Optional arguments appear in the function type prefixed with the **?** symbol.

```
# let sp_incr ?inc:(x=1) y =  y := !y + x ;;
val sp_incr : ?inc:int -> int ref -> unit = <fun>
```

The function `sp_incr` behaves like the function `incr` from the Pervasives module.

```
# let v = ref 4  in  sp_incr v ; v ;;
- : int ref = {contents = 5}
```

However, one can specify a different increment from the default.

```
# let v = ref 4  in  sp_incr ~inc:3 v ; v ;;
- : int ref = {contents = 7}
```

A function is applied by giving the default value to all the optional parameters until the actual parameter is passed by the application. If the argument of the call is given without a label, it is considered as being the first non-optional argument of the function.

```
# let f ?(x1=0) ?(x2=0) x3 x4 = 1000*x1+100*x2+10*x3+x4 ;;
val f : ?x1:int -> ?x2:int -> int -> int -> int = <fun>
# f 3 ;;
- : int -> int = <fun>
# f 3 4 ;;
- : int = 34
# f ~x1:1 3 4 ;;
- : int = 1034
# f ~x2:2 3 4 ;;
- : int = 234
```

An optional argument can be given without a default value, in this case it is considered in the body of the function as being of the type *'a option*; `None` is its default value.

構文 : **fun ?***name*:*p* -> *exp*

```
# let print_integer ?file:opt_f n =
    match opt_f with
        None → print_int n
      | Some f → let fic = open_out f in
                  output_string fic (string_of_int n) ;
                  output_string fic "\n" ;
                  close_out fic ;;
val print_integer : ?file:string -> int -> unit = <fun>
```

By default, the function `print_integer` displays its argument on standard output. If it receives a file name with the label `file`, it outputs its integer argument to that file instead.

注意

If the last parameter of a function is optional, it will have to be applied explicitly.

```
# let test ?x ?y  n  ?a ?b  =  n ;;
val test : ?x:'a -> ?y:'b -> 'c -> ?a:'d -> ?b:'e -> 'c = <fun>
# test 1 ;;
- : ?a:'_a -> ?b:'_b -> int = <fun>
# test 1 ~b:'x' ;;
- : ?a:'_a -> int = <fun>
# test 1 ~a:() ~b:'x' ;;
- : int = 1
```

Labels and objects

Labels can be used for the parameters of a method or an object's constructor.

```
# class point ?(x=0) ?(y=0) (col : Graphics.color) =
    object
      val pos = (x,y)
      val color = col
      method print ?dest:(file=stdout) () =
          output_string file "point (" ;
          output_string file (string_of_int (fst pos)) ;
          output_string file "," ;
          output_string file (string_of_int (snd pos)) ;
          output_string file ")\n"
    end ;;
class point :
  ?x:int ->
  ?y:int ->
  Graphics.color ->
  object
    val color : Graphics.color
    val pos : int * int
    method print : ?dest:out_channel -> unit -> unit
  end

# let obj1 = new point ~x:1 ~y:2 Graphics.white
   in obj1#print () ;;
point (1,2)
- : unit = ()
# let obj2 = new point Graphics.black
   in obj2#print () ;;
point (0,0)
- : unit = ()
```

Labels and optional arguments provide an alternative to method and constructor overloading often found in object-oriented languages, but missing from Objective Caml.

This emulation of overloading has some limitations. In particular, it is necessary that at least one of the arguments is not optional. A dummy argument of type *unit* can always be used.

```
# class number ?integer ?real () =
    object
      val mutable value = 0.0
      method print = print_float value
      initializer
        match (integer,real) with
            (None,None) | (Some _,Some _) → failwith "incorrect number"
          | (None,Some f) → value <- f
```

```
          | (Some n,None) → value <- float_of_int n
    end ;;
class number :
  ?integer:int ->
  ?real:float ->
  unit -> object val mutable value : float method print : unit end
```

```
# let n1 = new number ~integer:1 () ;;
val n1 : number = <obj>
# let n2 = new number ~real:1.0 () ;;
val n2 : number = <obj>
```

Polymorphic variants

The variant types of Objective Caml have two principal limitations. First, it is not possible to extend a variant type with a new constructor. Also, a constructor can belong to only one type. Objective Caml 3.04 features an alternate kind of variant types, called **polymorphic variants** that do not have these two constraints.

Constructors for polymorphic variants are prefixed with a **`** (backquote) character, to distinguish them from regular constructors. Apart from this, the syntactic constraints on polymorphic constructors are the same as for other constructors. In particular, the identifier used to build the constructor must begin with a capital letter.

構文 : `` `Name ``

ou

構文 : `` `Name type ``

A group of polymorphic variant constructors forms a type, but this type does not need to be declared before using the constructors.

```
# let x = `Integer 3 ;;
val x : [> `Integer of int] = `Integer 3
```

The type of `x` with the symbol **`[>`** indicates that the type contains at least the constructor `` `Integer `` *`int`*.

```
# let int_of = function
      `Integer n → n
    | `Real r → int_of_float r ;;
val int_of : [< `Integer of int | `Real of float] -> int = <fun>
```

Conversely, the symbol **`[<`** indicates that the argument of `int_of` belongs to the type that contains at most the constructors `` `Integer `` *`int`* and `` `Real `` *`float`*.

It is also possible to define a polymorphic variant type by enumerating its constructors:

構文 : **type** *t* **= [‘***Name*$_1$ **| ‘***Name*$_2$ **|** ... **| ‘***Name*$_n$ **]**

or for parameterized types:

構文 : **type** *('a,'b,...) t* **= [‘***Name*$_1$ **| ‘***Name*$_2$ **|** ... **| ‘***Name*$_n$ **]**

```
# type value = [ ‘Integer of int | ‘Real of float ] ;;
type value = [ ‘Integer of int | ‘Real of float]
```

Constructors of polymorphic variants can take arguments of different types.

```
# let v1 = ‘Number 2
  and v2 = ‘Number 2.0 ;;
val v1 : [> ‘Number of int] = ‘Number 2
val v2 : [> ‘Number of float] = ‘Number 2
```

However, v1 and v2 have different types.

```
# v1=v2 ;;
Characters 4-6:
  v1=v2 ;;
     ^^
This expression has type [> ‘Number of float] but is here used with type
  [> ‘Number of int]
```

More generally, the constraints on the type of arguments for polymorphic variant constructors are accumulated in their type by the annotation **&**.

```
# let test_nul_integer = function ‘Number n → n=0
  and test_nul_real = function ‘Number r → r=0.0 ;;
val test_nul_integer : [ ‘Number of int] -> bool = <fun>
val test_nul_real : [ ‘Number of float] -> bool = <fun>
# let test_nul x = (test_nul_integer x) || (test_nul_real x) ;;
Characters 56-57:
  let test_nul x = (test_nul_integer x) || (test_nul_real x) ;;
                                                          ^
This expression has type [ ‘Number of int] but is here used with type
  [ ‘Number of float]
```

The type of `test_nul` indicates that the only values accepted by this function are those with the constructor `‘Number` and an argument which is at the same time of type *int* and of *float*. That is, the only acceptable values are of type *'a*!

```
# let f () = test_nul (failwith "returns a value of type 'a") ;;
Characters 12-20:
  let f () = test_nul (failwith "returns a value of type 'a") ;;
             ^^^^^^^^
Unbound value test_nul
```

The types of the polymorphic variant constructor are themselves likely to be polymorphic.

```
# let id = function ‘Ctor → ‘Ctor ;;
val id : [ ‘Ctor] -> [> ‘Ctor] = <fun>
```

The type of the value returned from `id` is "the group of constructors that contains at least `‘Ctor`" therefore it is a polymorphic type which can instantiate to a more precise type. In the same way, the argument of `id` is "the group of constructors that contains no more than `‘Ctor`" which is also likely to be specified. Consequently, they follow the general polymorphic type mechanism of Objective Caml knowing that they are likely to be weakened.

```
# let v = id ‘Ctor ;;
val v : _[> ‘Ctor] = ‘Ctor
```

`v`, the result of the application is not polymorphic (as denoted by the character _ in the name of the type variable).

```
# id v ;;
- : _[> ‘Ctor] = ‘Ctor
```

`v` is monomorphic and its type is a sub-type of "contains at least the constructor `‘Ctor`". Applying it with `id` will force its type to be a sub-type of"contains no more than the constructor `‘Ctor`". Logically, it must now have the type "contains exactly `‘Ctor`". Let us check.

```
# v ;;
- : [ ‘Ctor] = ‘Ctor
```

As with object types, the types of polymorphic variant constructors can be open.

```
# let is_integer = function
      ‘Integer (n : int) → true
    | _ → false ;;
val is_integer : [> ‘Integer of int] -> bool = <fun>
# is_integer (‘Integer 3) ;;
- : bool = true
# is_integer ‘Other ;;
- : bool = false
```

All the constructors are accepted, but the constructor `‘Integer` must have an integer argument.

```
# is_integer (‘Integer 3.0) ;;
Characters 13-25:
  is_integer (‘Integer 3.0) ;;
              ^^^^^^^^^^^^
This expression has type [> ‘Integer of float] but is here used with type
  [> ‘Integer of int]
```

As with object types, the type of a constructor can be cyclic.

```
# let rec long = function ‘Rec x → 1 + (long x) ;;
val long : ([ ‘Rec of ’a] as ’a) -> int = <fun>
```

Finally, let us note that the type can be at the same time a sub-group and one of a group of constructors. Starting with a a simple example:

```
# let ex1 = function ‘C1 → ‘C2 ;;
val ex1 : [ ‘C1] -> [> ‘C2] = <fun>
```

Now we identify the input and output types of the example by a second pattern.

```
# let ex2 = function ‘C1 → ‘C2 | x → x ;;
```

```
val ex2 : ([> `C1 | `C2] as 'a) -> 'a = <fun>
```

We thus obtain the open type which contains at least `` `C2 `` since the return type contains at least `` `C2 ``.

```
# ex2 ( `C1 : [> `C1 ] ) ;;  (* is a subtype of [<`C2|`C1| .. >`C2] *)
- : _[> `C1 | `C2] = `C2
# ex2 ( `C1 : [ `C1 ] ) ;;   (* is not a subtype of [<`C2|`C1| .. >`C2] *)
Characters 6-9:
  ex2 ( `C1 : [ `C1 ] ) ;;   (* is not a subtype of [<`C2|`C1| .. >`C2] *)
        ^^^
This expression has type [ `C1] but is here used with type [> `C1 | `C2]
The first variant type does not allow tag(s) `C2
```

`LablTk` *Library*

The interface to Tcl/Tk was integrated in the distribution of Objective Caml 3.04, and is available for Unix and Windows. The installation provides one new command: `labltk`, which launches a toplevel interactive loop integrating the `LablTk` library.

The `LablTk` library defines a large number of modules, and heavily uses the language extensions of Objective Caml 3.04. A detailed presentation of this module falls outside the scope of this appendix, and we invite the interested reader to refer to the documentation of Objective Caml 3.04.

There is also an interface with Gtk, written in class-based style, but it is not yet part of the Objective Caml distribution. It should be compatible with Unix and Windows.

OCamlBrowser

OcamlBrowser is a code browser for Objective Caml, providing a `LablTk`-based graphical user interface. It integrates a "navigator" allowing to browse various modules, to look at their contents (names of values and types), and to edit them.

When launching OCamlBrowser by the command `ocamlbrowser`, the list of all the compiled modules available (see figure B.2) is displayed. One can add more modules by specifying a path to find them. From the menu `File`, one can launch a toplevel interactive loop or an editor in a new window.

When one of the modules is clicked on, a new window opens to display its contents (see figure B.3). By selecting a value, its type appears in bottom of the window.

In the main window, one can search on the name of a function. The result appears in a new window. The figure B.4 shows the result of a search on the word `create`.

There are other possibilities that we let the user discover.

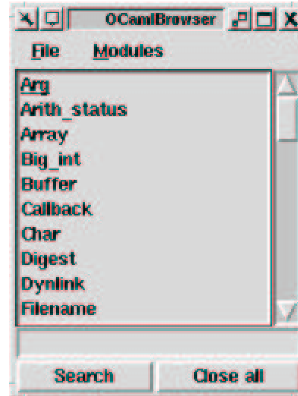


図 B.2: OCamlBrowser : the main window

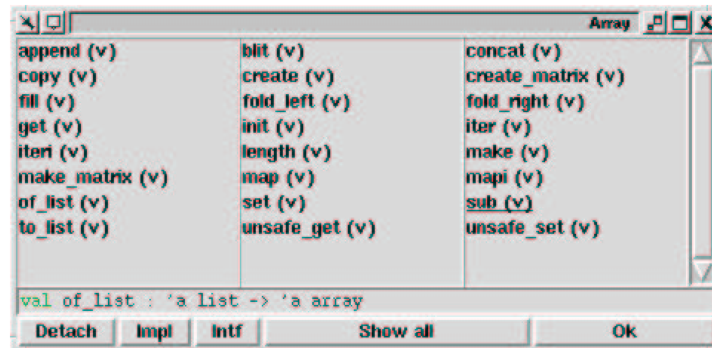


図 B.3: OCamlBrowser : module contents

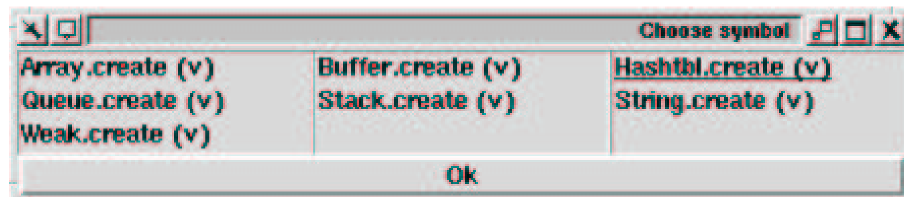


図 B.4: OCamlBrowser : search for `create`

参考文献

[AC96] María-Virginia Aponte and Giuseppe Castagna. Programmation modulaire avec surcharge et liaison tardive. In *Journées Francophones des Langages Applicatifs*. INRIA, January 1996.

[AHU83] Alfred Aho, John Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[And91] G. Andrews. *Concurrent Programming : Principles and practices*. Benjamin Cumming, 1991.

[Ari90] Ben Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall, second edition, 1990.

[ASS96] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, second edition, 1996.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1986.

[BLH00] Olivier Ballereau, Frédéric Loulergue, and Gaétan Hains. High level BSP programming: BSML and BSlambda. In Stephen Gilmore, editor, *Trends in Functional Programming, volume 2*. Intellect, 2000.

[CC92] Emmanuel Chailloux and Guy Cousineau. Programming images in ML. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, 1992.

[CCM87] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The Categorical Abstract Machine. *Science of Computer Programming*, 8:173–202, 1987.

[CDM98a] Rémy Card, Éric Dumas, and Franck Mével. *Linux 2.0* **カーネルブック**. オーム社, 1998.

[CDM98b] Rémy Card, Éric Dumas, and Franck Mével. *The Linux Kernel Book*. Wiley, John & Sons, 1998.

[CKL96] Emmanuel Chailloux, Laurent Kirsch, and Stéphane Lucas. Caml2sml, un outil d'aide à la traduction de Caml vers Sml. In *Journées Francophones des Langages Applicatifs*. INRIA, January 1996.

[CL99] Sylvain Conchon and Fabrice Le Fessant. JoCaml: mobile agents for Objective-Caml. In *International Symposium on Agent Systems and Applications*, 1999.

[CM98] Guy Cousineau and Michel Mauny. *The functional approach to programming*. Cambridge University Press, 1998.

[CP95] Paul Caspi and Marc Pouzet. A functional extension to LUSTRE. In *8th International Symposium on Languages for Intensional Programming*, Sydney, May 1995. World Scientific.

[CS94] Emmanuel Chailloux and Ascánder Suárez. mlPicTeX, a picture environment for LaTeX. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, 1994.

[DDLP98] Marco Danelutto, Roberto Di Cosmo, Xavier Leroy, and Susanna Pelagatti. Parallel functional programming with skeletons: the ocamlp3l experiment. In *ML Workshop*. ACM SIGPLAN, 1998.

[DEMN98] Roland Ducournau, Jérôme Euzenat, Gérald Masini, and Amedeo Napoli, editors. *Langages et modèles à objets: état et perspectives de la recherche*. INRIA, 1998.

[Eng98] Emmanuel Engel. *Extensions sûres et praticables du système de types de ML en présence d'un langage de modules et de traits impératifs*. PhD thesis, Université Paris-Sud, Orsay, France, mai 1998.

[FC95] Christian Foisy and Emmanuel Chailloux. Caml Flight: a Portable SPMD Extension of ML for Distributed Memory Multiprocessors. In *Conference on High Performance Functional Computing*, April 1995.

[FF98] Robert B. Findler and Matthew Flatt. Modular Object-Oriented Programming with Units and Mixins. In *International Conference on Functional Programming*. ACM, 1998.

[FW00] Jun Furuse and Pierre Weis. Entrées/Sorties de valeurs en Caml. In *JFLA'2000 : Journées Francophones des Langages Applicatifs*, Mont Saint-Michel, January 2000. INRIA.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[HF+96] Pieter Hartel, Marc Feeley, et al. Benchmarking implementations of functional languages with "Pseudoknot", a float-intensive benchmark. *Journal of Functional Programming*, 6(4), 1996.

[HS94] Samuel P. Harbison and Guy L. Steele. *C: A reference manual.* Prentice-Hall, fourth edition, 1994.

[Hui97] Christian Huitema. *IPv6 – The New Internet Protocol.* Prentice Hall, 1997.

[Jon98] Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, 1998.

[Ler90] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.

[Ler92] Xavier Leroy. Programmation du système Unix en Caml Light. Technical report 147, INRIA, 1992.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc.* O'Reilly, second edition, 1992.

[LRVD99] Xavier Leroy, Didier Rémy, Jérôme Vouillon, and Damien Doligez. The objective caml system release 2.04. Technical report, INRIA, December 1999.

[MdR92] Michel Mauny and Daniel de Rauglaudre. Parser in ML. Research report 1659, INRIA, avril 1992.

[MNC+91] Gérald Masini, Amedeo Napoli, Dominique Colnet, Daniel Léonard, and Karl Tombre. *Object-Oriented Languages.* Academic Press, New York, 1991.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, 1991.

[MTH97] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML (revised).* MIT Press, 1997.

[Rep99] John Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

[Rob89] Eric S. Robert. Implementing exceptions in C. Technical Report SRC-40, Digital Equipment, 1989.

[Rou96] François Rouaix. A Web navigator with applets in Caml. In *Proceedings of the 5th International World Wide Web Conference, in Computer Networks and Telecommunications Networking*, volume 28, pages 1365–1371. Elsevier, May 1996.

[RV98] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998. A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.

[Sed88] Robert Sedgewick. *Algorithms.* Addison-Wesley, second edition, 1988.

[Ste92] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

[Tho99] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison Wesley, seconde edition, 1999.

[Tur85] David A. Turner. Miranda: A non-strict functional language with polymorphic types. In J. Jouannaud, editor, *Proceedings International Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16, New York, NY, September 1985. Springer-Verlag.

[Wil92] Paul. R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, number 637 in LNCS, pages 1–42. Springer-Verlag, 1992.

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, 1995.

[ス 00] W. リチャード スティーブンス. **詳解** *UNIX* **プログラミング**. ピアソン・エデュケーション, 2000.

概念索引

A

abstract machine 199
abstract type 411
abstraction 21
application 21
- of a function 21
- partial 24

arity 22

B

big numbers 240
binding
- static 26

BNF .... 参照: 文法
boolean 15
bytecode 197
- dynamic loading 242
- the compiler .... 参照: compiler

C

C (the langage) 317
callback 343
cartesian product 16
character 14
character string 14
client-server 632
closure 23, 26
- from C 343
- representation 334

command line 235
- parsing of 236

compilation 197
- portability 208

compilation unit 201
compiler 197
- bytecode 199, 202
- command 201
- linking 202
- native 204

conditional 18
constraint 421
constructor 33, 44
- constant 45
- functional 45

cyclic
- type 701

D

declaration
- of an exception 54
- external 320
- of function 21
- global 19
- local 20
- by matching 40
- of an operator 25
- recursive 27
- simultaneous global 19
- simultaneous local 21
- of type 41

scope 48
of value 19
digest .... 参照: fingerprint
dot notation 214
dynamic loading 242

E
environment 23, 334
evaluation
deferred 105
event 132
exception 53
declaration 54
handling 56
printing 239
raising 55
with C 345
expression
functional 21
external declaration ... 参照: declaration

F
file
extension
.ml 202
.mli 202
interface 202
finalization 340
fingerprint 227
floating-point number 12
floats
arrays of 333
representation 333
function 21
for finalization 340
higher order 25
mutually recursive 27
partial 53
polymorphic 28
recursive 27
registration with C 344
tail recursive 94
function call .... 参照: application
functional 25
functor 423

G
GC 249, 253
from C 336
ストップ・アンド・コピー 258
漸進的 263
多世代 261
保守的 262
マーク・アンド・スイープ 256
マイナー 263
メジャー 263

H
Has-a 445
hashing 227

I
inlining 204
input-output 223
with C 325
installing Objective Caml 2
MacOS 4
Linux 4
online HTML help 5
Unix 5
Windows 2
integer 12
interface 411
graphical 353
with C 319, 325
with the system 234
interoperability 317
intput-output 229
Is-a 447

L
label 708
lazy (data) 105
library 213
preloaded 215
standard 215
linearization 228
linking 202
with C 322
list 17
association 221
matching 39

M
matching 33
exhaustive 35

destructive . . . 110
module . . . 414
constraint . . . 415
local definition . . . 427
opening . . . 214
parameterized . . . 423
sub-module . . . 422

O
OCamlBrowser . . . 716
operator
declaration . . . 25
optional (argument) . . . 710

P
pair . . . 16
pattern
character interval . . . 39
combining . . . 36
guard . . . 38
matching . . . 33
naming . . . 37
wildcard . . . 33, 35
pattern matching . . . 参照: matching
persistence . . . 228
persistent
value . . . 228
polymorphism . . . 28
portability . . . 208
protocol . . . 645
http . . . 646

R
reader-writer . . . 613
record . . . 42

S
scope of a variable . . . 26
self . . . 449
session . . . 206
sharing . . . 231
signature . . . 414
constraint . . . 415
socket . . . 629
standalone executable . . . 207
stream . . . 108
strict (language) . . . 95
string . . . 参照: character string
strings
representation . . . 332
structure . . . 414
super . . . 449

T
this . . . 449
toplevel . . . 205
construction . . . 206
options . . . 205
Toplevel loop
directives . . . 205
trace . . . 275
tree . . . 50
tuple . . . 16
type
abstract . . . 411
constraint . . . 30, 421
constructor . . . 33
declaration . . . 41
enumerated . . . 44
function . . . 49
functional . . . 21
mutually recursive . . . 41
parameterized . . . 30, 41, 47
product . . . 41
record . . . 42
recursive . . . 47
sum . . . 41, 44
sum types
representation . . . 331
union . . . 44

U
UML . . . 443

V
value
declaration . . . 19
exploration . . . 325
function . . . 334
global declaration . . . 19
immediate . . . 327
local declaration . . . 19
persistent
type . . . 234
representation . . . 325
in C . . . 320

structured 328
variable
bound 26
free 23, 26
type 28
variants
polymorphic 713

Z
Zinc 199
interpreter 207

あ
値
共有 67
原子的 68
構造を持つ 68
構築 251
表示 282

い
インスタンス
クラス 440, 444
変数 440
インターフェース 479

え
演算子
結合法則 307
優先順位 307

お
オブジェクト 440
コピー 475
生成 444

か
ガーベージコレクション ...... 参照: GC
ガーベージコレクタ .... 参照: GC
仮想
クラス 455
メソッド 455
型
オブジェクト 454
制約 458, 459
開いた 454, 457
型変数
弱い 73
関数
再帰的
トレース 276
多相
トレース 278

く
クラス 440
インスタンス 444
インターフェース 479
仮想 455
型 457
型パラメータを持つ 464
局所宣言 480
継承 447
多重継承 461
抽象 455
開いた型 457
変数 481
クラスインスタンス 444
クリティカルセクション 606

け
継承 447

こ
構文解析 297
ocamlyacc 305
衝突 302, 307
シフト簡約衝突 303
ストリーム 299, 308
トップダウン 299
ボトムアップ 302
コンパイラ
デバッグ 280
プロファイリング 283

さ
参照 72

し
識別子 290
字句解析 290
ocamllex 295
ストリーム 291
シグナル 594
字句要素 297
ocamlyacc 306
実行スタック 282

集約 445
衝突 . . . . 参照: 構文解析

す
スタック 250
スタベーション 609
ストップ・アンド・コピー 258
ストリーム
　構文解析 299, 308
　字句解析 291
スレッド . . . . 参照: 軽量プロセス

せ
生産者-消費者 609
生成規則 . . . . 参照: 文法
セマフォア 611

そ
相互排除 606
束縛
　遅延 450
ゾンビ 589

た
代入 72
タグビット 255
多相性
　包含的 470

ち
チャネル 592
抽象
　クラス 455
　メソッド 455

つ
通信チャネル 75

て
デストラクタ 252
デバッガ . . . . 参照: デバッグ
デバッグ 275, 280

と
同期 606
トレース 275

は
パイプ 591
　名前付き 592
配列 66

ひ
ヒープ領域 250
評価
　評価順序 84
表現
　正規 292
　正則 292

ふ
ファイル
　拡張子
　　.mll 295
　　.mly 305
部分型 470
プロセス 582
　軽量 602
　生成 584, 604
プロファイリング 282
　機械語コード 284, 286
　バイトコード 283, 285
文法 297
　規則 297
　定義 297
　文脈依存 308

へ
並行性 601, 602
ベクトル . . . . 参照: 配列
変更可能 66

ほ
ポインタ 72
　弱い 267
包含的
　多相性 470
保守的 262

ま
マーク・アンド・スイープ 256

め
メソッド 440
　仮想 455
　抽象 455
メッセージ送信 440
メモリ

解放 250, 251
管理 249
キャッシュ 267
再利用
暗黙 253
明示的 252
自動的
解放 249
静的 250
動的 250
明示的な割り当て 251
割り当て 250

も
モジュール
依存性 274
文字列 70

る
ルート 253

れ
レコード
変更可能フィールド 71
連接 77

わ
割り当て
静的 250
動的 250

言語要素索引

Symbol
& 15
&& 15
! 72
[< 713
[> 713
() 16
** 13
*. 13
* 13, 16
+. 13
+ 13
-. 13
-¿ 21
- 13
/. 13
/ 13
:: 18
:= 72
:¿ 470
: 30, 708
; 77
¡- 67, 70, 71
¡= 15
¡¿ 15
¡ 15
== 15
= 15
¿= 15
¿} 475
¿ 15
? 710
@ 18
[] 17
444, 458
% 223
ˆ 14
_ 35
{ ¡475 459
708
‘ 713
—— 15

A
accept 632
acos 14
add_ available_ units 242
add_ interfaces 242
alarm 596
alloc.h 325
allow_ unsafe_ modules 242
and (keyword) 19, 41
append 218
Arg (module) 236
argv 235
Arith_ status (module) 241
Array (module) 66, 217, 218, 221
array (type) 66
as (keyword) 37, 458, 701
asin 14

assoc 151, 221
assq 221
atan 14

B
background 120
big_int (type) 240
bind 630, 631
blit 222
blit_image 125
bool (type) 15
bprintf 224
broadcast 611
Buffer (module) 217
button_down 133

C
Callback (module) 343
catch 239
ceil 14
char (type) 14
char_of_int 14
chdir 235
check 268
class (keyword) 441
clear_available_units 242
clear_graph 119
close 580, 631
close_graph 119
close_in 75
close_out 75
close_process 593
color (type) 120
combine 151, 221
command 235
compact 266
concat 218
Condition (module) 611
connect 630, 632
constraint (keyword) 459
copy 222, 476
cos 14
create 66, 222, 268, 604, 606, 611
create_image 125
create_process 585
current_point 120

D
delay 605
Delayed 106
descr_of_in_channel 581
descr_of_out_channel 581
Digest (module) 223, 227
do (keyword) 79
done (keyword) 79
downto (keyword) 79
draw_arc 121
draw_circle 121
draw_ellipse 121
draw_image 125
dump_image 125
dup 580
dup2 580
Dynlink (module) 242

E
else (keyword) 18
end (keyword) 414, 441
End_of_file 74
eprintf 224
error 242
error (type) 577
error_message 577
establish_server 635
Event (module) 615
event 132
exception (keyword) 54
exists 51, 220
exit 605
exn (type) 54
exp 14
external (keyword) 320

F
failwith 54
false 15
file 227
file_exists 235
Filename (module) 238
fill 222
fill_poly 121
fill_rect 121
filter 221
find 221
find_all 221
flatten 220

float (type) 12
float_ of_ string 14
floor 14
fold_ left 51, 219
fold_ right 219
for (keyword) 79
for_ all 26, 220
force 106
foreground 120
Format (module) 223
format (type) 224, 226
fprintf 224
from_ channel 229
from_ string 229
fst 17
full_ major 266
fun (keyword) 23
function (keyword) 21
functor (keyword) 423

G
Gc (module) 265
Genlex (module) 290
get 218, 265, 268
get_ image 125
getcwd 235
getenv 235
gethostbyaddr 629
gethostbyname 629
gethostname 628
getservbyname 629
getservbyport 629
global_ replace 294
Graphics (module) 117

H
handle_ error 577
Hashtbl (module) 217, 227
hd 18, 220
host_ entry (type) 628

I
if (keyword) 18
image 125
in (keyword) 20
in_ channel 74
in_ channel_ of_ descr 581
inet_ addr (type) 628
inet_ addr_ of_ string 628
init 179, 242
initializer (keyword) 452
input 75
input_ line 75
int 179
int (type) 12
int_ of_ char 14
int_ of_ string 14
interactive 235
iter 218
iter2 221
iteri 222

K
key_ pressed 133
kill 594, 604

L
labltk (command) 716
Lazy (module) 106
lazy (keyword) 106
length 218
let (keyword) 19, 20
lexbuf (type) 295
Lexing (module) 295
lineto 121
List (module) 18, 217, 218, 220
list (type) 17
listen 630, 632
loadfile 242
loadfile_ private 242
lock 606
log 14
log10 14
lseek 581

M
major 266
make 222
make_ image 125
make_ lexer 291
make_ matrix 222
Map (module) 424
map 26, 51, 218
map2 221
mapi 222
Marshal (module) 223, 229

match (keyword) 33, 110
Match_ Failure 36
matched_ string 294
max_ array_ length 235
mem 52, 220
mem_ assoc 221
mem_ assq 221
memory.h 325
memq 52, 220
method (keyword) 441
minor 266
mkfifo 592
mlvalues.h 325
mod 13
module (keyword) 414
module type (keyword) 414
mouse_ pos 133
moveto 120
mutable (keyword) 71
Mutex (module) 606

N
new (keyword) 444
next 110
None 267
not 15
nth 218
Num (module) 240
num (type) 240

O
object (keyword) 441
ocaml (command) 201, 205
ocamlbrowser (command) 716
ocamlc (command) 201, 202, 204
ocamlc.opt (command) 201
ocamldebug (command) 280
ocamldep (command) 274
ocamllex (command) 295, 313
ocamlmktop (command) .. 118, 201, 206
ocamlopt (command) 201
ocamlopt.opt (command) 201
ocamlrun (command) 201, 207
ocamlyacc (command) 305, 313
of (keyword) 44
of_ channel 109
of_ list 150, 223
of_ string 109
open (keyword) 214, 413
open_ connection 638
open_ flag (type) 579
open_ graph 119
open_ in 75
open_ out 75
open_ process 592
openfile 579
option (type) 267
or 15
OS_ type 235
out_ channel 74
out_ channel_ of_ descr 581
output 75

P
parse 237
parser (keyword) 110, 291
partition 221
Pervasives (module) 215
pipe 591
plot 121
point_ color 120
print 239
print_ newline 75
print_ stat 265
print_ string 75
Printexc (module) 239
Printf (module) 223
printf 224
private (keyword) 453
process_ status (type) 590

Q
Queue (module) 217

R
raise (keyword) 55
Random (module) 216
ratio (type) 240
read 580
read_ key 133
read_ line 75
rec (keyword) 27
receive 615
-rectypes 703
ref (type) 72
regexp 294

register 343
remove 235
remove_ assoc 221
remove_ assq 221
rename 235
rev 220
rev_ append 220
rgb (type) 120

S
search_ forward 294
seek_ command (type) 581
send 615
service_ entry (type) 629
Set (module) 424
set 221, 265, 268
set_ binary_ mode_ in 581
set_ binary_ mode_ out 581
set_ color 120
set_ font 120
set_ line 120
set_ signal 595
set_ text_ size 120
shutdown_ connection 638
sig (keyword) 414
sigalrm 596
sigchld 597
sigint 595
signal 595, 611
signal_ behavior (type) 594
sigusr1 597
sigusr2 597
sin 14
sleep 588
snd 17
SOCK_ STREAM 630
sockaddr (type) 631
socket 630
socket (type) 630
socket_ domain (type) 630
socket_ type (type) 630
Some 267
Sort (module) 217
split 221
sprintf 224
sqrt 14
Stack (module) 217, 410
Stack_ overflow (exception) 93
stat 265
status 132
stderr 74, 577
stdin 74, 577
stdout 74, 577
Str (module) 294
Stream (module) 108
stream (type) 108
String (module) 217
string 227
string (type) 14
string_ of_ float 14
string_ of_ inet_ addr 628
string_ of_ int 14
struct (keyword) 414
sub 222
sync 615
Sys (module) 235
Sys_ error 75

T
tan 14
then (keyword) 18
Thread (module) 604
ThreadUnix (module) 641
time 179, 235
tl 18, 220
to (keyword) 79
to_ buffer 229
to_ channel 229
to_ list 223
to_ string 229, 239
token (type) 290
#trace (directive) 275
true 15
try (keyword) 56
try_ lock 607
type (keyword) 41

U
unit (type) 16
Unix (module) 576
Unix_ error 577
unlock 606
#untrace (directive) 275
#untrace_ all (directive) 275

V

val (keyword) 412, 441
val mutable (keyword) 441
Value 106
value 320, 326
virtual (keyword) 455

W

wait 589, 611
wait_ next_ event 132
wait_ time_ read 648
wait_ time_ write 648
waitpid 590
Weak (module) 217, 267
when (keyword) 38
while (keyword) 79
with (keyword) 33, 44, 56, 110, 421
word_ size 235
write 580

Liste de diffusion par messagerie électronique

Si vous souhaitez recevoir périodiquement les annonces de nouveaux produits O'Reilly en français, il vous suffit de souscrire un abonnement à la liste d'annonces

```
parutions-oreilly
```

Merci d'expédier en ce cas un message électronique à `majordomo@ora.de` contenant (dans le corps du message) :

```
subscribe parutions-oreilly votre_adresse_email
```

Exemple :

```
subscribe parutions-oreilly jean.dupond@ici.fr
```

Cette liste ne véhicule que des annonces et non des *discussions*, vous ne pourrez par conséquent pas y poster. Son volume ne dépasse pas quatre messages par mois.

En cas de problème technique écrire à

```
parutions-oreilly-owner@ora.de
```

Site Web

Notre site Web `http://www.editions-oreilly.fr/` diffuse diverses informations :

- le catalogue des produits proposés par les éditions O'Reilly,
- les *errata* de nos ouvrages,
- des archives abritant les exemples,
- la liste des revendeurs.