VDM-SL，VDM++簡易リファレンス

国立情報学研究所 石川 冬樹

平成 23 年 6 月 28 日

本書では，VDM-SL Toolbox および VDM++ Toolbox における文法定義に基づき[1]，VDM-SL，VDM++の代表的な文法をまとめる．各記述者が与える部分（キーワードではない部分）についてはイタリック書体で示している．各構文の詳細や網羅的な文法定義については，それぞれの言語マニュアルを参照していただきたい（ツールの Web サイトから取得可能[2]）．

目 次

[1] バージョン 8.1 時点のマニュアルを参照した．
[2] http://www.vdmtools.jp/, http://www.overturetool.org/

1　組み込み型

以下に VDM++における型を示す．相等判定の演算子（ = ）と不等判定の演算子（ <> ）は共通して用いられる．

1.1　基本データ型

1.1.1　bool： ブール型

値 `true`, `false`

演算子 表 1 および相等・不等判定演算子

演算子	意味
`not` *b*	*b* ではない（否定）
a `and` *b*	*a* と *b* の両方が成り立つ（論理積）
a `or` *b*	*a* と *b* の少なくとも片方が成り立つ（論理和）
a `=>` *b*	*a* が成り立つならば *b* が成り立つ（ `not` *a* `or` *b* と同値）（含意）
a `<=>` *b*	*a* と *b* の真偽値が一致する（同値）

表 1: bool 型に対する演算子

1.1.2　real・rat・int・nat・nat1： 数値型

型 real（実数型），rat（有理数型），int（整数型），nat（0 以上の自然数型），nat1 （1 以上の自然数型）

値 `3`, `-2.5`, `0`, `11`, ...

演算子 表 2 および相等・不等判定演算子

演算子	意味
`-` *x*	負符号（*x* の符号を反転した値）
`abs` *x*	*x* の絶対値（absolute value）
`floor` *x*	*x* を超えない最大の整数
x `+` *y* （`-`, `*`, `/`）	四則演算（加算，減算，乗算，除算）
x `div` *y*	整数除算の商（integer division）
x `mod` *y* （`rem`）	整数除算の余り（`mod` は法・modulus，`rem` は剰余・remainder）
x `**` *y*	べき乗（*x* の *y* 乗）
a `<` *b* （`>`, `<=`, `>=`）	不等号

表 2: 数値型に対する演算子

1.1.3　char： 文字型

文字列は char 型の値の列として表現される（1.2.6 節）．

値 `'a'`, `'b'`, ..., `'1'`, ...

演算子 相等・不等判定演算子

1.1.4 引用型

合併型（1.2.1 節）で用いられる.

値 `<red>`, `<Japan>`, `<error-mode>`, ...

演算子 相等・不等判定演算子

1.1.5 token：トークン型

値 `mk_token(3)`, `mk_token({-1, true})`, `mk_token("VDM")`, ...（mk_token により値を表現）

演算子 相等・不等判定演算子

1.2 複合データ型

1.2.1 合併型

型の宣言 `char | int , <Japan> | <France> | <USA>`, ...

値 列挙されている型の値のいずれか

演算子 相等・不等判定演算子

1.2.2 選択型（nil の許容）

型の宣言 `[bool], [<Japan> | <France> | <USA>]`, ...

値 列挙されている型の値または nil

1.2.3 組型

型の宣言 `nat * bool * char, seq of char * bool`, ...

値 `mk_(3, true, 'a'), mk_({mk_token("VDM"), true})`, ...（mk_ により値を表現）

演算子 相等・不等判定演算子および下記の演算子

- 組 *t* の *n* 番目の要素を表す *t*`.#`*n*

1.2.4 レコード型

型の宣言

```
compose RecordA of
  key : nat
  description : seq of char
end
```

（ここで：の部分を `:-` とすると，相等判定・不等判定の際にその要素が無視される）

型の宣言（別形式）

```
RecordA ::
  key : nat
  description : seq of char
```

値 `mk_RecordA(5, "VDM++")`, `mk_RecordA({1, "B Method"})`, … （ mk_ の後にレコード名を添えて値を表現）

演算子 相等・不等判定演算子および下記の演算子

- レコード *r* の識別子 *i* を持つ要素の値を表す *r.i*

固有の式

- レコード *r* の要素の値を，識別子から値への対応づけを与えて更新したものを返す `mu` 式
  `mu(` *r*, *key*1|− > *val*1, *key*2− > *val*2, ... `)`
- レコード型 *R* かどうかを判別する `is_`*R* 式

1.2.5 集合型

値 `{1, 2, 3}`, `{'a', 'c', 'x'}`, `{ }`, …

演算子 表 3 および相等・不等判定演算子

固有の式

- 内包的定義
  `{` *exp* `|` *pat*1*a*, *pat*1*b*, ... `in set` *s*1, *pat*2*a*, *pat*2*b*, ... `in set` *s*2, `...`
  `&` *cond* `}`
  （集合による束縛 `in set` *S* ではなく型による束縛：*T* も記述できる）
- 整数 *m* 以上で，整数 *n* 以下である整数をすべて含む集合 {*m*, ..., *n*}

演算子	意味	演算の例と結果	
e in set *s*	*e* が *s* の要素であるか	`3 in set {1,2,3}`	true
e not in set *s*	*e* が *s* の要素ではないか	`3 in set {1,2,3}`	false
*s*1 union *s*2	*s*1 と *s*2 の和集合	`{1,2,3} union {2,4}`	`{1,2,3,4}`
*s*1 inter *s*2	*s*1 と *s*2 の共通部分	`{1,2,3} inter {2,4}`	`{2}`
*s*1 *s*2	*s*1 と *s*2 の差集合	`{1,2,3}\{2,4}`	`{1,3}`
*s*1 subset *s*2	*s*1 が *s*2 の部分集合であるか（等しくてもよい）	`{2,3} subset {1,2,3}`	true
*s*1 psubset *s*2	*s*1 が *s*2 の真部分集合であるか（等しい場合は除く）	`{2,3} psubset {2,3}`	false
card *s*	*s* の濃度（要素数）	card `{2,5,6}`	3
dunion *ss*	集合の集合 *ss* 内の全要素の和集合	`dunion {{1,2,3}, {2,4},{1,2,5}}`	`{1,2,3, 4,5}`
dinter *ss*	集合の集合 *ss* 内の全要素の共通部分	`dunion {{1,2,3}, {2,4},{1,2,5}}`	`{2}`
power *s*	*s* のべき集合（すべての部分集合の集合）	`power {1,2}`	`{{},{1}, {2},{1,2}}`

表 3: 集合に対する演算子

1.2.6 列型

値 `[1, 2, 3]`, `['a', 'c', 'x']`, `[ ]`, ...

演算子 表 4 および相等・不等判定演算子

固有の式

- 内包的定義

  `[` *exp* `|` *pat*1*a*, *pat*1*b*, ... `in set` *s*1, *pat*2*a*, *pat*2*b*, ... `in set` *s*2, `...`
  & *cond* `]`

  （束縛に用いている集合 *s*1, *s*2, ⋯ は数値の集合）

- 列 *l* の *m* 番目の要素から *n* 番目の要素までの部分列を抜き出す *l*(*m*, ..., *n*)

演算子	意味	演算の例と結果	
l(*i*)	*l* の *i* 番目の要素	`['a','b','c'](2)`	'b'
hd *l*	空列ではない列 *l* の先頭要素	`hd [1,2,3]`	1
tl *l*	空列ではない列 *l* の先頭要素を除いた列	`tl [1,2,3]`	[2,3]
len *l*	*l* の長さ	`[1,2,3]`	3
elems *l*	*l* の要素からなる集合	`[1,2,3,2,3]`	`{1,2,3}`
inds *l*	*l* のインデックスの集合	`[1,2,3,2,3]`	`{1,2,3,4,5}`
*l*1 ^*l*2	*l*1 と *l*2 を連結した列	`[1,2]^[3,2]`	`{1,2,3,2}`
conc *ll*	列の列 *ll* に含まれる列をすべて連結した列	`conc [[1,2], [3,2],[4,1]]`	`[1,2,3,2, 4,1]`
l ++ *m*	インデックスとそれに対応する変更後の値を示す写像 *m* を用い，その対応づけに従って *l* の要素の値を変更した列	`['a','b','c','d'] ++ {1|->'x',4|->'y'}`	`['x','b', 'c','y']`

表 4: 列に対する演算子

1.2.7 写像型

値 `{1 |-> true, 2 |-> false, 3 |-> true}`,
`{'A' |-> "41", 'B' |-> "42"}`, `{ |-> }`, ...

演算子 表 5, 表 6 および相等・不等判定演算子

固有の式

- 内包的定義

  `{` *e*1 `|->` *e*2 `|` *pat*1*a*, *pat*1*b*, ... `in set` *s*1, *pat*2*a*, ... `in set` *s*2, `...`
  & *cond* `}`

  （集合ではなく型による束縛もあるが，実行できないため省略している）

演算子	意味	演算の例と結果
m(*d*)	*m* のキー *d* に対応する値	`{1|->'a',2|->'b'}(2)` 'b'
dom *m*	*m* の定義域（キーの集合）	`dom {1|->'a',2|->'b'}` `{1,2}`
rng *m*	*m* の値域（対応づけられる値の集合）	`rng {1|->'a',2|->'b'}` `{'a','b'}`
inverse *m*	*m* のキーと値を逆にした写像（*m* は inmap）	`inverse {1|->'a', 2|->'b'}` `{'a'|->1, 'b'|->2}`
s <: *m*	*m* の関連づけのうちキーが *s* に含まれるもののみ抜き出した写像	`{1,2} <: {1|->'a',2|->'b', 3|->'c'}` `{1|->'a', 2|->'b'}`
s <-: *m*	*m* の関連づけのうちキーが *s* に含まれないもののみ抜き出した写像	`{1,2} <-: {1|->'a',2|->'b', 3|->'c'}` `{3|->'c'}`
m :> *s*	*m* の関連づけのうち値が *s* に含まれるもののみ抜き出した写像	`{1|->'a',2|->'b', 3|->'c'} :> {'a','b'}` `{1|->'a', 2|->'b'}`
m :-> *s*	*m* の関連づけのうち値が *s* に含まれないもののみ抜き出した写像	`{1|->'a',2|->'b', 3|->'c'} :-> {'a','b'}` `{3|->'c'}`

表 5: 写像に対する演算子（1）

*m*1 munion *m*2	*m*1 と *m*2 に含まれる関連づけを両方含む写像（同じキーに違う値が結びつけられていない必要がある）	`{1|->'a',2|->'b'} munion {1|->'a',3|->'c'}`	`{1|->'a', 2|->'b', 3|->'c'}`
*m*1 ++ *m*2	*m*1 を *m*2 に含まれる関連づけで上書きする	`{1|->'a',2|->'b'} ++ {1|->'x',3|->'c'}`	`{1|->'x', 2|->'b', 3|->'c'}`
merge *ms*	写像の集合 *ms* に含まれるすべての写像を munion により結合	`merge {{1|->'a',2|->'b'}, {1|->'a',3|->'c'},} {2|->'b',4|->'d'}}`	`{1|->'a', 2|->'b', 3|->'c', 4|->'d'}`
*m*1 comp *m*2	map A to B である *m*2 と map B to C である *m*1 を合成し，map A to C を作成	`{1|->'a',2|->'b'} comp {'x'|->1,'y'|->2}`	`{'x'|->'a', 'y'|->'b'}`
m ** *n*	*m* の対応づけを自身に *n* 回適用（定義域と値域は同じ型であるとする）	`{1|->2,2|->3} {3|->1} ** 2`	`{1|->3, 2|->1, 3|->2}`

表 6: 写像に対する演算子（2）

1.2.8 関数型

型の宣言 `int +> bool`, `int * seq of char +> ()`, …
（ `->` は部分関数を， `+>` は全関数をそれぞれ表す）

演算子 相等・不等判定演算子および下記の演算子（ただし相等・不等判定は理論上・概念上は定義されるが，インタプリタ上では基本的に計算不能と考えた方がよい.）

- 関数 *f* に引数 *a*1，*a*2，…，*an* を適用する $f(a1, a2, ..., an)$
- 関数 *f*2 を適用した後に関数 *f*1 を適用する合成関数を表す *f*1 `comp` *f*2
- 引数と戻り値の型が同一の関数 *f* を *n* 回適用する *f* `**` *n*

2　式・文

以下に VDM++における式・文の構文を示す．式と文に共通して利用できる構文，式固有の構文，文固有の構文の順に挙げる．

2.1　if 式・文

```
if cond1 then e1
elseif cond2 then e2
else e3

if cond1 then s1
elseif cond2 then s2
else s3
```

*cond*1 が成り立つ際に式 *e*1（文 *s*1）を，*cond*2 が成り立つ際に式 *e*2（文 *s*2）を，その他の場合に式 *e*3（文 *s*3）を評価する．elseif は省略可能である．VDM++ Toolbox の文法では if 式における else も省略可能としている（VDM-SL 標準では省略可能ではない）．

2.2　cases 式・文

```
cases e :
  p11, p12, ..., p1n    -> e1,
  p21, p12, ..., p2n    -> e2,
    ...                      ,
  pm1, pm2, ..., pmn -> em,
  others                -> eo
end

cases e :
  p11, p12, ..., p1n    -> s1,
  p21, p12, ..., p2n    -> s2,
    ...                      ,
  pm1, pm2, ..., pmn -> sm,
  others                -> so
end
```

式 *e* を評価した結果がパターン *p*11，*p*12, … *p*1*n* のいずれかにマッチする場合には式 *e*1（文 *s*1）を，というようにマッチするパターンに対応する式（文）を評価する．どれにもマッチしない場合 others に対応する式（文）を評価する．others は省略可能である．

2.3　let 式・文

```
let p1 : T1 = e1, p2 : T2 = e2, ..., pn : Tn = en in e
let p1 : T1 = e1, p2 : T2 = e2, ..., pn : Tn = en in s
```

型 *T*1 をとるパターン *p*1 に式 *e*1 を，パターン *p*2 に式 *e*2 を，といったように束縛を行い（パターン中に含まれる識別子の値を定め），式 *e*（文 *s*）を評価する．型の記述は省略可能である．let 式・文では一時的な関数定義を埋め込むこともできるがここでは省略する．

2.4 def 式・文

```
def p1 : T1 = e1; p2 : T2 = e2; ...; pn : Tn = en in e
def p1 : T1 = e1; p2 : T2 = e2; ...; pn : Tn = en in s
```

操作の中でのみ用いる．型 *T*1 をとるパターン *p*1 に式 *e*1 を，パターン *p*2 に式 *e*2 を，といったように束縛を行い（パターン中に含まれる識別子の値を定め），式 *e*（文 *s*）を評価する．型による束縛はオプショナルであり，また `in set` *S* という形式で集合束縛を書くこともできる．

2.5 let be 式・文

```
let p in set S be st cond in e
let p in set S be st cond in s
```

パターン *p* が集合 *S* の要素とマッチし，かつその束縛を行った際に *cond* を満たすような束縛を選び，式 *e*（文 *s*）を評価する．集合束縛の部分（ `in set` *S* ）は型束縛（ `:` *T* ）でもよい．

2.6 限量式

2.6.1 forall 式

```
forall pat1a, pat1b, ... in set s1,
       pat2a, pat2b, ... in set s2,
       ...
       patna, patnb, ... in set sn,
    & cond
```

forall 式においては，列挙されたパターン *pat*1*a*，*pat*1*b*, … をそれぞれ対応する集合 *s*1，*s*2, … のどの要素に対してマッチし束縛させたとしても，式 *cond* の評価が true となるかどうか，を bool 値にて返す．

2.6.2 exists 式

```
exists pat1a, pat1b, ... in set s1,
       pat2a, pat2b, ... in set s2,
       ...
    & cond
```

exists 式においては，列挙されたパターン *pat*1*a*，*pat*1*b*, … をそれぞれ対応する集合 *s*1，*s*2, … のどの要素に対してマッチし束縛させたとしても，式 *cond* の評価が true となるような束縛が存在するかどうか，を bool 値にて返す．

2.6.3 exists1 式

```
exists1 pat1a, pat1b, ... in set s1,
        pat2a, pat2b, ... in set s2,
        ...
    & cond
```

exists1 式においては，列挙されたパターン *pat*1*a*，*pat*1*b*, … をそれぞれ対応する集合 *s*1，*s*2, … のどの要素に対してマッチし束縛させたとしても，式 *cond* の評価が true となるような束縛がただ一つ存在するかどうか, を bool 値にて返す.

2.7 ラムダ式

```
lambda pat1 : T1, pat2 : T2, ..., patn : Tn & e
```

受け取った引数を *pat*1 から *patn* にパターンマッチして *e* を評価するような陽関数定義に相当する．*T*1 から *Tn* はそれぞれのパターンの型を示す．ただし，上記は厳密には「*n* 個の引数を受け取る関数」というよりも「1 個の引数を受け取り,『*n* – 1 個の引数を受け取る関数』を返す関数」と見なされる．その際には左側の引数から適用される.

2.8 その他の式

その他以下のような式が利用可能である.

iota 式 集合の中に条件を満たす要素がただ 1 つある場合のみ実行でき，その要素を取り出す.

is 式 `is_bool(x)`，`is_token(x)` などとしてある値がある型に属するかを判定する．`is(x, bool)` といった記法もある.

new 式（VDM++のみ） キーワード `new` に続いてコンストラクタ呼び出しを記述することにより新たなインスタンスを生成し，その参照が返される.

クラス判定式（VDM++のみ） ある値があるクラス（の子孫クラス）に属するかの判定式などが利用可能である（isofclass，isofbaseclass，samebaseclass，sameclass）.

undefined 値が未定義であることを表す際に用いる.

2.9 代入文

2.9.1 代入文

```
var := e
```

var で指定される変数や変数を用いた参照の値を式 *e* の評価により得られる値に置き換える.

2.9.2 多重代入文

```
atomic(
  var1 := e1
  var2 := e2
  ...
  varn := en
)
```

不変条件の評価に対して原子的に，代入文を実行する．

2.10 ブロック文

```
(
  dcl var1:T1 := e1, var2:T2 := e2, ..., varn:Tn := en;
  s1;
  s2;
  ...
  sn
)
```

dcl 部分において一時変数を定義し，文 *s*1，*s*2，…，*sn* を順次実行する．dcl 部分は省略可能であり，また各一時変数の初期値の設定も省略可能である．ブロック文においては，途中で戻り値を返す文があると，その時点でブロック文全体の返り値が出たとして実行が終了する．

2.11 for 文

2.11.1 整数インデックスに対する for 文

```
for id = e1 to e2 by e3 do s
```

整数の値を持つ識別子 *id* の値を，*e*1 から始めて *e*3 ずつ増やしながら *e*2 になるまで繰り返し *s* を実行する．

2.11.2 集合に対する for 文

```
for all pat in set S do s
```

集合 *S* のすべての要素を一つずつパターン *pat* にマッチし束縛させ，文 *s* を実行する．

2.11.3 列に対する for 文

```
for all pat in l do s
```

列 *l* のすべての要素を順にパターン *pat* にマッチし束縛させ，文 *s* を実行する．`in reverse` *l* と書くことにより逆順にたどらせることもできる．また，*pat* の部分において束縛（*pat* `in set` *S* や *pat* : *T*）を用いることにより，条件を満たす要素のみ抜き出した部分列に対する繰り返しを記述することもできる．

2.12 while 文

```
while cond do s
```

条件式 *cond* が成り立つ間文 *s* を繰り返し実行する．

2.13 例外処理

2.13.1 trap 文

```
trap pat with s1 in s2
```

文 *s*2 の評価を行い，例外が発生しない場合は *s*2 の評価結果がこの `trap` 文全体の評価値となる．例外が発生した場合，それがパターン *pat* とマッチするならば，例外処理内容を表す文 *s*1 が評価される．捕捉（マッチ）されなかった例外は，この `trap` 文の評価結果となり，上位の呼び出し元に返される．

2.13.2 tixe 文

```
tixe
  pat1 |-> s1,
  pat2 |-> s2,
  ...                  ,
  patn |-> sn

in s
```

最初に文 *s* の評価を行い，例外が発生しない場合は *s* の評価結果がこの `tixe` 文全体の評価値となる．例外が発生した場合，それがパターン *pat*1，*pat*2，⋯，*patn* と順にマッチされ，マッチが成功される場合は対応する例外処理文 *si* が評価される．もしもこの文でも例外が発生した場合，同様の例外マッチを再帰的に行う．

2.13.3 exit 文

```
exit e
```

例外を発生させる．式 *e* は例外の種類を示す値として，この例外を捕捉するかどうかの判定に用いられる（オプショナル）．

2.13.4 always 文

```
always s1 in s2
```

文 *s*2 の評価終了後に，例外発生の有無にかかわらず文 *s*1 を評価する．

2.14 その他の文

その他以下のような式が利用可能である．

return 文 操作内において `return` *exp* として式の値を返す．戻り値がない場合には式の値は与えない．

skip 文 何も行わない文を表す．

error 文 エラーを起こして終了することを表す．

非決定文 複数の文を任意の順序で実行することを表す．

仕様記述文 陰操作定義のように外部節，事前条件，事後条件により抽象的に文を定義する（実行できない）．

start 文（VDM++のみ） オブジェクトにスレッドを割り当てて実行する．

startlist 文（VDM++のみ） オブジェクトの集合のそれぞれにスレッドを割り当てて実行する．

3 仕様記述

3.1 型定義ブロック

型定義ブロックでは， `types` に続いて任意の個数の型定義を行う．各型定義は不変条件を与えない場合以下の形式をとる．

*Tname*1 `=` *T*1`;`

不変条件を与える場合以下の形式となる．

*Tname*1 `=` *T*1
`inv` *pat* `==` *cond*`;`

*Tname*1 は定義する型の名前，*T*1 はその名前に結びつける型の表現である．不変条件は，その方の値をパターン *pat* にマッチして条件式 *cond* を評価したときに常に真である，と与えられる．VDM++の場合アクセス修飾子と `static` キーワードを，型名 *Tname*1 の宣言の前につけることができる（どちらが先でもよい）．レコード型を用いる場合 = 記号ではなく `::` 記号を用いた簡易記法が提供されている（付録 1.2.4 におけるレコード型の説明を参照）．

3.2 定数定義ブロック

定数定義ブロックでは， `values` に続いて任意の個数の定数定義を行う．各定数定義は以下の形式をとる．

*pat*1 `:` *T*1 `=` *exp*1`;`

式 *exp*1 の値をパターン *pat*1 にマッチし，その中に含まれる識別子に定数値を束縛する．型 *T*1 の宣言は省略可能である．VDM++の場合アクセス修飾子と `static` キーワードを *pat*1 の前につけることができる（どちらが先でもよい）．

3.3 状態定義ブロック（VDM-SL のみ）

状態定義ブロックでは，`state` に続いて状態を構成するレコード構造の定義，および不変条件定義，初期化定義を行う．

状態定義は以下の形式をとる．

```
state statename of
  id1 : T1
  id2 : T2
  ...
  idn : Tn
inv inv_pat == inv_cond
init init_pat == init_cond
end
```

状態 *statename* を構成する構成要素の識別子（*id*1 など）とその型名（*T*1 など）を列挙する．不変条件は，その方の値をパターン inv_pat にマッチして条件式 inv_cond を評価したときに常に真である，と与えられる．初期化も同様である（$init_pat$，$init_cond$）．不変条件，初期化は省略可能である．

3.4 インスタンス変数定義ブロック（VDM++のみ）

インスタンス変数定義ブロックでは，`instance variables` に続いて任意の個数のインスタンス変数定義および不変条件定義を行う．

各インスタンス変数定義は不変条件を与えない場合以下の形式をとる．

```
var1 : T1 = exp1;
```

型 *T*1 に属する変数名 *var*1 を宣言する．初期値の代入 `:=` *exp*1 は省略可能である．型 *T*1 の宣言は省略可能である．VDM++の場合アクセス修飾子と `static` キーワードを *var*1 の前につけることができる（どちらが先でもよい）．

各不変条件定義は以下の形式をとり，条件式 *cond* を与える．

```
inv cond;
```

3.5 関数定義ブロック

関数定義ブロックでは，`functions` に続いて任意の個数の関数定義を行う．各関数は以下のいずれかの形式をとる．以下において改行は書面における見やすさのためであり実際には必要ない．

陰関数定義は以下の形式をとる．

```
fun1
  [@tv1, @tv2, ..., @tvm]
  (pat1 : T1, pat2 : T2, ..., patn : Tn)
  varret : Tret
  pre precond
  post postcond;
```

1 行目の *fun*1 は関数名を表す．2 行目の @*tv*1 などは，多相関数の定義に用いる型変数を表している（省略可能である）．3 行目の `pat1` および `T1` などは引数とその型を表す．4 行目の *varret* および *Tret* は戻り値に与える名前とその型を表す．5 行目は事前条件を表す式 *precond* を与えている（省略可能である）．6 行目は事後条件を表す式 *postcond* を与えている．最後には必ずセミコロンで定義を終了する．

拡張陽関数定義は以下の形式をとる．

```
fun1
  [@tv1, @tv2, ..., @tvm]
  (pat1 : T1, pat2 : T2, ..., patn : Tn)
  varret : Tret
  == exp
  pre precond
  post postcond;
```

陰関数定義に加えて，5 行目に関数本体の式 *exp* が加わったものになっている．陰関数定義と異なり事後条件が省略可能である．最後には必ずセミコロンで定義を終了する．

陽関数定義は以下の形式をとる．

```
fun1
  [@tv1, @tv2, ..., @tvm]
  : T1 * T2 * ... * Tn -> Tret
fun1 (pat1, pat2, ..., patn)
  == exp
  pre precond
  post postcond;
```

1 行目の *fun*1 は関数名を表す．2 行目の @*tv*1 などは，多相関数の定義に用いる型変数を表している（省略可能である）．3 行目では引数の型（ *T*1 など）と戻り値の型 *Tret* を示している．ここでは関数型の定義（1.2.8 節）を示しており，全関数の場合には -> ではなく+> を用いる．4 行目では引数のリスト（ *pat*1 など）を与えている．5 行目では関数本体を表す式 *exp* を与えている．6 行目は事前条件を表す式 *precond* を与えている（省略可能である）．7 行目は事後条件を表す式 *postcond* を与えている（省略可能である）．陽関数定義の事後条件においては，戻り値を意味するキーワード RESULT を用いることができる．最後には必ずセミコロンで定義を終了する．

VDM++の場合いずれにおいてもアクセス修飾子と `static` キーワードを *var*1 の前につけることができる（どちらが先でもよい）．また拡張陰関数定義および陽関数定義における関数本体においては，一般的な式のほか，is not yet specified または is subclass responsibility を与えることもできる．

3.6 操作定義ブロック

操作定義ブロックでは，`functions` に続いて任意の個数の操作定義を行う．各操作は以下のいずれかの形式をとる．以下において改行は書面における見やすさのためであり実際には必要ない．

陰操作定義は以下の形式をとる．

```
op1
  (pat1 : T1, pat2 : T2, ..., patn : Tn)
  varret : Tret
  ext
    mode1 var1a, var1b, ..., var1p : Te1
```

```
    mode2 var2a, var3b, ..., var2q : Te2
    ...
    modem varma, varmb, ..., varmr : Tem
  pre precond
  post postcond;
```

1 行目の *op*1 は操作名を表す．2 行目の `pat1` および `T1` などは引数とその型を表す．3 行目の *varret* および *Tret* は戻り値に与える名前とその型を表す（省略可能である）．4〜8 行目は外部節において状態変数（VDM-SL の場合）・インスタンス変数（VDM++の場合）へのアクセスを表すモード（ *mode*1 などの部分，値としては読み取り rd または書き込み wr ）を宣言している．外部節における型（*Te*1 など）は省略可能である．最後から 2 行目は事前条件を表す式 *precond* を与えている（省略可能である）．一番最後の行では事後条件を表す式 *postcond* を与えている．最後には必ずセミコロンで定義を終了する．

拡張陽操作定義は以下の形式をとる．

```
op1
  (pat1 : T1, pat2 : T2, ..., patn : Tn)
  varret : Tret
  == s
  ext
    mode1 var1a, var1b, ..., var1p : Te1
    mode2 var2a, var3b, ..., var2q : Te2
    ...
    modem varma, varmb, ..., varmr : Tem
  pre precond
  post postcond;
```

陰操作定義に加えて，4 行目に操作本体の文 *s* が加わったものになっている．陰操作定義と異なり事後条件が省略可能である．最後には必ずセミコロンで定義を終了する．

陽操作定義は以下の形式をとる．

```
op1
  : T1 * T2 * ... * Tn ==> Tret
op1 (pat1, pat2, ..., patn)
  == exp
  pre precond
  post postcond;
```

1 行目の *op*1 は操作名を表す．2 行目では引数の型（ *T*1 など）と戻り値の型 *Tret* を示している．3 行目では引数のリスト（ *pat*1 など）を与えている．4 行目では操作本体を表す式 *exp* を与えている．5 行目は事前条件を表す式 *precond* を与えている（省略可能である）．6 行目は事後条件を表す式 *postcond* を与えている（省略可能である）．陽操作定義の事後条件においては，戻り値を意味するキーワード RESULT を用いることができる．最後には必ずセミコロンで定義を終了する．

VDM++の場合いずれにおいてもアクセス修飾子と `static` キーワードを *var*1 の前につけることができる（どちらが先でもよい）．また事後条件においては，インスタンス変数の後ろににチルダ記号 ~ を用いて，その操作の実行前の変数の値を参照することができる．拡張陰操作定義および陽操作定義における操作本体においては，一般的な文のほか，is not yet specified または is subclass responsibility を与えることもできる．

3.7 同期制約定義ブロック（VDM++のみ）

同期制約定義ブロックでは， sync に続いて任意の個数の関数の定義を行う．各許可述語は以下のいずれかの形式をとる．

以下の形式では，操作 *op*1 の実行に関する同期制約として条件式 *cond*1 を与える．

```
per op1 => cond1
```

同期制約内では以下の履歴式を用いることができる．

#req(*op*)	操作 *op* の呼び出しが要求された回数
#act(*op*)	操作 *op* の呼び出しが起動され実行が開始した回数
#fin(*op*)	操作 *op* の呼び出しが起動され実行が完了した回数
#active(*op*)	現在実行中である *op* の個数 （ = #act(*op*) - #fin(*op*) ）
#waiting(*op*)	現在実行を待っている操作 *op* の呼び出しの個数 （ = #req(*op*) - #act(*op*) ）

以下の形式では，複数の操作に対して排他制御を指定する．

```
mutex(op1, op2, ..., opn)
```

以下の形式では，そのクラスおよび親クラス内において記述された操作すべてに対して排他制御が指定される．

```
mutex(all)
```

3.8 スレッド定義ブロック（VDM++のみ）

スレッド定義ブロックでは， thread に続いてそのクラスのインスタンスにスレッドが割り当てられた際に実行する文 *s* を高々一つ指定する．