# Chapter 6

# Text Constraints

The pattern language in LAPIS is called *text constraints* (TC). The TC language has the region algebra at its core, in much the same way that the regular expression pattern languages in Perl and awk are fundamentally based on regular expressions, but with additional operators to make the language expressive enough for practical use. Since the semantics of region algebra operators have already been presented in Chapter 3, the current chapter focuses largely on the syntax of the language.

TC incorporates some usability guidelines into its design, in the spirit of Natural Programming [Mye98]. TC avoids the keyword *and*, which is dangerously ambiguous for reasons discussed in the next section. TC also uses a simple, universal rule for operator precedence and associativity, and avoids most parentheses by deducing expression structure from indentation. Although indentation has been used to specify block structure in other languages (notably Python and Haskell), TC is the first language that uses indentation for structuring infix expressions.

The usability of TC was tested in a small user study, and the results of the user study were fed back into language design. The description of TC presented in this chapter includes the changes that resulted from the user study. These changes and the user observations that motivated them are discussed when the corresponding language features are presented in this chapter. However, since the user study evaluated not only TC but also other features of the LAPIS user interface in which TC is embedded, a complete discussion of the study itself is postponed until Chapter 7.

## 6.1 Goals

There were three primary goals in the design of TC.

- **Expressiveness.** The language should be able to combine and constrain lightweight structure using all the power of the region algebra.
- **Readability.** Novice users should be able to read and comprehend expressions in the language without difficulty, so that an intelligent system that infers patterns from examples can display the patterns as one form of feedback.
- **Low error rate.** Novice users should be able to write TC expressions without making the kinds of syntactic and logical errors that are common in other programming languages, particularly *and*/*or* confusion and operator precedence problems.

These goals were motivated by perceived weaknesses in the prevailing text pattern languages: regular expressions and grammars. Both regular expressions and grammars fail the expressiveness test, because they cannot express intersection or negation of patterns. Both also are notoriously difficult to read, partly because of their heavy use of punctuation operators (*, +, ?, [], $, ^, |), and partly because the languages' low expressiveness forces many patterns to be long and complicated. Finally, like most programming languages, regular expressions force users to memorize precedence and associativity rules, resulting in errors. Although recent research has improved the readability and writability of regular expressions by using a visual notation [Bla01, JB97], this work did nothing for expressiveness.

From a survey of the novice programming literature, Pane and Myers distilled a list of usability guidelines for designers of novice programming systems [PM96]. Of the 29 guidelines in 8 categories, the following were the most useful for designing TC:

- **Beware of misleading appearances.** The language should not encourage creation of expressions that can be easily misinterpreted. A classic example of this problem is a program whose indentation is inconsistent with its block structure. TC avoids this problem by treating indentation as grammatically significant, so that indentation affects both the user's interpretation and the system's interpretation of an expression.

- **Avoid subtle distinctions in syntax.** The language should avoid subtle syntactic distinctions that have drastically different semantics. In Perl, for example, the single quote (') and double quote (") look very similar but have considerably different meanings. In almost all languages, 123 and "123" are completely different types. In TC, all quote marks have the same semantics, and 123 and "123" both refer to the literal string pattern 123, since TC has no need to distinguish numeric constants from string constants.

- **Consistency in notation.** The language should be self-consistent, uniform, and free from unnecessary exceptions.

- **Naturalness of the programming language.** Draw on natural language where appropriate to make the language more familiar or mnemonic, but beware of encouraging users to form overly high expectations of the system's ability to understand freeform natural language. Like other end-user programming languages, most notably HyperTalk [App02], TC uses natural keywords and a syntax that mimics natural language phrase structure for many common patterns.

- **Viscosity.** Viscosity measures how much programmer effort is required to make a small change to a program. Visual programming languages typically have high viscosity, since large parts of the program may need to be shuffled around to make room for the change. Old versions of BASIC had high viscosity, since lines may need to be renumbered to make room for a new line of code. Similarly, when indentation is used to represent expression structure, making a small change to an expression may require changing the indentation of a large block of code. In order to make TC less viscous in this respect, the TC editing environment in LAPIS automatically adjusts the indentation of subexpressions as the user types.

- **Use signaling to highlight important information.** Secondary signaling, such as syntax coloring or indentation, give perceptual cues to the meaning of an expression. The TC editor

in LAPIS follows this principle, not only using indentation but also drawing lines showing a pattern's expression structure.

TC is not a general-purpose programming language, so its design does not raise as many challenges that the design of a general-purpose programming language would. Yet there are particular syntactic issues, known to be troublesome from general-purpose programming language design, that TC must face squarely:

- **Boolean operators.** Observations of novice programmers and database users have shown that users tend to confuse *and* with *or*, probably because these words are used ambiguously in natural language [GDCG90, PM00]. The more troublesome of the two operators seems to be *and*, since it can be used in natural language to mean not only Boolean conjunction ("I want candy bars that are chewy and sweet") but also disjunction ("I want candy bars from Hershey and Cadbury"). One study of SQL expressions written by novice users found that it is more common to use *and* incorrectly in place of *or* than vice versa [GDCG90]. TC incorporates these observations into its design by omitting the *and* keyword entirely. Boolean conjunction is represented implicitly by predicate application. Since expert users expect to be able to use *and* and *or*, however, TC also provides an expert mode in which *and* has its conventional meaning.

- **Operator precedence, associativity, and parentheses.** Any language with infix operators faces the problem of precedence and associativity. For example, C has 17 levels of operator precedence, and some of its operators are left-associative while others are right-associative [KR88]. Operator precedence and associativity are well-known problem areas for both novice and expert programmers. In fact, programming style books often recommend redundant parentheses when the precedence or associativity is not completely obvious [KP99]. Unfortunately, parentheses are not an ideal solution either, because a number of studies have shown that novice users tend to misinterpret or ignore parentheses [GDCG90, PM00]. TC avoids precedence and associativity confusion by establishing a common rule for all operators, and avoids most parenthesis problems by structuring expressions with indentation.

## 6.2 Language Description

This section describes the TC pattern language. An alphabetical reference to all the operators in the language may be found in Appendix B.

### 6.2.1 Basic Lexical Rules

TC is case-insensitive, so keywords and identifiers may be written in uppercase, lowercase, or mixed capitalization without affecting their interpretation. Case-insensitivity also applies to literal patterns and regular expressions by default, so the literal pattern `"apple"` matches not only `apple`, but also `Apple` and `APPLE`. Literals and regular expressions may be made case-sensitive using the `case-sensitive` keyword, described in more detail in Section 6.2.8.

Line breaks are significant in TC. A line break ends one or more subexpressions, and the indentation of the subsequent line indicates which parent expression is resumed. Expression indentation is described in more detail in Section 6.2.16.

Comments are started by # and terminated by a line break. Comments can begin anywhere in a line. For example:

```
# This is a comment
"apple"  # This is also a comment
"#this is not a comment because it's quoted"
```

### 6.2.2 Basic Syntactic Rules

All TC expressions are drawn from the same syntactic class, hereafter called *expr*. In the sections that follow, TC syntax is defined by syntactic productions of the form

```
expr ::= production
```

Taken together, all the productions make up a grammar for TC, although an ambiguous one. The parse of a TC expression is determined unambiguously by a combination of precedence rules, indentation, and explicit parentheses. A full discussion of these rules is postponed until Section 6.2.16, after the TC operators have been presented. For the time being, the examples will use explicit parentheses to clarify how an expression should be parsed. In practice, however, most users would write these examples with indentation instead of explicit parentheses.

Either parentheses or curly braces can be used to bracket an expression explicitly:

```
expr ::= (expr)
expr ::= {expr}
```

TC makes no semantic distinction between the two kinds of delimiters. In other languages, curly braces are used to bracket statements, and parentheses to bracket expressions. Since some TC expressions are naturally regarded as statements (e.g., pattern definitions) and others as expressions (patterns themselves), it seems natural to allow both kinds of delimiters but not enforce any difference between them. Parentheses and curly braces are not completely interchangeable, however: a left parenthesis must be terminated by a right parenthesis, not a right curly brace.

Despite the usability problems of explicit parentheses, they are included in TC for three reasons: (1) to allow TC expressions to be written as arguments on a program's command line, where linebreaks and indentation would be difficult to specify; and (2) to allow long lines to be continued; and (3) to match the expectations of expert users.

### 6.2.3 Pattern Identifiers

Any pattern may be named by an identifier. An identifier is a sequence of any characters except whitespace, quotes, parentheses, curly braces, =, or #, as long as it does not match a TC keyword, literal, or regular expression:

```
id ::= [^ \t\n'"(){}=#]+
```

Note that the space of identifiers permitted by TC is larger than most other languages, which typically allow only sequences of alphanumeric characters or underscores in identifiers. As a result, TC identifiers can include punctuation such as `<`, `>`, `[`, and `]`, a property which is put to good use by the HTML parser described in the next section.

A TC expression may reference a pattern by its identifier:

```
expr ::= id
```

The assignments between identifiers and patterns are stored in a global library. The patterns in this library are not just TC expressions. In general, a pattern is a Java object that implements the `Pattern` interface:

```
interface Pattern {
    RegionSet match (Document doc);
}
```

Given some document, the `match` method returns the set of regions that match the pattern in the document.

All compiled TC expressions implement the `Pattern` interface, but nothing prevents writing an arbitrary Java object that implements `Pattern`. This allows external parsers to be integrated into TC, as described in the next section.

### 6.2.4 Parsers

A parser is a Java class that defines a collection of related patterns for some kind of text structure, like HTML or Java syntax. These patterns are generally *not* TC expressions. Instead, the patterns are small Java objects that not only represent structure found by the parser (such as grammar nonterminals) but also serve as entry points into the parser. This section describes the architecture of parsers in more detail.

A parser must implement the `Parser` interface:

```
interface Parser {
    void bind (PatternLibrary lib);
}
```

A parser's `bind` method is called only once, when the parser is first loaded by LAPIS. The `bind` method takes the global pattern library as its only argument, and assigns each of the parser's pattern objects to the appropriate name in the library. The `bind` method essentially performs dynamic linking that makes the parser's identifiers available for use in TC expressions.

Notice that the `Parser` interface has no method for causing a parser to scan a document. Instead, parsing is triggered by calling the `match` method of one of the parser's pattern objects. The parsers currently in LAPIS use a call to `match` as an opportunity to find the matches to *all* of the parser's patterns in a single pass over the document. The region sets found by this parsing pass are then stored in a cache, so that a future application of any of the parser's patterns to the same document can simply look up the answer in the cache.

This design mediates between the traditional approach to parsing, which uses a left-to-right scan through a document, and the region-set approach, which uses the region algebra to manipulate the entire document at once. Parsers generated by parser-generators like lex, yacc, and JavaCC are easy to integrate into this system, and so are hand-coded scanners. At the same time, the pattern library is kept simple. An identifier is always bound to a single pattern, and a pattern applied to a document always produces a single region set. The TC evaluator, and the user, need not be aware of how this region set was produced: whether by a TC expression using region algebra operations or by a traditional parser using a left-to-right scan.

LAPIS includes three built-in parsers. A complete list of the identifiers bound by each parser can be found in Appendix A, but here is a brief overview:

- The HTML parser is a hand-coded finite state machine, which makes it extremely tolerant of illegal HTML. The HTML parser binds several kinds of identifiers into the pattern library. *Tags* are represented by suggestive identifiers like `<p>`, `</p>`, `<a>`, `</a>`, and `<img>`. (Note that there is nothing special about the use of punctuation like <, /, and > in these identifiers. The punctuation marks are not TC operators; they are simply part of the identifier.) *Elements* match complete HTML elements from start tag to end tag (if any). The identifier for an element is the tag name in square brackets, such as `[p]`, `[a]`, and `[img]`. Finally, *attributes* match a name-value attribute in a tag. Examples include `href-attr` or `width-attr`.

- The Java parser is generated from a grammar by the JavaCC parser-generator [Jav00]. The grammar is based on the Java grammar included with JavaCC. The original grammar produced an abstract syntax tree from Java source. In order to make the generated parser into a LAPIS parser, the syntax tree node classes are augmented with region information (start and end offset), and whenever a node of a certain type (such as `Statement` or `Expression`) is reduced by the parser, the node's region is added to the region set for that type. Adapting the JavaCC grammar to LAPIS required about 100 lines of code, most of which is generic code that would adapt any JavaCC-generated parser. No changes to the grammar productions were required.

- The Character parser is a hand-coded lexical analyzer that detects runs of character classes, such as `Letters`, `Digits`, and `Whitespace`. These patterns could be represented by independent regular expressions instead, but the Character parser is more efficient because it makes only a single pass over the document.

### 6.2.5 Pattern Definitions

A TC expression may be assigned to a pattern identifier by the `is` operator:

*expr* ::= *id* **is** *expr*

This expression assigns *expr* to *identifier* in the global pattern library. Like all TC expressions, an assignment expression also has a value — it returns the region set matching *expr* in the current document. This behavior is convenient when defining patterns interactively, because it causes LAPIS to highlight the region set matched by the pattern definition.

If another pattern is already assigned to the identifier in the library, then the new assignment replaces it. Before the new assignment is made, however, all identifiers in the new definition are first *bound*, i.e., replaced with their current assignments in the library. For example, consider the following definitions:

```
Fruit is "apple" or "orange"
Fruit is Fruit or "banana" or "pear"
```

When the first definition is evaluated, the pattern `"apple" or "orange"` is assigned to the identifier `Fruit`. The second definition redefines `Fruit`, but first binds the original definition for `Fruit` so that `Fruit` effectively represents the pattern `("apple" or "orange") or "banana" or "pear"`.

Binding identifiers at definition time allows library patterns to be constrained or generalized without access to their original definitions and without danger of infinite regress. However, early binding has a tradeoff — any definitions added to the library between the original definition of `Fruit` and its redefinition continue to use the original definition of `Fruit`. For example, suppose another definition is evaluated between the two `Fruit` definitions:

```
Fruit is "apple" or "orange"
Produce is Vegetable or Fruit
Fruit is Fruit or "banana" or "pear"
```

Since the reference to `Fruit` in `Produce` is bound at definition time, it continues to be bound to just `"apple" or "orange"` even after `Fruit` itself is extended to include other possibilities. Thus, with early binding, the meaning of a definition is dependent on when it was added to the library.

An alternative strategy is late binding: the definition for an identifier is not resolved until the expression is actually evaluated. With late binding, `Produce` would always use the latest definition of `Fruit`. However, late binding can fall victim to infinite regress when implemented naively. For example:

```
Fruit is Fruit or "banana" or "pear"
```

With naive late binding, this expression evaluates to

```
(Fruit or "banana" or "pear") or "banana" or "pear"
```

and so on. Late binding needs a way to prevent or break the recursion.

Probably the simplest solution is to forbid recursive definitions. Many macro languages adopt this strategy, including the C preprocessor (which has the rule that a macro can be expanded at most once in any given macro expansion). The no-recursion requirement could be checked at definition time by searching for cycles in the dependency graph, so that the system can raise an error as soon as a definition creates a cycle. A disadvantage of forbidding recursive definitions is that users would be unable to redefine an identifier like `Fruit` in terms of itself. Every change to `Fruit` would require the user to edit the original definition, which might be complex and intimidating.

In order to allow revisions of definitions, one could permit self-referential expressions (using early binding for the self-reference, but late binding for all other references) and forbid mutually recursive definitions. However, it is often useful to write mutually constraining definitions:

```
Paragraph is [p] containing Sentence
Sentence is (from CapitalizedWord to ".") in Paragraph
```

These definitions indicate that a Paragraph must contain a Sentence and a Sentence must lie in a Paragraph. With early binding, the first definition would generate an error, because `Sentence` is not yet bound. With late binding, the definitions would be accepted, but evaluating them would cause an infinite regress. In this particular case, the mutual recursion can be removed with an auxiliary definition that serves as a base case:

```
Paragraph0 is [p]
Sentence is (from CapitalizedWord to ".") in Paragraph0
Paragraph is Paragraph0 containing Sentence
```

LAPIS uses this technique to define `Paragraph` and `Sentence` in its built-in library. Obviously, however, this solution can only represent a bounded depth of recursion.

Another tradeoff between early binding and late binding is related to the LAPIS implementation. When a pattern like `Produce` is matched against a document, the resulting region set is stored in a cache associated with the document, so that future searches for `Produce` can simply look up the answer. The cache entry is invalidated when the document changes or when the definition of `Produce` changes. With late binding, however, the meaning of `Produce` may change whenever a change is made to any of the definitions it depends on, such as `Fruit`. When `Fruit` is redefined under a late binding regime, it must invalidate the cache entries for all the patterns that depend on it.

To summarize the tradeoffs, early binding makes definitions order-dependent, and late binding must check for dependency cycles and invalidate cache entries of dependent definitions. The LAPIS implementation currently uses early binding, which is easy to implement. In hindsight, for the reasons given above, a combination of early binding (for self-referential definitions like `Fruit`) and late binding (for other references) might have been preferable.

### 6.2.6 Visible and Hidden Identifiers

Many of the patterns in the LAPIS library are defined by TC expressions. When defining patterns for reuse in a library, however, it is important to be able to control the public interface exposed by your patterns — which identifiers are accessible to clients of the library, and which should be considered internal to the implementation. TC supports this distinction with a typographic convention: any identifier starting with an at-sign (@) is a *hidden identifier*.

For example, the definitions for `Paragraph` and `Sentence` given in the previous section would be improved if `Paragraph0` were a hidden identifier:

```
@Paragraph is [p]
Sentence is (from CapitalizedWord to ".") in @Paragraph
Paragraph is @Paragraph containing Sentence
```

These definitions produce only two public, visible identifiers, `Paragraph` and `Sentence`, but `@Paragraph` is kept hidden from the user, so that it can be renamed or removed as the definitions of `Paragraph` and `Sentence` evolve.

Hidden identifiers are treated differently from visible identifiers in several ways:

- Hidden identifiers do not appear in the LAPIS pattern library browser (Figure 6.1). A mode that shows all identifiers, both visible and hidden, would be useful for debugging purposes, but LAPIS does not currently provide it.

- Hidden identifiers are not automatically imported from other namespaces (see the next section).

- Hidden identifiers are not used for inference (Chapter 9) or outlier finding (Chapter 10).

Hidden identifiers are not protected, however. If a user knows that `@Paragraph` is a hidden identifier defined by the `Paragraph`/`Sentence` definitions, then the user can write a pattern referring to `@Paragraph` just like a visible identifier. Hidden identifiers do not enforce information hiding; they merely encourage it. However, only an advanced user of TC — in particular, one who wants to write a modular system of patterns to be installed in the library — should need to know about or use hidden identifiers.

Hidden identifiers can also be overwritten by new definitions, just like visible identifiers. A potential disadvantage of hidden identifiers — collisions between hidden names — is alleviated by the namespace mechanism described in the next section. Hidden identifiers are generally used only inside a namespace.

### 6.2.7 Namespaces

Every identifier in TC belongs to some *namespace*. Namespaces are hierarchical pathnames, with the components separated by dots. The root namespace is denoted by a single dot, "." For example, `.Business.Address.ZipCode` refers to the identifier `ZipCode` in the namespace `.Business.Address`. An identifier may simultaneously denote a pattern and a namespace; for example, `.Characters.Whitespace` is an identifier denoting runs of whitespace, but it is also the namespace for `.Characters.Whitespace.Spaces` and `.Characters.Whitespace.Tabs`, which denote runs of particular kinds of whitespace.

Namespaces have two purposes in TC. First, they make the pattern library easier to browse by subdividing it into hierarchical categories. Figure 6.1 shows a screenshot of the LAPIS pattern library, which organizes the identifiers in the library by namespace. Second, namespaces reduce the risk of collisions between identifiers and permit modularity in parsers and definitions. For example, a Java parser and a C++ parser can both use identifiers like `Statement`, `Type`, and `Expression` as long as those identifiers are placed in different namespaces.

Many languages use some kind of namespace or package mechanism, including C++, Tcl, Perl, and Java. In all these languages, the user must explicitly import other namespaces in order to access their identifiers. TC takes a different approach. In TC, *all* namespaces are imported automatically. Lookup rules give preference to identifiers in the current namespace, and ambiguous identifiers must be disambiguated by an explicit namespace prefix. These rules are described below.

Every TC expression is evaluated relative to some namespace, called the *current namespace*. By default, the current namespace is the root namespace (dot). The current namespace may be changed by the `prefix` operator:

```
expr ::= prefix identifier expr
```

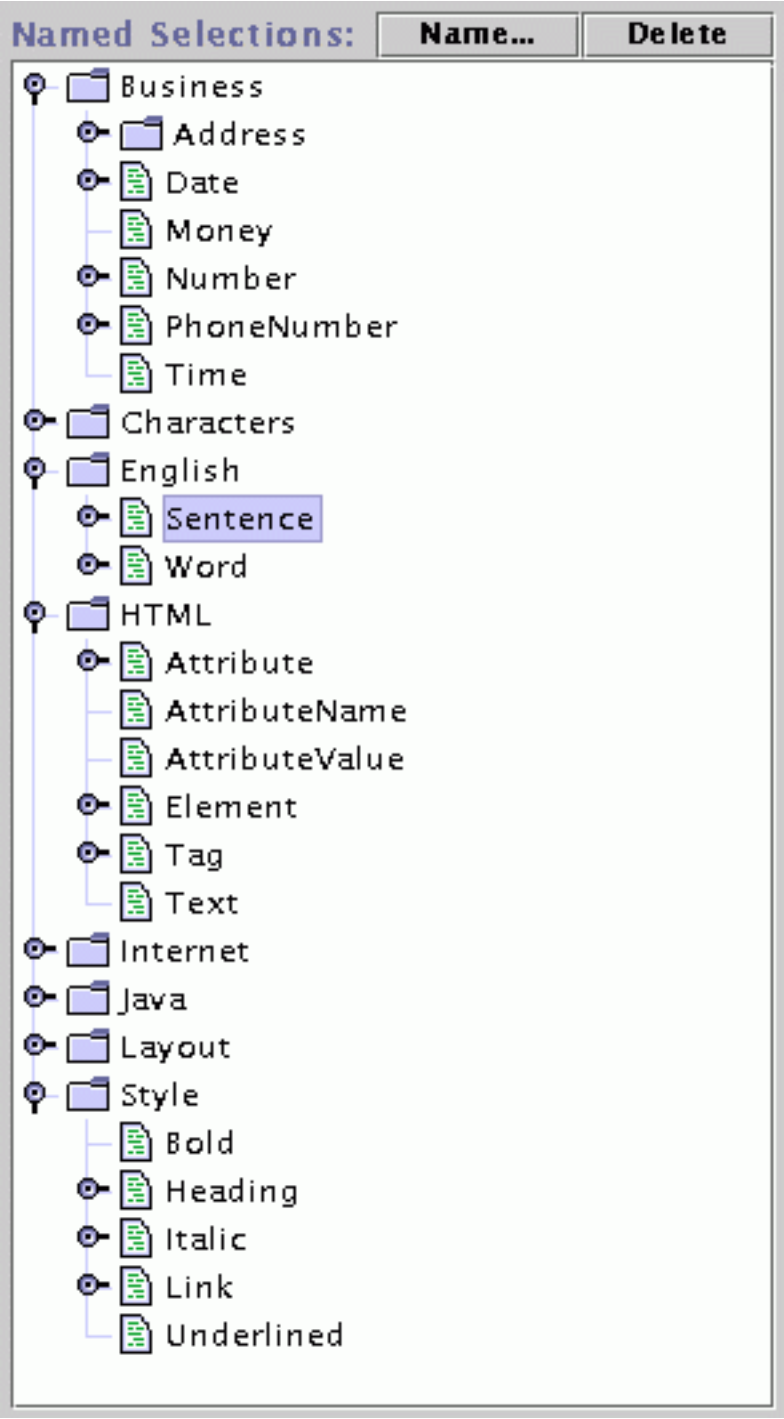



Figure 6.1: The LAPIS pattern library shows the namespace hierarchy.

This expression changes the current namespace to *identifier* for the scope of *expr*. For example,

```
# current namespace is .
prefix Grocery {
   # current namespace is .Grocery
   prefix Produce {
       # current namespace is .Grocery.Produce
   }
}
```

Any identifier, whether denoting a namespace or a pattern, is either absolute or relative. An absolute identifier starts with a period, e.g., `.Grocery.Produce.Fruit`. An absolute identifier describes a complete path from the root namespace. A relative identifier lacks a starting period; examples include `Grocery.Produce.Fruit`, `Produce.Fruit`, or just `Fruit`. A relative identifier must be converted into an absolute identifier by interpreting it relative to the current namespace. However, the interpretation of relative identifiers depends on whether the identifier is being *defined* (by a `prefix` or `is` expression) or *used* (referenced in a pattern).

When a relative identifier is used in a definition, the relative identifier is simply appended to the current namespace. For example, all of the following definitions define the identifier `.Grocery.Produce.Fruit`:

```
prefix Grocery {
   prefix Produce {
       Fruit is "apple" or "orange"
   }
}

prefix Grocery.Produce {
    Fruit is "apple" or "orange"
}

Grocery.Produce.Fruit is "apple" or "orange"

prefix SomeOtherNamespace {
    prefix .Grocery.Produce {
        Fruit is "apple" or "orange"
    }
}
```

The last example in particular shows that namespaces are not protected from each other. Any namespace can define identifiers in any other namespace using an absolute identifier.

When a relative identifier is referenced in a pattern, TC searches for the identifier in the pattern library using the following procedure. Assume the current namespace is *namespace* (an absolute identifier) and the referenced identifier is *identifier* (either relative or absolute).

1. If *identifier* is absolute, then look up *identifier* in the library. If *identifier* is found, return it. Otherwise report that *identifier* is not found.

2. Look up *namespace.identifier*. If it is found, return it; otherwise, continue searching.

3. Look up all identifiers named *.*identifier*, where * is a namespace path of any length starting from the root, possibly empty. If there is exactly one match, return it. If there is more than one match, report an error to the user listing all the identifiers matching *.*identifier* and suggesting an unambiguous identifier for each one.

4. Report that *identifier* is not found.

The name resolution algorithm is designed so that most users can ignore the existence of namespaces entirely. If an identifier occurs uniquely (in only one namespace in the library), then the identifier can always be used directly, with no need for a namespace qualifier or explicit import. All the identifiers in the built-in LAPIS library are unique, so the user can refer to `PhoneNumber` or `URL` or `Whitespace` without knowing or caring which namespaces these identifiers are found in. If the user defines a pattern with the same name as an existing identifier, then the user's pattern takes precedence, since the user's definition is stored in the current namespace (the root namespace, by default) and the current namespace always takes precedence over other namespaces. Thus a user can write

```
Statement is Sentence ending "."
```

and proceed to use `Statement` to refer to this definition, blissfully unaware that the Java parser also defines an identifier `Statement` with completely different semantics in the `Java` namespace. The shadowed meaning of `Statement` is still accessible with `Java.Statement`, and any expressions evaluated in the `Java` namespace continue to use the Java version of `Statement`.

When two namespaces define the same identifier, and neither one is the current namespace, the user must add a namespace qualifier to disambiguate them. As an interesting side-effect of step 3 in the name resolution algorithm, however, the qualified name need not be absolute. It is sufficient to use *any suffix* of the namespace path that uniquely describes the desired identifier. For example, suppose the library contains two identifiers named `State`: `.Business.Address.State`, and `.Programming.Verilog.State`. Then the former identifier can be written uniquely as `Address.State`, and the latter as `Verilog.State`. As a result, a qualified name need not be longer than necessary.

The main drawback of TC's namespace mechanism is that adding new identifiers to the library may cause existing patterns to break. For example, suppose a C++ parser is added to the library, so that the identifiers `Statement`, `Expression`, and `Type` occur not only in the `Java` namespace but also in the `C++` namespace. Then existing patterns that refer to `Statement` will subsequently fail with an error message, because `Statement` is ambiguous without a qualifier. Namespaces in other languages also suffer from this problem, but to a lesser degree. For example, a Java program that imports all the classes in the `java.io` package may become uncompilable if a class with a conflicting name is added to the `java.io` package, but not if a class is added to some `foo.bar.baz` package that the program never refers to.

TC patterns can be made robust to this kind of change by using absolute identifiers for all references outside the current namespace. To make this more convenient, hidden local identifiers can be defined as aliases for the foreign identifiers. For example:

```
prefix MyNamespace {
    @stmt is .Java.Statement
    @word is .English.Word
    IfStatement is @stmt starting @word = "if"
}
```

The hidden identifiers `@stmt` and `@word` have the effect of explicitly importing the foreign identifiers `.Java.Statement` and `.English.Word` into the `MyNamespace` namespace. This technique is essentially identical to the explicit-import-with-aliasing feature found in Modula-3 and Python. The only missing piece is an automatic check that *all* foreign references are absolute, in order to avoid accidental foreign references due to misspellings or omitted definitions. This check could be made by simply disabling step 3 in the name resolution algorithm when the user requests it with a special directive. TC does not yet provide this directive, however.

### 6.2.8 Literals

A string in quotes returns the set of all regions that literally match the string:

```
expr ::= "literal"
expr ::= 'literal'
```

Either single quotes or double quotes may be used, with the same effect.

A literal pattern may return an overlapping region set. For example, when the pattern "aa" is matched against the string `aaaa`, the resulting region set contains three regions: `[aa]aa`, `a[aa]a`, and `aa[aa]`. A literal pattern can only return an overlapping region set if some strict prefix of the pattern is also a suffix of the pattern.

By default, literal matching ignores alphabetic case. Literal matching can be made case-sensitive using the `case-sensitive` operator:

```
expr ::= case-sensitive expr
```

This operator has the effect of making matching case-sensitive for the entire scope of *expr*, which may be a single literal:

```
case-sensitive "apple"
```

or may be an entire system of definitions:

```
case-sensitive {
    keyword is "while" or "do" or "if" or "for" ...
}
```

The complementary operator `not case-sensitive` can be used to revert to case-insensitive matching within the scope of a `case-sensitive` operator:

```
expr ::= not case-sensitive expr
```

No metacharacters or escape codes are recognized within the literal string. In particular, the backslash character just means backslash. Metacharacters are troublesome because they impose a memory load on users, and using backslash as an escape code causes trouble on platforms that use backslash for other purposes, such as separating components of a pathname. TC provides other ways to match characters that require escape codes in other systems:

- Single quotes or double quotes can be matched by using the other quote character to delimit the string. For example, the pattern `"Bob's your uncle"` matches a single quote by surrounding it with double quotes, and `'He said "hello" to me'` does the reverse. If a literal match contains both single quotes and double quotes, then it can be split up into multiple concatenated patterns using `then`:

  ```
  'He said "' then "Bob's your uncle" then '" to me'
  ```

- The library patterns Linebreak and Tab match \n and \t.

Advanced users who want to use escape codes to match nonprintable characters can fall back to regular expressions, described below, in which all the usual escape codes are available.

### 6.2.9 Inferring Quotes

When TC is used for interactive pattern matching, quoting literal patterns can be a hassle. Observations from the TC user study bear this out: novice TC users frequently forget to put quotes around literal patterns. Quoting has also been identified as a key usability problem by other researchers [Bru97]. It seems likely that novice users are tempted to omit quotes by prior experience with Find commands in other text editors, since these commands generally accept *only* a literal pattern, with no need for quoting.

TC addresses the problem of omitted quotes by automatically inferring quotes around single tokens when the token is neither a keyword nor a defined identifier. Quote inference is implemented by adding a step to the name resolution algorithm from Section 6.2.7:

1. If *identifier* is absolute, then look up *identifier* in the library. If *identifier* is found, return it. Otherwise report that *identifier* is not found.

2. Look up *namespace.identifier*. If it is found, return it; otherwise, continue searching.

3. Look up all identifiers named **.identifier* (where * is a namespace path of any length). If there is exactly one match, return it. If there is more than one match, report an error to the user listing all the identifiers matching **.identifier* and suggesting an unambiguous identifier for each one.

4. **Search for *identifier* as a literal pattern in the current document. If at least one match is found, then treat this pattern as the literal pattern "*identifier*".**

5. Otherwise report that *identifier* is not found.

Since TC identifiers are drawn from a very large lexical class — only whitespace, quotes, parentheses, curly braces, # and = cannot appear in an identifier — this technique can automatically quote many common kinds of constants, such as:

```
55.3
$200
247-3940
5/18/02
www.cmu.edu
http://www.yahoo.com/
```

Automatic quote inference has a number of tradeoffs. First, the technique only applies to identifier tokens. TC cannot automatically quote multiword phrases, nor can it quote a word that happens to be a keyword, like `in`, `or`, or `before`. Although it is tempting to try automatic multiword quoting whenever a syntax error occurs, the number of ways to "correct" an expression by adding quotes explodes exponentially with the number of tokens in the expression. If a multiword expression contains no reserved words or special characters, however, it may be reasonable to quote the entire expression automatically. For example, if the user wrote

```
Gettysburg Address
```

then the system would interpret it as the quoted literal "Gettysburg Address". This approach is used by Bruckman's MOOSE system [Bru97]. One problem with automatic multiword quoting, however, is that system error messages become more complex when neither the original pattern nor the quoted pattern succeeds. The gist of the error message would be:

```
No matches found, either because Gettysburg Ad-
dress isn't a properly formed pattern or because "Gettys-
burg Address" isn't found in the document.
```

Automatic quoting of a single-word pattern like `Gettysburg` doesn't face this problem, because an error message like

```
No matches found, because Gettys-
burg isn't found in the library or the document.
```

adequately describes the situation. For this reason, TC does not currently do automatic multiword quoting.

Another problem with automatic single-word quoting is that a user accustomed to omitting quotes from literals may write a quoteless pattern that accidentally refers to a identifier defined in the library. For example, suppose the user executes

```
line containing state
```

intending it to be interpreted as `line containing "state"`, but the `state` identifier actually resolves to the pattern `.Business.Address.State` matching states of the US. This can lead to a disturbing situation where the user can *see* regions that match the intended pattern, but the actual pattern is matching something different, with no error message to suggest what might have gone wrong.

Similarly, automatic single-word quoting can make references to undefined identifiers appear legal when the identifier happens to occur as a literal in the document. This is less of a problem for interactive patterns than it is for pattern definitions written for reuse in the pattern library. A future version of TC will have a directive that disables automatic quoting, so that expert users can write patterns for reuse that are checked so that they only use explicit quoting.

### 6.2.10 Regular Expressions

A regular expression surrounded by slashes returns the set of regions matching the regular expression:

```
expr ::= /regexp/
```

For example,

```
/[0-9]+-[0-9]+-[0-9]+/
```

matches three groups of digits separated by dashes, such as 333-555-12203. Like literal patterns, regular expression patterns are case-insensitive by default. A regular expression may be made case-sensitive using the `case-sensitive` operator.

LAPIS uses an off-the-shelf regular expression library to match regular expressions, so the syntax and semantics of a regular expression depends how the library interprets it. LAPIS currently uses the Jakarta-regexp library for Java [Jak99]. The operators recognized by this library are shown in Figure 6.2.

The regular expression library does not return *all* regions that match a regular expression. It returns a *nonoverlapping* region set of matches found in a left-to-right scan of the string. For example, when the expression `/aa/` is matched against the string `aaaa`, the resulting region set includes only the two regions `[aa]aa` and `aa[aa]`. Furthermore, when the regular expression contains a greedy repetition operator *, +, or ?, the expression matches as many occurrences of the repetition as possible, so `/a.*a/` matches *all* of the string `abbaaba`. Conversely, the nongreedy repetition operators *?, +?, and ?? match as few occurrences as possible. Matching `/a.*?a/` against the string `abbaaba` returns two regions, `[abba]aba` and `abba[aba]`.

This behavior is inherent in the regular expression library, and conforms to the expectations of users familiar with regular expressions in other systems. One unfortunate consequence, however, is that the literal pattern `"aa"` is not equivalent to the regular expression `/aa/`, since the former can return an overlapping region set but the latter cannot. One might fix this particular inconsistency by restarting the regular expression matcher after every match — i.e., after the matcher returns a region $[s, e]$, restart it from position $s + 1$. This change would also affect how other regular expressions were interpreted: for example, `/a*/` matched against `aaaa` would produce four regions, `[aaaa]`, `a[aaa]`, `aa[aa]`, `aaa[a]`, and `aaaa[]`, which is different from its interpretation in

| Expression | Matches |
|---|---|
| *character* | *character* itself |
| . | any character |
| [abc] | character class: any of a, b, or c |
| [a-zA-Z] | character range a-z or A-Z |
| [^abc] | any character except a, b, or c |
| E* | zero or more occurrences of E |
| E+ | one or more occurrences of E |
| E? | zero or one occurrences of E |
| (E) | grouping |
| E\|F | either E or F |
| EF | E followed by F |
| *c* | quotes a metacharacter *c* |
| \n, \r, \t | newline, carriage return, tab |
| \0nnn, \xhh | an ASCII character in octal (0nnn) or hexadecimal (xhh) |
| \\ | backslash |
| \1, \2, \3, etc. | the same as the *n*th parenthesized subexpression |
| ^ | the start of a line |
| $ | the end of a line |
| \w, \s, \d | word, space, or digit character |
| \W, \S, \D | non-word, non-space, or non-digit |
| \b, \B | word boundary or non-word boundary |
| [:alpha:], [:alnum:], [:digit:], etc. | predefined character classes |
| E*?, E+?, E?? | nongreedy closures: matches as *few* occurrences of E as possible |
| E{n}, E{n,}, E{n,m} | exactly n, at least n, or between n and m occurrences of E |

Figure 6.2: Regular expression operators supported by LAPIS.

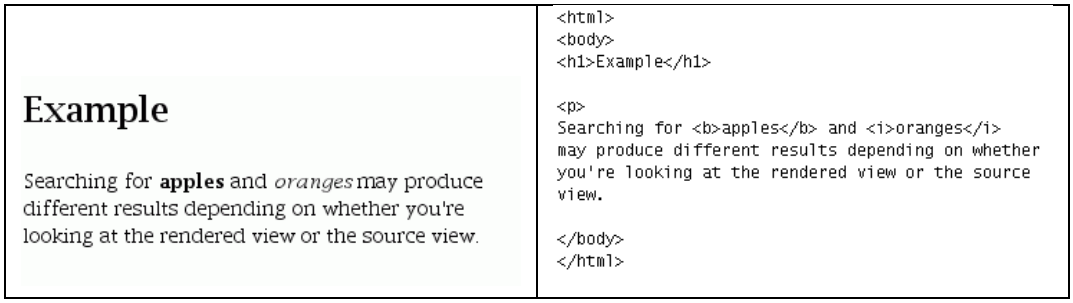



Figure 6.3: Different views of a simple HTML page: (a) rendered; (b) source.

other systems. The dilemma is between internal consistency with TC literal patterns, and external consistency with regular expressions in other systems. Since regular expressions are included in TC largely for the sake of expert users to transfer their experience from systems like Perl and awk, the dilemma is resolved in this case in favor of external consistency.

For future work, it would be interesting to implement a regular expression library that can produce *all* the regions matching a regular expression. Such a library would be more consistent with the behavior of other TC operators, but considerable care would be needed to make it run as efficiently as existing regular expression libraries.

### 6.2.11 Matching Against HTML

LAPIS can display web pages as well as text files. By default, web pages are displayed in a *rendered view* in which the HTML markup is interpreted as formatting instructions (Figure 6.3(a)). A web page may also be viewed in a *source view* in which the HTML markup is visible (Figure 6.3(b)). A literal or regular expression pattern can specify which of these views it should match against.

By default, a literal or regular expression is matched against the current view. When the rendered view is displayed as in Figure 6.3(a), the literal pattern `"apples and oranges"` matches the text shown. When the source view is showing as in Figure 6.3(b), the literal pattern must be `"<b>apples</b> and <i>oranges</i>"` to match the equivalent place in the text.

A TC expression can force matching against a particular view using the `view source` or `view rendered` operators:

```
expr ::= view source expr
expr ::= view rendered expr
```

Any literal or regular expression patterns in the scope of *expr* is matched against the specified view, unless overridden by another `view source` or `view rendered` operator.

The current view has no effect on patterns defined by parsers (Section 6.2.4). A parser may choose to look to scan the source view, the rendered view, or both when it parses a document. For example, the HTML parser parses the source view, while the Character parser parses the rendered view.

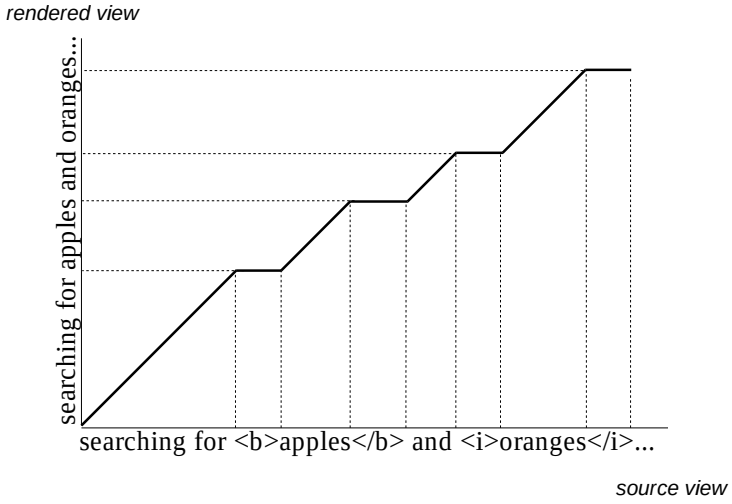



Figure 6.4: Coordinate map mapping from source view to rendered view for part of an HTML document.

### 6.2.12 Coordinate Maps

Each view has a different *coordinate system*, a mapping of integers to character positions. The coordinate systems of the two views are related to each other by a *coordinate map*, a binary relation that equates an offset in one view to an equivalent offset in the other view. A simple coordinate map that maps between source and rendered views is shown in Figure 6.4. A coordinate map is not necessarily a function, but all coordinate maps are monotonically increasing in the sense that if $[a, b]$ and $[c, d]$ are related by the map, then either $a \leq c$ and $b \leq d$ or $a \geq c$ and $b \geq d$. The inverse of a coordinate map from source view to rendered view is the coordinate map from rendered view to source view. Although the domain and range of a coordinate map are technically nonnegative integers, it is convenient to extend the map over the reals (as shown in Figure 6.4) and regard it as a piecewise linear, monotonically increasing curve. This perspective allows a coordinate map to be represented compactly by the endpoints of the linear pieces in sorted order. A coordinate can be transformed through a coordinate map in $O(\log N)$ time by finding the endpoints above and below it with binary search and then interpolating.

Coordinate maps are used to transform pattern matches found in one view into the other view. By convention, all region sets returned by TC expressions are relative to the source-view coordinate system, so that TC operators that combine region sets need not do any coordinate transformations. When a literal or regular expression pattern matches against the rendered view, it transforms the resulting region set into source-view coordinates before returning it.

Coordinate maps are also used to highlight the result of a pattern match in the rendered view. Since the pattern matcher returns a region set in source-view coordinates, LAPIS must transform the region set to rendered-view coordinates before it can highlight the regions.

Coordinate maps are also used to translate region sets between different versions of a document. This technique is used by simultaneous editing to update cached region sets as the user edits the document (Chapter 9).

| TC Syntax | Algebra Operator | Section |
|---|---|---|
| *expr*? **in** *expr* | *in* | 3.5.2 |
| *expr*? **contains** *expr* | *contains* | 3.5.2 |
| *expr*? **equals** *expr* | *equals* $_W$ | 3.6.13 |
| *expr*? **just before** *expr* | *just-before* $_W$ | 3.6.13 |
| *expr*? **just after** *expr* | *just-after* $_W$ | 3.6.13 |
| *expr*? **starts** *expr* | *starts* $_W$ | 3.6.13 |
| *expr*? **ends** *expr* | *ends* $_W$ | 3.6.13 |
| *expr*? **anywhere before** *expr* | *before* | 3.5.2 |
| *expr*? **anywhere after** *expr* | *after* | 3.5.2 |
| *expr*? **overlaps** *expr* | *overlaps* | 3.6.3 |
| *expr*? **overlaps start of** *expr* | *overlaps-start* | 3.5.2 |
| *expr*? **overlaps end of** *expr* | *overlaps-end* | 3.5.2 |
| *expr* **then** *expr* | *then* $_W$ | 3.6.13 |
| **from** *expr* **to** *expr* | *fromto* | 3.6.8 |
| **balanced from** *expr* **to** *expr* | *balances* | 3.6.8 |
| **start of** *expr* | *start-of* | 3.6.1 |
| **end of** *expr* | *end-of* | 3.6.1 |
| *expr* **trim** *expr* | *trim* | 3.6.14 |
| *n***th** *expr* | *nth* | 3.6.5 |
| *n***th** *expr* (**before**\|**after**\|**in**) *expr* | *nth* $-$ *op* | 3.6.5 |
| **flatten** *expr* | *flatten* | 3.6.12 |
| **melt** *expr* | *melt* | 3.6.12 |
| **nonzero** *expr* | *nonzero* | 3.6.4 |

Figure 6.5: TC pattern-matching operators with their equivalent region algebra operators from Chapter 3. Several of the relational operators can be either unary or binary, as indicated by the optional first operand *expr*?.

### 6.2.13 Pattern Matching Operators

At the heart of the TC language lie the region algebra operators (Chapter 3). TC supports the six fundamental relations of the region algebra (*in, contains, before, after, overlaps-start,* and *overlaps-end*), although several of them are renamed for usability reasons explained below. In addition to these fundamental operators, TC provides most of the derived operators whose semantics were defined in Chapter 3. The complete set of pattern-matching operators is shown in Table 6.5.

The names of the operators were chosen with some care. Some of these design decisions were made by following the usability principles listed in Section 6.1. Others resulted from observing the errors made by users in the TC user study, described in Section 7.6. Some of the more interesting design decisions are discussed below.

- **One official keyword, several unofficial synonyms.** Table 6.5 lists only one keyword for each operator, but TC actually accepts a number of synonyms, shown in Table 6.6. For example, *starts* can also be written as *starting*, *starts with*, *starting with*, *beginning with*, or *at start of*. All patterns presented by the system — in tutorial materials, online help, or self-disclosure — only use the official keyword, to avoid giving the impression that *starts* and

| Operator | Synonyms |
|---|---|
| `in` | inside, of |
| `contains` | containing, containg *[sic]* |
| `equals` | equal to, equalling, equaling, = |
| `just before` | directly before, right before, jbef |
| `just after` | directly after, right after, jaft |
| `anywhere before` | abef |
| `anywhere after` | aaft |
| `starts` | starts with, starting, starting with, at start of, begins, begins with, beginning, beginning with, at beginning of |
| `ends` | ends with, ending, ending with, at end of, finishes, finishing, finishes with, finishing with |
| *n*`th` | *n*st, *n*nd, *n*rd, *n*d, first, second, third, ..., tenth |
| `trim` | trimming, trimming off |
| `not` | but not |

Figure 6.6: Synonyms for TC operators.

*begins* might mean different things (following the rule Avoid Misleading Appearances). But by accepting synonyms as well, the system becomes more tolerant of a user's faulty memory. Synonyms are also useful as abbreviations of commonly-used but verbose keywords. For example, an experienced TC user can just use *jbef* instead of writing out *just before.* The tradeoffs of accepting synonyms are twofold. First, synonyms consume more reserved words, reducing the space of identifiers available to the programmer. Second, synonyms must be chosen judiciously. A word or phrase that might be a synonym for more than one operator should not be accepted for any of them. For example, *followed by* is a possible synonym for *just before*, but it might also mean *anywhere before*, *then*, or *from-to*, so TC doesn't accept this synonym for any of them.

- **Containing vs. contains**. Originally, the official keyword for the *contains* operator was *containing*. This decision was justified by the fact that it makes TC expressions read more like grammatically correct English, which, it was hoped, would make the expressions both easier to generate and easier to comprehend. Finding a "line containing apple" sounds better than finding a "line contains apple". However, users tended to misspell *containing* as *containg* — a natural mistake, since the keyword is quite long and repeats *in* twice. After several users made this mistake, the official keyword was changed to *contains*, which is shorter and less apt to be misspelled, and TC was modified to accept both *containing* and *containg* as synonyms for *contains*. Subsequent users made no spelling mistakes with *contains*. Accepting common misspellings of keywords is not a new idea, of course. For example, AMPL [FGK93] accepts *sovle* as a synonym of *solve*.

- **Concatenation.** In other pattern languages, notably regular expressions and grammars, concatenation is expressed implicitly by juxtaposing expressions, as in *expr1 expr2 expr3*. This approach was considered but discarded in favor of an explicit concatenation keyword, on the grounds that an explicit keyword would contribute more to TC's readability. The choice

of keyword, *then*, was motivated by a study [PRM01] which found that nonprogrammers describing a programming task in natural language often used *then* to describe sequencing.

- **Starts vs. at start of/starting with.** An early version of TC provided two operators for the same-start relation: *at start of* and *starts with*. The *at start of* operator was equivalent to the algebra operator *starts-in*, that is, it matched regions that both started at the same place and lay completely inside another region. The *starts with* operator, on the other hand, was equivalent to the algebra operator *starts-contains*, which matched regions that started at the same place and completely contained another region. This choice of operators was later reconsidered in light of usability guidelines (Avoid Subtle Distinctions in Syntax), since it seemed likely that users would confuse *at start of* and *starts with*. Fortunately, for many patterns, the direction of containment is implicit. For example, in the pattern *Word starting with "s"*, it is unnecessary to have the extra constraint that the Word must completely contain "s", since that is implicit in the definition of Word. For common uses, then, the two operators can be replaced by a single operator *starts* that requires only that two regions have the same start point, saying nothing about which region contains the other. The user now has only one operator to remember, and no subtle distinctions between keywords: *at start of* and *starts with* are now synonyms for the *starts* operator. In the occasional pattern where the containment relation must be specified, it can be made explicit by combining *starts* with *in* or *contains*. The *ends* operator followed a parallel evolution.

- **Before vs. anywhere before**. Although *before* is one of the six fundamental region relations, it is not as useful in practice as the more specialized operators that are derived from it, particularly *just before*. *Before* is a much weaker constraint than *just-before*. An expression like *before R* effectively splits the string into two parts, and only the last region in *R* plays any role in this split — most of the structure described by *R* is completely irrelevant. Thus *before* is useful only for splitting the entire string around a single landmark, which is not often done. The expression *just-before R*, on the other hand, is a constraint that depends on the position of every region in *R*, and it may "split" the string into as many pieces as there are regions in *R*.

  Since *just before* is far more useful in practice than *before*, there is no particular need for the *before* operator to have such a short name, especially since it is prone to confusion with *just before* (Avoid Misleading Appearances). As a result, the operator was renamed to *anywhere before*, which clearly disambiguates it from *just before*. If the user writes a pattern simply using *before*, the system generates an error message asking the user to choose between *just before* and *anywhere before*. Similarly, the *after* operator is renamed to *anywhere after*.

### 6.2.14 Ignoring Background

Several relational operators test whether two regions have coincident endpoints: *just before*, *just after*, *starting*, *ending*, *equals*, and *then*. As discussed in Section 3.6.13, it is useful to weaken these tests so that endpoints are considered equivalent as long as they are only separated by irrelevant regions, called the background.

All the adjacency-testing operators in TC have an implicit background parameter, specified as a region set. Any characters covered by the background region set are considered irrelevant.

| File Type | Default background |
|---|---|
| Plain text | Whitespace, punctuation |
| HTML | Whitespace, punctuation, tags |
| Java | Whitespace, comments |

Figure 6.7: Default background for different file types.

Since the background is specified by a region set, instead of merely a set of character classes like whitespace and punctuation, the background can include rich high-level structure like HTML markup tags or Java comments.

The default background depends on the type of document being matched against. Table 6.7 shows the default background used by LAPIS for the different document types it recognizes. This background can be changed for the scope of an expression using the *ignoring* operator:

```
expr ::= expr ignoring background
```

This operator sets the background to the result of evaluating *background*, for all the adjacency operators in the lexical scope of *expr*. Background can be turned off, so that adjacency tests require strict adjacency, using `expr ignoring nothing`. The current background is available as the pattern identifier `Background`, so that an `ignoring` expression can incrementally add or remove regions from the current background.

In TC, the background is lexically scoped, not dynamically scoped, so changing the background with `ignore` does not affect any patterns that are referenced through pattern identifiers. This is a side-effect of early binding of pattern identifiers, and has the same advantages and disadvantages as early binding (Section 6.2.5).

In other text-processing systems, the effect of background is achieved by tokenizing the text, discarding irrelevant characters like whitespace, punctuation, or comments. Subsequent processing operates only on the resulting stream of tokens. In these systems, the background has already been discarded by the time pattern-matching or other processing occurs, so individual patterns cannot specify which characters should be ignored. In TC, on the other hand, the background is explicit, and it can be varied from pattern to pattern.

### 6.2.15 Boolean Operators

Three TC operators remain to be discussed: the operators for union, intersection, and difference. Previous research has shown these operators to be error-prone (Section 6.1), so some effort was devoted to choosing a syntactic representation that minimize the level of ambiguity or confusion.

Probably the most troublesome word is *and*. Although *and* is the most natural English word for describing Boolean conjunction, appearing as a keyword in nearly every high-level programming language since Fortran, it is also highly ambiguous, having many other meanings in English that novice programmers may inadvertently attribute to the keyword. *And* is not a satisfactory mnemonic because it suggests too many possible meanings. Unfortunately, no simple alternative is any better: words like *with*, *also*, *plus*, *likewise* do not seem to reduce the ambiguity.

TC avoids the ambiguity of *and* for novice users by providing no intersection operator by default. Instead, intersection is represented by relational operators. For example, the algebra

expression

*Word* ∩ *starts* "c" ∩ *ends* "e"

is represented by the TC expression

```
(Word starting "c") ending "e"
```

In general, any algebra expression of the form $A \cap$ *op* $B$ or ( *op* $B) \cap A$ can be represented without intersection by the TC expression `A` *`op`* `B`.

If neither operand of the intersection is a unary relational operator, then we introduce one — in particular, the *equals* operator. For example, the algebra expression

*Word* ∩ "apple"

might be converted first to the algebra expression

*Word* ∩ *equals* "apple"

which then becomes the TC expression

```
Word equals "apple"
```

In general, any expression of the form $A \cap B$ can be represented by the TC expression `A equals B`. One might argue that this means that `equals` is simply taking the place of `and` as the intersection keyword, but this is not entirely accurate. First, `equals` is used only for intersecting "noun-like" expressions, i.e. expressions that are not predicates. Naively replacing every intersection with `equals`, even predicate expressions, would result in unreadable expressions like

```
Word equals (starting "c") equals (ending "e")
```

Second, `equals` provides background-sensitive intersection, which treats two regions as equal as long as their corresponding endpoints differ only by irrelevant background characters. For intersecting noun-like expressions, this is often precisely what is desired. For example, to find lines that contain a URL by itself, one would write

```
Line equals URL
```

which would match correctly even if the `Line` regions included the linebreak and the `URL` regions omitted it, as long as the current background includes whitespace. If the background is set to nothing, then `equals` behaves like ordinary intersection.

Despite the dangers of `and`, several users in the user study noticed its absence and asked for it. To meet these users' expectations, and for consistency with other query languages, TC also provides the `and` keyword:

*`expr`* `::=` *`expr`* **`and`** *`expr`*

By default, however, using `and` in a pattern generates a warning that explains its ambiguity and suggests alternative ways to express each of `and`'s possible meanings. This warning can be turned off by the user, if desired.

Union is specified by the `or` operator:

```
expr ::= expr or expr
```

which can be written more verbosely as follows:

```
expr ::= either expr or expr
```

The `either` keyword is useful for grouping the expression using indentation, explained in more detail in the next section.

Finally, set difference is specified by `not`:

```
expr ::= expr not expr
```

In practice, the second operand of `not` is usually a relational operator:

```
EmailAddress not ending "cmu.edu"
```

but it may also be a noun-like expression:

```
Fruit not "orange"
```

As mentioned in Table 6.6, `but not` may be used as a synonym for `not`.

### 6.2.16 Operator Precedence and Indentation

Having introduced all the operators in TC, it remains to show how operators are grouped into expressions. Expression grouping in TC is determined by three kinds of rules: (1) precedence and associativity, (2) indentation, and (3) explicit parentheses. This section describes these rules and how they interact to determine the parse of a TC expression.

For simplicity, all TC operators share the same level of precedence. Unlike arithmetic operators, whose precedence relationships were well-established long before being encoded in a programming language, the TC operators are generally unencumbered by convention. Also, unlike arithmetic, the "natural" precedence for an expression with natural language keywords seems to depend on a semantic interpretation of the expression. Consider the following expressions with identical operators but significantly different "natural" interpretations:

```
(1) Bag containing Apple or Orange containing Worm
(2) Bag containing Apple or Box containing Orange
```

The first expression might be naturally interpreted as "a bag with either an apple or orange in it that has a worm", while the second expression is more naturally interpreted as "either a bag containing an apple or a bag containing an orange". The first interpretation gives higher precedence to `or`; the second, to `containing`.

Rather than define an arbitrary precedence order which might seem natural in some cases but very unnatural in others, and which in any case would have to be memorized, TC takes the simple approach of no precedence at all.

Since TC has only one precedence level, the parse of an unparenthesized, single-line expression is determined by associativity. All operators in TC are right-associative, so that the expression

```
A before B containing C
```

is equivalent to the fully-parenthesized expression:

```
A before (B containing C)
```

Right associativity is actually a side-effect of the indentation rule, which is explained next.

In order to specify expression grouping without parentheses, a TC expression may be split onto multiple indented lines. Each indented line starts with an operator, and the indentation of this operator determines which subexpression is its left operand. In particular, the operator should be indented just beyond the first token of the expression that becomes its left operand. Continuing the previous example, if we want to write `(A before B) containing C` without using parentheses, the indented expression would be:

```
A before B
  containing C
```

Since `containing` is indented just beyond `A`, it captures all of `(A before B)` as its left operand, so the interpretation is `(A before B) containing C`.

If the second line were indented more deeply, as in:

```
A before B
           containing C
```

then `containing` would capture only `B` as its left subexpression, so the interpretation would be `A before (B containing C)`. Extending this rule to single-line expressions explains why all TC operators are right-associative. An operator on the same line as the previous token captures only the token to its immediate left, so that

```
A before B containing C
```

is interpreted as `A before (B containing C)`.

A more precise definition of TC's indentation structure rules is deferred until the next section, which also presents an algorithm for parsing an indented expression.

Although indentation is used to signify block structure in other languages, like Python and Haskell, TC is unique in using indentation to structure binary infix expressions. The chief advantage of indentation in this context is that it allows a sequence of nested infix operators to be regarded as a sequence of constraints. For example, the indented expression

```
A just before B
  containing C
  equal to D
```

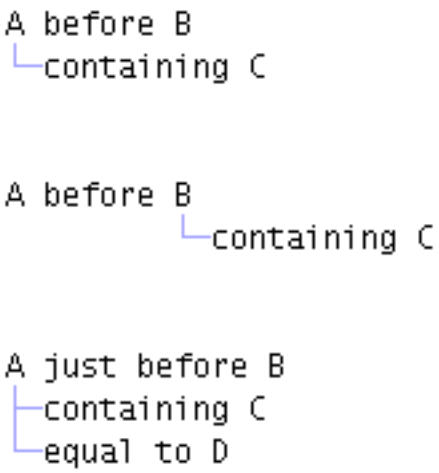



Figure 6.8: The LAPIS pattern editor automatically displays light blue *tie lines* to indicate which token is modified by an indented expression.

can be regarded as a search for regions matching `A` that satisfy three additional constraints: `just before B`, `containing C`, and `equal to D`. The indentation structure shows the parallelism between the three constraints, and highlights the fact that the result of the pattern match will be regions from `A` (not `B`, `C`, or `D`).

LAPIS reinforces the visual meaning of indentation by drawing *tie lines* between an indented operator and the first token of its left operand. Figure 6.8 shows how LAPIS displays the examples shown above. As the user adjusts the indentation of a pattern in the LAPIS pattern editor, the tie lines are automatically redrawn to reflect how the pattern would be interpreted. Tie lines are a form of secondary notation that gives the user more feedback about the meaning of an expression.

Expression indentation has several tradeoffs, however. The first disadvantage is greater *viscosity*, the cost of making a change to an expression. Changing one line of an expression may require adjustments to the indentation of other lines too, in order to keep operators lined up with the appropriate tokens. Fortunately, this problem can be solved by environment support. The pattern editor in LAPIS automatically adjusts the indentation of subsequent lines as the user edits, in such a way that the expression structure is preserved. The pattern editor also has support for indenting and outdenting groups of lines simultaneously, in order to facilitate moving entire subexpressions up and down in the parse tree. These operations are described in more detail in the discussion of the LAPIS user interface (Chapter 7).

Attaching syntactic meaning to indentation also has the effect of making linebreaks syntactically significant, which in turn may lead to undesirably long lines. This problem is more serious in TC than in languages that use indentation only for block structure. In Python and Haskell, which use indentation only for block structure, a new block generally adds only 4 spaces (or at minimum, only 1 space) to the depth of indentation. In TC, however, the depth of indentation can depend on the length of the tokens and operators in the previous line. For example, suppose this expression is too long to fit comfortably on a line:

```
A contains B contains C contains D contains E contains F
```

There is no way to shorten this expression by inserting linebreaks, because the subsequent lines would have to be indented to nearly the same column positions as before:

```
A contains B
            contains C
                      contains D
                                contains E
                                          contains F
```

Other languages in which linebreaks are significant (such as Python and BASIC) deal with this problem with a special character that represents a continued line. In TC, rather than introduce a new special character, long lines can be split by adding explicit parentheses:

```
A contains B contains C contains (
    D contains E contains F
)
```

Parentheses (or curly braces, which are treated the same by TC) effectively block indentation grouping. An operator inside parentheses cannot capture an operand expression outside the parentheses.

Finally, a problem arises when tab characters are used in indentation, since tab widths can vary from editor to editor, changing the indentation and therefore the meaning of the expression. The pattern editor in LAPIS avoids this problem simply by avoiding tabs. Pressing the Tab key inserts only spaces, never tabs. Many other editors can be similarly configured. If an external editor is used to create a TC expression with embedded tabs, LAPIS uses a tab width of 8 to interpret the tabs. This tab width can be changed in the LAPIS preferences dialog. In general, however, the safest bet is to avoid tab characters entirely. A future version of TC may encourage this strategy by giving a warning when embedded tabs are encountered.

### 6.2.17 Parsing Indentation Structure

This section describes how TC expressions are parsed. The overall process has four phases:

1. Tokenization, in which the expression is divided into a sequence of tokens augmented by line and column position information;

2. Structuring, in which indentation and explicit parentheses are interpreted to connect the tokens into a tree;

3. Syntax checking, in which the token tree is checked for violations of syntactic and semantic rules; and

4. Translation, in which the token tree is translated into a syntax tree by a simple transformation.

### Tokenization

Tokenization uses a conventional lexical analyzer to split the expression into tokens. Some operators in TC are represented by a multi-word phrase, such as `just before` or `overlaps start of`; these multi-word operators are represented by a single token after tokenization. However, multi-word operators in which the words are separated by other expressions use a different token for each component. For example, the `from-to` operator uses different tokens for `from` and `to`. All the synonyms for an operator are represented by a canonical token for that operator. Comments and whitespace are removed by the tokenization phase, but each token is marked with its line number and the starting and ending column position of the lexeme that generated it, so that indentation information can be recovered by the next phase.

### Structuring

The structuring phase takes the sequence of tokens and connects it into a *token tree* representing the expression's structure. The token tree differs from a conventional syntax tree in two ways. First, the tree is arranged in *preorder*, so that the first token in an expression is at the root of the tree, rather than *infix order*, in which an operator would be at the root of the tree. Second, every token is a node of the tree, even if it is part of a multiple-token operator like `from-to`, `either-or`, or parentheses. In a conventional syntax tree, a multiple-token operator would occupy only a single node of the tree. An illustration of these differences can be found in Figure 6.9, which compares how arithmetic expressions parsed with conventional operator precedence would be represented as syntax trees and as token trees. Note that a token tree is not necessarily a binary tree, even if all the operators in the tree are binary.

The token tree representation is ideal for indentation parsing because it puts the first token of an expression at the root of its subtree. For any given token $T$, all tokens that appear to the right of $T$ — either on the same line, or on subsequent lines indented deeper than $T$ — become descendants of $T$ in the token tree. Figure 6.10 shows the token tree for several TC expressions, in which the reader can readily verify this property. As a result, the token tree can be constructed by a simple algorithm that uses a stack to keep track of "open" tokens, i.e. tokens which appear above and to the left of the current line and column position. When a new token is returned by the tokenizer, any tokens that start to the right of the new token are popped off the stack. The top of the stack then becomes the new token's parent in the token tree.

Algorithm 6.1 illustrates this procedure. The algorithm has a wrinkle designed for smarter parsing of multi-token operators like `from-to`, `either-or`, and parentheses (lines 6–13). When the second token in a multi-token operator is encountered — for example, the `to` of a `from-to` expression — the algorithm looks down the stack (using the POPTO procedure) to find the corresponding first token, in this case `from`. If the matching `from` token is found, POPTO pops the stack so that `to` can be matched with `from`. This rule allows the algorithm to parse a one-line `from-to` expression, such as:

```
from "Albuquerque" to "Phoenix"
```

Following strict indentation rules, the `to` token would be a child of `"Albuquerque"`, which would be a syntax error. Peeking down the stack to find the matching `from` token allows the system to interpret the expression more sensibly.

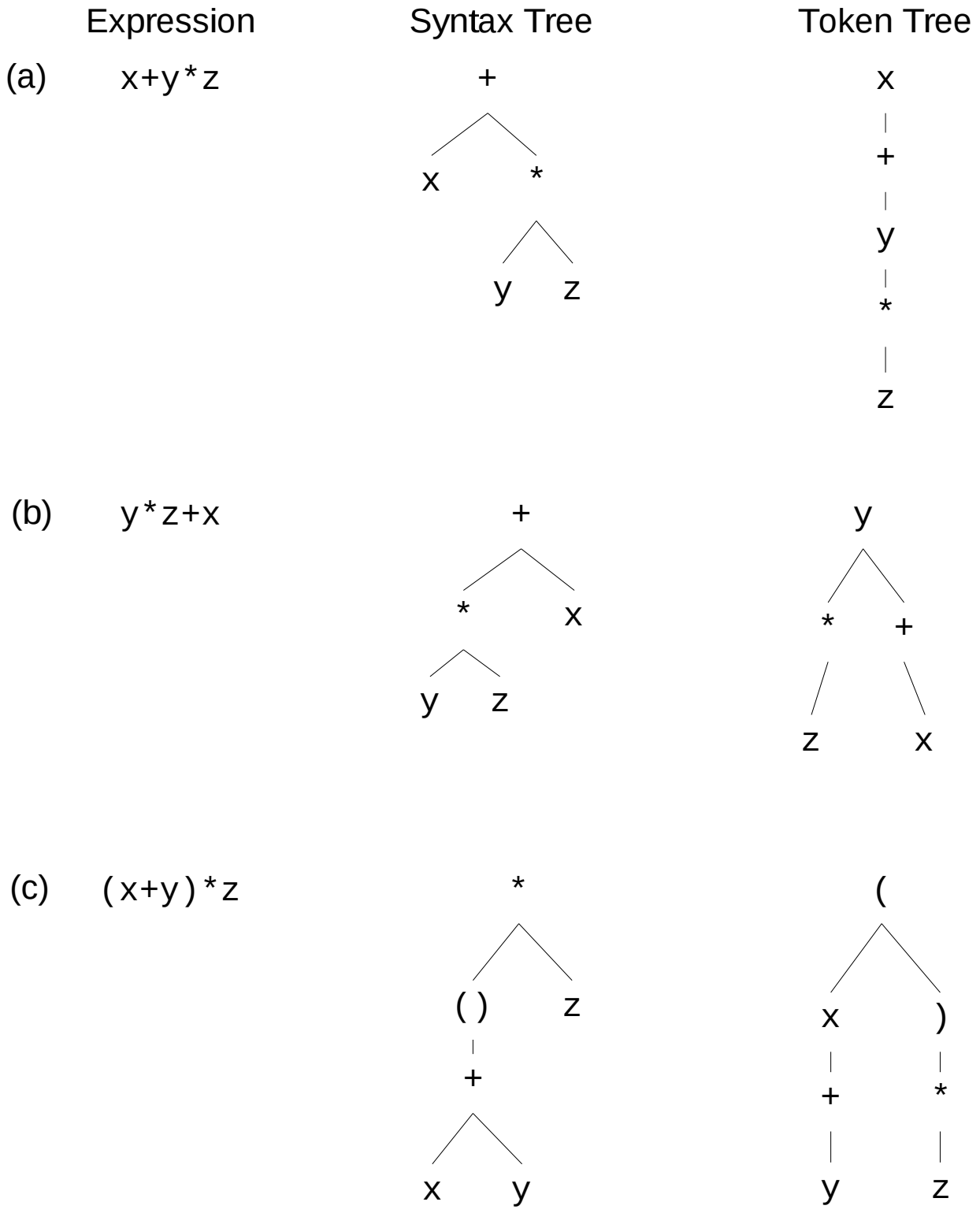



Figure 6.9: Comparison of token trees with syntax trees for several arithmetic expressions (parsed with conventional precedence, which gives multiplication higher precedence than addition).

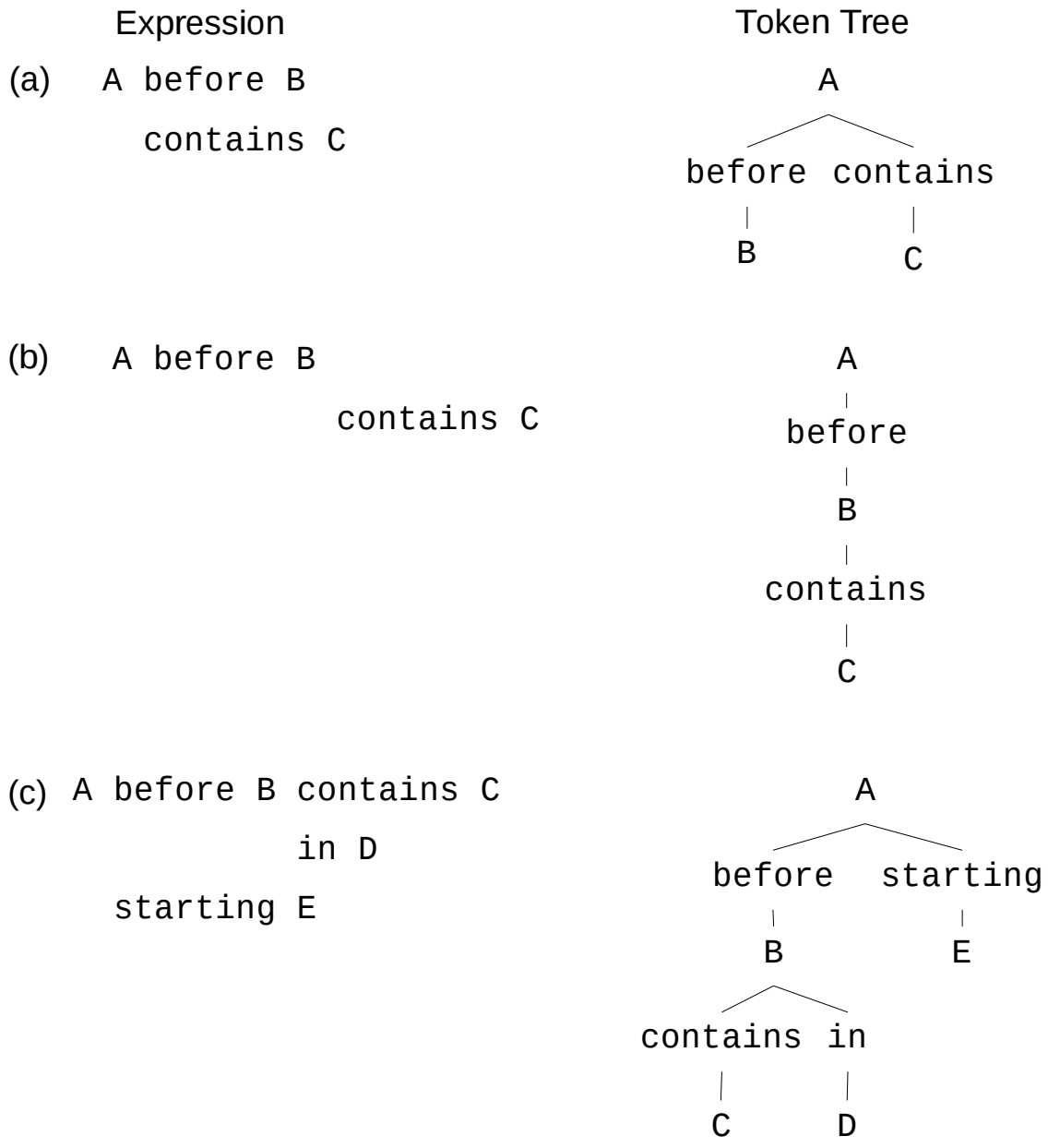



Figure 6.10: Token trees for several TC expressions.

If the stack does not contain a matching `from` token, however, POPTO does nothing, and the algorithm reverts to the parent indicated by indentation rules. Note that the structuring algorithm does not report any syntax errors. Syntax errors are delayed until the next phase of parsing. The tokenization and structuring algorithms are designed to always generate *some* token tree, even if it might be syntactically invalid, so that the token tree can be used to display tie lines (Figure 6.8) continuously while the user is editing a pattern.

**Algorithm 6.1** PARSEINDENTATION constructs the token tree for a TC expression from its indentation.

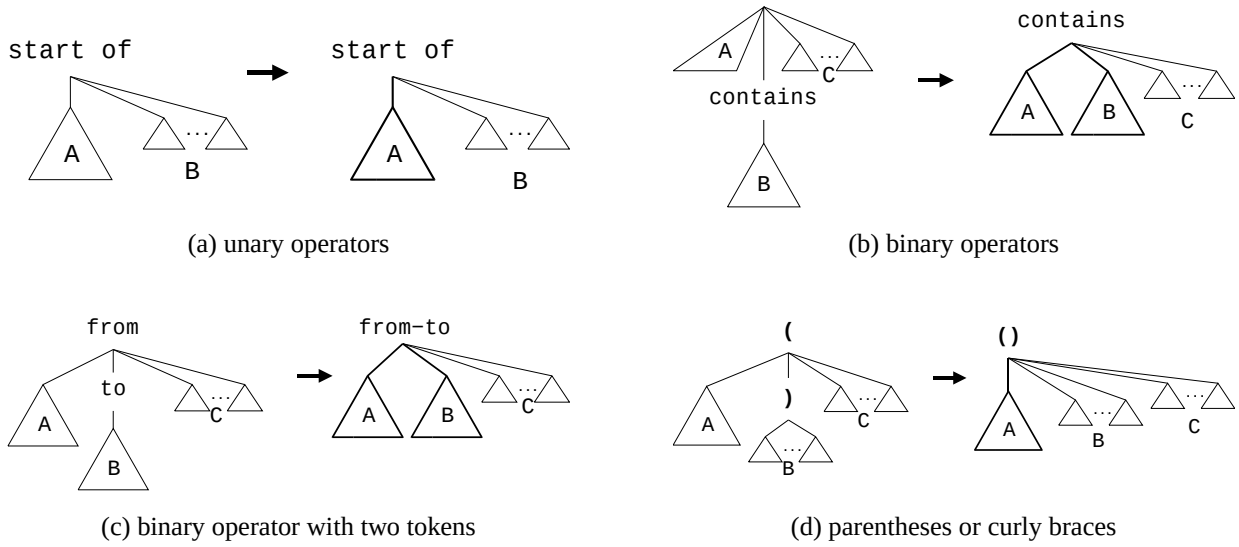
```
PARSEINDENTATION()
 1  stack ← new STACK
 2  PUSH(stack, new TOKEN( ROOT , column = −1))
 3   for each t in token sequence from tokenizer
 4   do while TOP(stack).column ≥ t.column and TOP(stack) is not an unmatched ( or {
 5        do POP(stack)
 6        if t = OR
 7           then POPTO(stack, EITHER )
 8        else if t = TO
 9           then POPTO(stack, FROM )
10        else if t = )
11           then POPTO(stack, ( )
12        else if t = }
13           then POPTO(stack, { )
14        add t as a new child of TOP(stack)
15        PUSH(stack, t)
16

POPTO(stack, token)
 1   for each t in stack from top of stack down
 2   do if t = token
 3         then pop off all tokens on stack above t
 4              return
 5   return (with stack unchanged)
```

Parentheses and curly braces form a special case. The stack cannot be popped past a left parenthesis or curly brace until it has been matched by a right parenthesis or curly brace (line 4). When a right parenthesis or brace is encountered, it pops off all tokens until it reaches a left parenthesis or brace. Thus an expression inside parentheses or braces is completely isolated from the tokens outside it, so the indentation of the parenthesized expression can be independent of the expressions around it.

### Syntax Checking

After the token tree has been constructed, it is checked for violations of syntax rules:




(a) unary operators

(b) binary operators

(c) binary operator with two tokens

(d) parentheses or curly braces

Figure 6.11: Rules for translating a token tree into a syntax tree. Light lines are token tree edges, and dark lines are syntax tree edges. Rules are applied to nodes in preorder, left-to-right, until all token tree edges have been transformed into syntax tree edges.

- operators with the wrong number of operands (e.g. `contains` dangling at the end of a line, with no right-hand side)
- incomplete multi-token operators (e.g. `from` without a corresponding `to`)
- unbalanced parentheses or curly braces
- operands of the wrong syntactic category (e.g. `prefix` and `is` require that one operand be an identifier)

All these tests can be made by a single pass over the token tree, checking local rules at every node. Failed tests display an error message to the user.

### Translation

Finally, the token tree is transformed into a tree that can be evaluated, i.e., an abstract syntax tree in which the operands of each operator are its children. Although the token tree could conceivably be evaluated instead, such an evaluation would need to pass results *down* the tree as well as *up*. For example, evaluating the arithmetic expression token tree shown in Figure 6.9(b) would involve first computing the result of $y * z$ in bottom-up fashion, then passing that result down to the $+$ node to be added to $x$. Evaluating the syntax tree in Figure 6.9(b) is much simpler, because the evaluation is entirely bottom-up.

The transformation from a token tree to a syntax tree follows simple local rules. The rules for representative classes of operators are depicted in Figure 6.11.

```
Roundtrip Fares Departing From BOSTON, MA To
-------------------------------------------------
        $109            INDIANAPOLIS, IN
        $89             PITTSBURGH, PA

Roundtrip Fares Departing From PHILADELPHIA, PA To
-------------------------------------------------
        $79             BUFFALO, NY
        $89             CLEVELAND, OH
        $89             COLUMBUS, OH
        $89             DAYTON, OH
        $89             DETROIT, MI
        $79             PITTSBURGH, PA
        $79             RICHMOND/WMBG., VA
        $79             SYRACUSE, NY
```

Figure 6.12: Excerpt from an email message announcing cheap airfares.

## 6.3 Examples

This section gives some examples of using TC patterns to describe structure in text. The examples are divided into three kinds: plain text, web pages, and source code. More examples of TC patterns can be found in the LAPIS pattern library (Appendix A).

### 6.3.1 Plain Text

Patterns for plain text typically refer to delimiters like punctuation marks and linebreaks. Consider the following example of processing email messages. Several airlines distribute weekly email announcing low-price airfares. An excerpt from one message (from US Airways) is shown in Figure 6.12.

Describing the boundaries of the table itself is straightforward:

```
Table is from "Roundtrip Fares"
          to BlankLine
```

The rows of the table are just lines:

```
Flight is Line starting "$"
Fare is Number just after "$"
```

The origin and destination cities can be described in terms of their boundaries:

```
Origin is from AllCapsWord just after "From"
            to AllCapsWord just before "To"
            in Line starting Table
Destination is from AllCapsWord just after Fare
                 to end of Line
```

**AlphaWindow**,
Cumulus Technology Corp.,
1007 Elwell Court,
Palo Alto, CA, 94303,
(415) 960-1200,
$750,
Unix, **Discontinued**,
Alpha-numeric terminal windows, Window System

**Altia Design**, Altia,
5030 Corporate Plaza Dr \#300,
Colorado Springs, CO, 80919,
(800)653-9957 or (719)598-4299,
UNIX or Windows, IB

**Amulet**,
Brad Myers,
Human-Computer Interaction Institute,
Carnegie Mellon Univ,
Pittsburgh, PA, 15213,
(412) 268-5150,
amulet@cs.cmu.edu,
**FREE**,
X or MS Windows, portable toolkit, UIMS

Figure 6.13: Excerpt from a web page describing user interface toolkits.

Using these definitions, we can readily filter the message for flights of interest, e.g. from Boston to Pittsburgh:

```
Flight contains Destination contains "PITTSBURGH"
       in Table contains Origin contains "BOSTON"
```

The expression for the flight's origin is somewhat convoluted because flights (which are rows of the table) do not contain the origin as a field, but rather inherit it from the heading of the table.

### 6.3.2 Web Pages

Many web pages display data in a custom format, using HTML markup to set off important parts of the text typographically or spatially. Figure 6.13 shows part of a page describing user interface toolkits [Mye94].

The page describes over 100 toolkits with various properties: some are free, some are commercial; some run on Unix, others Microsoft Windows, others Macintosh, and others are cross-platform. To browse the page conveniently, the user might want to restrict the display to show only toolkits matching certain requirements – for example, toolkits running under both Unix and Microsoft Windows, sorted by price.

Each toolkit on this page is contained in a single HTML paragraph element. So the user might start by describing the toolkit as the Paragraph element, which is identified by the built-in HTML parser as `[P]`

```
Toolkit is [P]
```

Finding the prices is straightforward using Number, a pattern defined by the built-in US English parser:

```
Price is either "$" then Number
            or "FREE"
            in Toolkit
```

Finding toolkits that run under Macintosh is simple, since the page refers consistently to Macintosh as “Mac”. But Unix platforms are variously described as “X”, “X Windows”, or “Motif”, and Microsoft Windows is also called “MS Windows” or just plain “Windows”. One can deal with these problems by defining a pattern for each kind of platform that specifies all these possibilities and further constrains the matched literal to be a full word (not just part of a word):

```
Macintosh is Word = "Mac"
Unix is Word = either "Unix" or "X" or "Motif"
MSWindows is Word = either "PC"
                        or "Windows" not just after "X"
```

Using these definitions, the user can can readily filter the web page for toolkits matching certain requirements, e.g.:

```
Toolkit contains Unix
        contains MSWindows
```

for toolkits that run on both Unix and Microsoft Windows.

### 6.3.3 Source Code

Source code can be processed like plain text, but with a parser for the programming language, source code can be queried much more easily. LAPIS includes a Java parser, so the examples that follow are in Java.

Unlike other systems for querying and processing source code, TC operates on regions in the source text, not on an abstract syntax tree. At the text level, the user can achieve substantial mileage knowing only a few general types of regions identified by the parser, such as `Statement`, `Comment`, `Expression`, and `Method`, and using patterns to specialize them. For example, the Java parser in LAPIS identifies `Comment` regions, but does not specially distinguish the “documentation comments” that can be automatically extracted by Java’s `javadoc` utility. Figure 6.14 shows a Java method preceded by a documentation comment.

The user can find the documentation comments by constraining `Comment`:

```
DocComment is Comment starting "/**"
```

```
/**
 * Convert a local filename to a URL.
 * @param file File to convert
 * @return URL corresponding to file
 */
public static URL FileToURL (File file)
                    throws MalformedURLException {
    return new URL ("file:"
                    + toURLDelimiters
                        (file.getAbsolutePath ()));
}
```

Figure 6.14: A Java method with a documentation comment.

A similar technique can be used to distinguish public class methods from private methods:

```
PublicMethod is Method starting "public"
```

In this case, however, the accuracy of the pattern depends on programmer convention, since attributes like `public` may appear in any order in a method declaration, not necessarily first. All of the following method declarations are equivalent in Java:

```
public static synchronized void f ()
static public synchronized void f ()
synchronized static public void f ()
```

If necessary, the user can deal with this problem by adjusting the pattern (e.g., `Method starting Line contains "public"`) or relying on the Java parser to identify attribute regions (e.g., `Method contains Attribute equal to "public"`). In practice, however, it is often more convenient to use typographic conventions, like `public` always appearing first, than to modify the parser for every contingency. Since TC patterns can express such conventions, pattern matching might also be used to enforce them, if desired.

One can use `DocComment` and `PublicMethod` to find public methods that need documentation:

```
PublicMethod not just after DocComment
```

Java documentation comments can include various kinds of fields, such as `@param` to describe method parameters, `@return` to describe the return value, and `@exception` to describe exceptional return conditions. These fields can be described by TC expressions:

```
DocField is from "@"
             to end of Line
             in DocComment
ParamDoc is DocField starting "@param"
ReturnDoc is DocField starting "@return"
ExceptionDoc is DocField starting "@exception"
```

Using this structure, one can find methods whose documentation is incomplete in various ways. For example, this expression finds methods with parameters but no parameter documentation:

```
PublicMethod contains FormalParameter
             just after DocComment
                          not containing ParamDoc
```