Lexiparse: A Lexicon-based Parser for Lisp Applications
** DRAFT 0.93 ***

Drew McDermott
Department of Computer Science
Yale University
drew.mcdermott@yale.edu

September 6, 2005

Change history:

V. 0.93	2005-09-06:	Various bug fixes, in both the implementation and the manual.
V. 0.9	2004-11-25:	First alpha release

1 Grammars and Parsetrees

This manual describes a deterministic, recursive-descent, lexicon-based parser (called "Lexiparse").[1] Lexiparse works on a grammar organized around *lexemes,* to which are attached syntactic and "semantic" processing instructions. Grammars also include specifications of how to construct lexemes from streams of characters. The parser uses the lexical and syntactic specifications to turn strings into sequences of *parsetrees.* The "semantics" is expressed using *internalizers* that then turn parse trees into whatever data structures the syntactic structures are intended to correspond to.

The parser and lexer are built on an abstraction called the *generator*, which represents a lazy list. At the highest level the parser can be thought of as a generator of *internalized objects* from a stream of characters, which is decomposed thus:

$$\begin{tikzcd} \textit{chars} \arrow[r] & \begin{array}{c}\texttt{LEXICAL}\\\texttt{SCANNING}\end{array} \arrow[r, "\textit{lexemes}"] & \begin{array}{c}\texttt{SYNTACTIC}\\\texttt{ANALYSIS}\end{array} \arrow[r, "\textit{parsetrees}"] & \begin{array}{c}\texttt{INTERNAL-}\\\texttt{IZATION}\end{array} \arrow[r] & \textit{internalized objects} \end{tikzcd}$$

[1]The basic design comes from a parser Vaughn Pratt showed me in 1976 or thereabouts for an alternative Lisp syntax he called CGOL [Pra76].

Each stage (in italics) is a stream that is transformed into a stream whose elements are bigger chunks. Syntactic analysis is broken down into three subphases: grouping, tidying, and checking, which will be described below.

Parsetrees are the central organizing framework for many of Lexiparse's actions. A parsetree has an *operator* and a list of *subtrees*. For instance, the expression `a+b+c` might get parsed into a tree with operator `plus` and three subtrees, one for each operand. Associated with the operator are all the actions for creating, checking, and transforming the parsetree.

A *grammar* is a system of lexical, syntactic, and internalization definitions that specify how to build the parsetrees of a language and (optionally) transform them into an internal representation. A grammar is defined starting with the `def-grammar` macro, whose syntax is:

```
(def-grammar name
          :sym-case [ :preserve | :up | :down]
          :top-tokens top-token-spec
          (:definitions !())
          (:parent name_p)
          (:lex-parent name_l)
          (:syn-parent name_s)
          (:int-parent name_i)
          (:replace ()))
```

The *name* becomes a global variable whose value is this grammar, and it is also entered in a global table of grammar names. The `:sym-case` (default `:preserve`) determines whether characters are up-cased, down-cased, or unchanged before any of the later processes see them. See section 2. The `:top-tokens` argument specifies which lexical tokens are allowed to appear as the top operator in parsetrees of the grammar; see section 3. The actual syntactic, lexical, and internalizer definitions of the grammar can be given as the `:definitions` argument of `def-grammar`, or can be specified separately using the `with-grammar` macro (see page 5).

Grammars can be organized into inheritance structures, the definition of which is the job of the keyword arguments `:parent`, `:lex-parent`, `:syn-parent`, and `:int-parent`. See section 4.1. The keyword `:replace` is explained in section 3.5.

As a running example I will use a simple grammar called "Hobbsgram." I have been collaborating with Jerry Hobbs and others on the development of DAML-Time, an axiomatic temporal ontology [HFA$^+$02]. I prefer to use machine-readable axioms that can be directly imported into our inference systems. Jerry prefers an informal infix notation. So every time he would send me something like this

```
(A x,y)[[p1(x) & p2(y)] --> [q1(x,y) & q2(y)]]
```

I would have to translate it by hand into this:

```
(forall (x y) (if (and (p1 x) (p2 y)) (and (q1 x y) (q2 y))))
```

so I my software could type-check it.

Eventually I got tired of this, and wrote a little grammar to do the translation. The grammar took a couple of hours to write. (Some of it was cribbed from other grammars I have written.) I've made a couple of changes to the original for pedagogical purposes.

Lexiparse is intended for writing front ends for Lisp programs that process alien syntaxes. If you don't plan to write your main program in Lisp, and especially if you don't want to learn Lisp, it's probably not for you. In what follows I will assume some knowledge of Lisp, especially in the sections on internalizers. I use various facilities from [McD03], the YTools package of Lisp extensions.

This manual is written using a "literate programming" style. The grammar is interleaved into the manual in such a way that it can be extracted into a legal Lexiparse file. We start the file with the usual boilerplate:

```
<<Define File hobbsgram.lisp
;-*- Mode:  Common-lisp; Package:  ytools; Readtable:  ytools; -*-
(in-package :lexiparse)

(depends-on %lexiparse/ parser)

<<Insert:  sym-chars (p. 8)>>

(def-grammar hobbsgram
                   <<Insert:  hobbs-top-tokens (p. 29)>>

<<Insert:  hobbs-lexer (p. 5)>>
<<Insert:  hobbs-syn (p. 13)>>
<<Insert:  hobbs-internalizer (p. 32)>>
>>
```

For an explanation of the <<*Define* ...<<*Insert* ...>>>> notation, see appendix A.

2 Preliminaries: Generators and Lexical Scanning

To use Lexiparse you don't need to create generators, but it helps to know how to use them. Their API is very simple. The generator represents a "partially consumed" sequence of objects.[2] The objects are "consumed" by the program using the generator. Its state can be modeled as two lists, `prev` and `remaining`. The second list, `remaining`, contains the objects to be generated; `prev` represents the last object already generated. Moving an object from `prev` to `remaining` or vice versa allows for "peeking" and "unpeeking" on the sequence. Because `prev` has at most one element, one can only "unpeek" by one step.

The operations on a generator are:

`(generator-done` *g*`)` True if *g* can generate no more elements, that is, if *g*`.remaining` is empty.

`(generator-advance` *g*`)` Move one object from *g*`.remaining` to *g*`.prev`. Return that object. Generates an error if *g*`.remaining` is empty.

`(generator-reverse` *g*`)` Move one object from *g*`.prev` to *g*`.remaining`. Return that object. Generates an error if *g*`.prev` is empty.

`(generator-peek` *g*`)` Return the next object *g*`.remaining` without changing the state of *g*. If *g*`.remaining` is empty, returns `nil`.

The function `(print-gen` *g*`)` just prints every element generated by generator *g*. This can be useful for debugging.

The first stage of Lexiparse, *lexical scanning*, is to transform a generator of characters into a generator of *lexemes*, the basic organizing units of the grammar. Two standard character generators are `(string-gen` *`string`*`)` and `(file-gen` *`filename`*`)`, which create generators of all the characters in a string or file.

Before anything else is done with these characters, their case is normalized according to the `:sym-case` argument of the governing grammar. If the argument is `:preserve`, the character is left alone. If it is `:up` or `:down`, it is made upper-case or lower-case. In a case-sensitive Lisp, `:preserve` is the obvious choice; in a grammar for a case-insensitive language, written in an ANSI Lisp environment, it may be more natural to convert symbols to upper-case for reasons explained in section 3.

Lexical analysis is defined in terms of *lexical actions* associated with each character. In general a lexical action is a Lisp function of two arguments,

[2]The sequence might be infinite, but in the parsing application it never will be.

the current character generator and the current grammar. The function should advance the generator by one or more characters, and return a list of zero or more lexemes. (It should return zero only when the character generator is done.) Many standard lexical actions are predefined. In the simplest case a character gives rise to a single lexeme. A lexeme is either a *token* or a Lisp object, usally a symbol, string, or number. A token is an abstract entity corresponding to a syntactic entity in the language. I will typically give tokens names, such as `left-paren` or `right-arrow`, that suggest their lexical representation. To distinguish tokens from symbols, I will put angle brackets around them. So the character "`(`" is transformed to the token `<left-paren>`, whose name is the symbol `left-paren`.

We now define the code segment `hobbs-lexer` that was inserted above, although it too has holes to be filled in later.

```
<<Define hobbs-lexer
(with-grammar hobbsgram

   (def-lexical #\( (token left-paren))
   (def-lexical #\[ (token left-bracket))
   <<Insert:  other-easy-lexicals (p. 6)>>
   <<Insert:  dispatch (p. 7)>>
   <<Insert:  other-simple-dispatches (p. 7)>>
   <<Insert:  sym-def (p. 8)>>
   <<Insert:  num-def (p. 8)>>
)>>
```

The full form of `def-lexical` is

```
    (def-lexical c e [:name N] [:grammar r])
```

c is a character set (see below). N is a name for the set (purely for decorative purposes, but mandatory if the set has more than one element). r is the grammar in which the lexical information is being registered. If omitted, it defaults to the grammar specified by the enclosing `with-grammar` or `def-grammar` call (see below).

That leaves e, which is an expression that evaluates to a Lisp function h to handle characters as they occur in a generator g. When a character in the set c is seen, h is called with two arguments: g and the current grammar, and it is expected to return a list of lexemes. Such a procedure h is called a *lexical action.* (The "current grammar" that is passed to the lexical-action procedure isn't necessarily the grammar in force when the action was defined; see section 4.1).

Obviously, given where `(token left-paren)` occurs, `token` takes one (unevaluated) symbol as argument and returns a procedure that returns a list consisting of the token whose name is that symbol. In other words, associating the action `(token left-paren)` with the character `#\(` means that an occurrence of "(" in the character stream will produce an occurrence of the token `<left-paren>` in the lexeme stream. (Tokens are printed in the angle-bracket notation suggested by this example.)

We avoid specifying a `:grammar` argument to every use of `def-lexical` using the macro `(with-grammar` *g* `--`*`definitions`*`--)`, which makes *g* the default grammar in all the *definitions*, including `def-syntactic` and all the other `def-` macros defined below. If you prefer, you can put some or all of these definitions inside a `:definitions` argument of the `(def-grammar` *g* `...)` that defines the grammar in the first place:

```
(def-grammar name
   ...
   :definitions
      ((def-lexical ...)
       ...))
```

A long series of applications of `def-lexicals` defining single-character tokens can be condensed using the macro `(def-lexical-tokens` *l*`)`, where *l* is a list of pairs `(`*`char-set toknname`*`)`.

```
<<Define other-easy-lexicals
   (def-lexical-tokens ((#\, comma)
                        (#\) right-paren)
                        (#\] right-bracket)
                        (#\{ left-brace)
                        (#\} right-brace)
                        (#\~ not)
                        (#\/ divide)
                        (#\+ plus)
                        (#\& and)))
>>
```

A *char-set* is in general a list of characters, but if it has only one element, the parens around it can be suppressed. The same convention applies to `def-lexical` and other constructs referring to sets of characters. (We will further generalize the notion of *char-set* below.)

Many tokens correspond to *sequences* of characters. The lexical action associated with the first character in the sequence must then be to look at the next character(s) and decide how to proceed. We call this *dispatching* on the following characters. For instance, in the Hobbs grammar, there are three symbols '`-->`', '`--`', and '`-`', whose construction is governed by the following:

```
<<Define dispatch
   (def-lexical #\-
                (dispatch
                    (#\-
                        (dispatch (#\> (token right-arrow))
                                  (:else (token double-dash))))
                    (:else (token minus)))) >>
```

The `dispatch` action is of the form `(dispatch ---clauses---)`, where each *clause* is of the form

`(char-set lexical-action)`
or
`(:else lexical-action)`

As before, a *char-set* is a list of characters, or a single character and a *lexical-action* is (an expression that evaluates to) a lexical action as defined above. Obviously, one possible lexical action is a further series of `dispatch`es.

Here are how some other multi-character tokens are produced:

```
<<Define other-simple-dispatches
   (def-lexical #\= (dispatch
                       (#\< (token leq))
                       (:else (token equals))))

   (def-lexical #\< (dispatch
                       (#\= (token leq))
                       (#\-
                            (dispatch
                               (#\- (dispatch (#\> (token double-arrow))
                                              (:else (token left-arrow))))
                               (:else (token left-arrow))))
                       (:else (token less))))

   (def-lexical #\> (dispatch
                       (#\= (token geq))
                       (:else (token greater)))) >>
```

Almost all languages have a notion of symbols and numbers. Lexiparse provides some built-in facilities for defining these lexical classes. First, the character sets involved tend to be of the form “any letter” or “any lower-case letter.” To designate these, we broaden our lists of characters to include subgroups of the form

`#\`c_1 `- #\`c_2

Then symbols are defined thus:

```
<<Define sym-chars
(defvar hobbs-lexical-chars* '(#\- #\_ #\! #\*))
>>
```

```
<<Define sym-def
   (def-lexical (#\* #\_ #\a - #\z #\A - #\Z)
                (lex-sym hobbs-lexical-chars*)
       :name alphabetic)
>>
```

The lexical action `(lex-sym` E`)` gobbles up characters that are alphanumeric or in the set E until a character outside those groups is found, and returns the result as either a symbol token, or the token corresponding to a reserved word of the grammar (as specified by `def-syntactic`, described in section 3). In the Hobbs language, asterisks do not refer to multiplication, but are used as ordinary symbol constituents. This doesn't prevent us from using the one-character symbol “`*`” as a reserved word, but it does mean that “`a*b`” will be interpreted as a single symbol; you would have to write “`a * b`” to make the symbol “`*`” visible. (See the definition of “`*`” on page 15.)

The function `lex-number` is similar but simpler. The current version just tries to read a sequence of decimal digits and periods as a Lisp number.

```
<<Define num-def
   (def-lexical (#\0 - #\9) #'lex-number
     :name numeric) >>
```

The procedures `lex-positive-number` and `lex-negative-number` can be used following occurrences of “`+`” or “`-`”. One might write

```
   (def-lexical #\+ (dispatch
                      ((#\0 - #\9) #'lex-positive-number)
                      (:else (token plus))))
```

However, the Hobbs language does not work this way, because signed numbers didn't happpen to arise in the axioms.

To write your own lexical-analysis function, keep in mind that the first argument is the character generator, with the triggering character still waiting to be advanced over. The second argument is the current grammar, which will often be ignored. The function must return a list consisting of zero or more elements, each a token or terminal item (string, number, or symbol).

The procedure `(chars->lex` *r g*`)` takes a character generator *r* and a grammar *g*, and returns a generator of *g* lexemes.

3 Syntactic Analysis

The rest of the grammar is oriented around lexemes corresponding to tokens and reserved words of the language. All syntactic and "semantic" operations are associated with particular lexemes. The flavor of the resulting grammar is much like a system of macros in Lisp. The precedences specify grouping information, and once everything is grouped into a tree structure the remaining syntactic information is expressed in terms of transformations on the tree nodes.

The tree structure is expressed using the `Parsetree` data type. A *parsetree* is a Lisp `defstruct` structure with three slots visible to application programmers:

1. `(Parsetree-operator pt)`: The operator, a token. The name "operator" stems from the paradigmatic case where the tree denotes the application of an operation such as "+" to the arguments denoted by the subtrees. But it also connotes the activity of gathering the arguments together during parsing.

2. `(Parsetree-fixity pt)`: One of `:infix`, `:prefix`, or `:suffix`, depending on what syntactic role the operator played in the creation of this parsetree.

3. `(Parsetree-subtrees pt)`: A list of constituents. Some are lexemes, some are parsetrees.

For conciseness, I will express the "structure" of a parsetree using parentheses thus: `(:^+` *`operator --subtrees--`*`)`, usually with the fixity omitted. So a parsetree with operator `plus` and two subtrees, the second of which is a parsetree with operator `times`, might have the structure `(:^+ plus a (:^+ times 3 x))`. (The `:^+` symbol is mean to suggest a little tree structure.)

In most contexts the symbols `[*_]`, `[_*]`, and `[_*_]` may be used as synonyms for `:prefix`, `:suffix`, and `:infix`, respectively. When a parsetree is printed, or displayed using `parsetree-show` (appendix B), the variable `show-parsetree-fixity*` controls whether the fixity is displayed; if it has value true, one of these picturesque glyphs will be used instead of the corresponding keyword.

Parsetrees are produced using the syntactic properties of lexemes. A lexeme is either a `terminal` or an `operator`. An operator is defined using this macro:

```
(def-syntactic name [:lex [ nil | string]]
                    [:reserved b]
                    [:grammar grammar-name]
                    [:prefix prefix-info]
                    [:infix infix-info]
                    [:suffix suffix-info]
                    [:tidier tidier-spec]
                    [:checkers (checkers-spec*)]
                    [:internalizer internalizer-spec])
```

The keyword arguments can occur in any order; each can appear at most once. The `:grammar` argument is the grammar in which *name* is defined. It is usually obviated by the use of `with-grammar`.

The `:lex` keyword specifies the string, if there is one, that produces this token when processed by the lexical analyzer (section 2). This string is used in printing the name of the token, but is not otherwise used by Lexiparse.[3] A token prints as `<`*name*`>` if it has no `:lex` property, otherwise as `<`*name* `"`*lex*`">`. The `:reserved` keyword says whether the *token* is a reserved word of the language. If b = false (the default), then the only way the token can occur is via lexical analysis (or by being introduced during tidying; see section 3.3). If b = true, then a symbol with the name *token* will be treated as (and converted to) a syntactic token. In some cases there can be a reserved word with name w and a lexical token with the same name. In that case, because the syntactic action is associated with w, the two are equivalent. (In the Hobbs grammar, one example is the use of "`&`" and "`and`"

[3]It might be a good idea to have the system check that the lexical definitions and `:lex` declarations are consistent, but it's not entirely clear what that would mean. So treat `:lex` directions as comments, and remember to always keep your comments up to date!

as ways to signify conjunction.)[4] Any symbol is treated as a vanilla terminal symbol unless it is declared to be a reserved word. Numbers and strings are always terminal.

How tokens are printed is controlled by the global variables `print-token-long*` and `print-token-gram*`. Normally a token is printed simply as its name between angle brackets, as in `<and>`. But if `print-token-long*` is true, then single quotes are printed around the name of a reserved word, and the name is followed by the `:lex` property if it not `nil`. So the token `<and>` would be printed as `<'and' "&">` in "long mode." If in addition `print-token-gram*` is true, then the lexical information is followed by the name of the grammar the token comes from, as in `<'and' "&" hobbsgram>`.

Note that in ANSI Lisp, if the `:sym-case` value for the current grammar is `:preserve`, then symbols extracted from a character stream will print funny. The string "`do while (x>0) ...`" will become the lexeme stream "`|do| |while| LEFT-PAREN |x| GREATER 0 ....`" If `do` is a reserved word, it will have to be defined as `(def-syntactic |do| ...)`. To avoid having to write it that way, specify "`:sym-case :up`," and the lexeme stream will be "`DO WHILE LEFT-PAREN X GREATER 0 ....`" Now you can write `(def-syntactic do ...)`, without the vertical bars.

The rest of section 3 of this manual is devoted to explaining the remaining fields of `def-syntactic`. The next section explains `:prefix`, `:infix`, and `:suffix`. Section 3.2 explains how bracketed expressions work. In section 3.3 I discuss the `:tidier` and, in section 3.4, the `:checkers`. Section 3.5 treats the topic of "local grammars." In section 4 we finally leave syntax and talk about how parsetrees, having been built, tidied, and checked, get turned into arbitrary internal data structures.

3.1 Grouping

A syntactic token can occur as a prefix, infix, or suffix operator, depending on whether it precedes its arguments, occurs amidst its arguments, or follows its arguments. The same token can be used as a prefix operator and as an infix or suffix operator, but it can't occur sometimes as an infix and sometimes as a suffix.

Each kind of token requires somewhat different information, but the

[4]Because any symbol becomes a token when incorporated into a parsetree, the set of tokens and the set of syntactic operators are coextensive. I will tend to use the word "token" to refer to one of these objects when focusing on how it gets produced, and the word "operator" when focusing on how it influences the parsing process and how it is interpreted after a parsetree headed by it is built.

concept of *precedence* is central to all. The idea is simple. Suppose the following situation arises during syntactic analysis of a token string:

$$\ldots x_1\ [op_0]\ op_1\ x_2\ op_2 \ldots$$

where x_1 and x_2 are parsetrees or terminals, i.e., outputs from the parsing that has been done so far. op_1 is a prefix operator, if op_0 is present, an infix operator if it's absent. op_2 is an infix or suffix operator. The question is, do we group expressions so that x_2 gets incorporated into a parsetree with operator op_1 or do we group so that x_2 gets incorporated into a parsetree with operator op_2? (In the case where op_1 is infix, the choice boils down to whether x_2 is grouped with x_1, or with some expression to the right of op_2.) The question is decided by consulting the precedences of the two operators, which determine which direction x_2 is pulled. If the *right precedence* of op_1 is greater than or equal to the *left precedence* of op_2, then x_2 is pulled toward op_1; otherwise, it's pulled toward op_2. Right precedence determines how hard an operator pulls an operand to its right; left precedence, how hard it pulls an operand to its left. Another way to picture it is to imagine that precedences create "virtual parentheses." If op_1's right precedence is greater than or equal to op_2's left precedence, then it's as if there are parens around x_1 and x_2:

$$\ldots \texttt{(}x_1\ [op_0]\ op_1\ x_2\texttt{)}\ op_2 \ldots$$

If op_2's left precedence is higher than op_1's right precedence, then there is a "virtual left parenthesis" before x_2:

$$\ldots x_1\ [op_0]\ op_1\ \texttt{(}x_2\ op_2 \ldots$$

whose mate will eventually be found to the right of op_2 by repetition of the same sort of calculation.

If op_1 is a prefix operator, then the parser must do a very similar calculation, using left and right precedences exactly as before, the only difference being that the left virtual parenthesis comes right before op_1. Similarly, if op_2 is a suffix operator, the virtual right paren would come right after op_2.

The first phase of syntactic analysis is called *grouping* because it uses precedence, fixity, and "context" information to decide which operands go with which operators.

Context information is specified in the definition of each lexeme. The *prefix-info* clause for `def-syntactic` looks like this:

```
    :prefix (:precedence prec
             :context [numargs
```

```
                    | (:open separator closer)]
             [:local-grammar (<local-gram-spec>+)]])
```

(The `:precedence`, `:context`, and `:local-grammar` information may appear in any order. `:local-grammar` is discussed in section 3.5.)

The `:context` consists of either an integer ≥ 0 or the symbol `:open` followed by two lexeme names (or lists of lexeme names). The first case is for prefix operators that take a fixed number of arguments. (In this case one can write `:numargs` instead of `:context`.) The second case is for an operator that behaves like an open bracket (e.g., a left parenthesis). The lexeme *closer* is the corresponding close bracket. If the `closer` is a list of lexemes, then any of them is acceptable as the closing bracket. The lexeme *separator* is the token one expects to see between the elements inside the brackets (e.g., `<comma>` for `<left-paren>`). There must be at least one `closer`, but the separator can be omitted completely by writing `'()` or `nil`. I'll explain how brackets work in section 3.2.

First, here's a list of the non-bracket prefix operators of the Hobbs grammar:

```
<<Define hobbs-syn
(with-grammar hobbsgram

   <<Insert:  contiguity (p. 16)>>

   <<Insert:  hobbs-brackets (p. 19)>>

   (def-syntactic A :reserved t :prefix (:precedence 5 :numargs 1)
      :tidier quant-tidier)

   (def-syntactic E :reserved t :prefix (:precedence 5 :numargs 1)
      :tidier quant-tidier)

   (def-syntactic E! :reserved t :prefix (:precedence 5 :numargs 1)
      :tidier quant-tidier)

   <<Insert:  quant-tidier (p. 25)>>

  <<Insert:  quant-checker (p. 27)>>

   <<Insert:  infix-connectives (p. 15)>>

   (def-syntactic not :reserved t :prefix (:precedence 19 :context 1))
   (def-syntactic plus :prefix (:precedence 29 :numargs 1)
                       <<Insert:  infix-plus (p. 15)>>)

   (def-syntactic minus :prefix (:precedence 29 :numargs 1)
                        <<Insert:  infix-minus (p. 16)>>)

   <<Insert:  high-prec-ops (p. 15)>>
) >>
```

(The `:tidier` and `checker` for quantifiers will be explained in section 3.3.)

For infix tokens, the relevant `def-syntactic` clause is

```
    :infix ([:left-precedence left-prec
             :right-precedence right-prec
             | :precedence prec]
            :context [ :binary | :grouping
                       | (:open separator closer)
                       | :close]
            [:local-grammar name])
```

(As with prefix tokens, the precedence, `:context`, and `:local-grammar` information may appear in any order. See section 3.5 for information on `:local-grammar`.)

If the keyword `:binary` is supplied, then the parsetree headed by this token will have two subtrees, one for the left operand and one for the right. If it is `:grouping`, then it behaves much the same, except that iterated occurrences of the token will be collapsed into a single parsetree. The token `<plus>` is a good example; the expression a+b+c will be parsed as a single parsetree with three subtrees corresponding to a, b, and c.

If the `:context` spec is `(:open separator closer)`, then the effect is the same as for prefix operators, except that there is a left operand. A left paren is a prefix operator in "`(a+b)`", an infix operator in "`f(a+b)`." Further discussion is deferred to section 3.2.

Instead of supplying separate left and right precedences, one can write `:precedence` to declare them the same. Note that the precedence rules imply that two consecutive occurrences of an operator whose left and right precedences are equal will associate to the left.

```
<<Define left-paren-infix
   :.(def-syntactic left-paren ... .:
        :infix (:left-precedence 40
                 :right-precedence 0
                 :context
                    (:open comma right-paren))>>
```

Most of the infix tokens in the Hobbs grammar are of type `:grouping`:

```
<<Define infix-connectives
   <<Insert:  right-arrow (p. 15)>>
   (def-syntactic double-arrow :infix (:precedence 12 :context :grouping))

   (def-syntactic v :reserved t :infix (:precedence 17 :context :grouping))

   (def-syntactic or :reserved t :infix (:precedence 17 :context :grouping))

   (def-syntactic and :reserved t :infix (:precedence 18 :context :grouping))

   <<Insert:  binary-ops (p. 15)>>>>


<<Define infix-plus
                       :infix (:precedence 29 :context :grouping)>>


<<Define high-prec-ops
   (def-syntactic * :reserved true
                    :infix (:precedence 30 :context :grouping))

   (def-syntactic divide :infix (:precedence 30 :context :grouping)) >>
```

(Note that, as explained in section 2, "`*`" is defined as a reserved word, in contrast to, e.g., the token "`<divide>`," which is generated by occurrences of "`/`".)

The non-grouping tokens are defined thus:

```
<<Define right-arrow
   (def-syntactic right-arrow :infix (:right-precedence 13 :left-precedence 14
                                      :context :binary)) >>


<<Define binary-ops
   (def-syntactic greater :infix (:context :binary :precedence 20))

   (def-syntactic less :infix (:context :binary :precedence 20))
   (def-syntactic geq :infix (:context :binary :precedence 20))

   (def-syntactic leq :infix (:context :binary :precedence 20))

   (def-syntactic equals :infix (:context :binary :precedence 20)) >>
```

```
<<Define infix-minus
   :. (def-syntactic minus \ldots .:
                        :infix (:precedence 29 :context :binary) :.).:  >>
```

Note that all of these associate to the left except `right-arrow`.

The only thing left to explain is the *contiguity operator*, the invisible operator that separates the operands in expressions such as `(f g a)` in functional programming languages. In such languages, contiguity means function application, so `(f g a)` is what would be written `(f(g)(a))` in a more traditional notation. This syntactic token is always denoted by the symbol "`|`" (which must be written with a preceding backslash because Lexiparse uses the usual Lisp read table when reading grammar definitions). If the token is left undefined, then contiguous operands will cause a syntactic error. To define it, simply write `(def-syntactic \| ...)`. In what follows, the contiguity operator will be written `<|>`. (If a language has a token for the actual character "`|`", it should be named `vertical-bar`, or anything else but "`|`".)

In the Hobbs grammar, both implicit contiguity and explicit commas are used to separate operands, albeit in different contexts.

```
<<Define contiguity
   (def-syntactic \| :infix (:precedence 5 :context :grouping))

   (def-syntactic comma :infix (:precedence 6 :context :grouping))>>
```

In functional languages, contiguity has very high precedence, so that an expression such as "`f g a, h b`" is parsed as a tuple of two elements, `(f g a)` and `(h b)`. In the Hobbs language, it has a very low precedence, so that "`f g a, h b`" is parsed as four elements, `f`, `g`, `(a,h)`, and `b`.

The use of an invisible operator may raise some qualms about how to find the ends of operands. Suppose that operator `<$>` is a prefix operator taking two arguments, in a grammar that defines a contiguity operator. How do we parse "`$ a b c`"? One way to decide would be to examine the precedences of `<|>` and `<dollar>`. However, Lexiparse implements a simpler principle; it never uses contiguity to extend a prefix operator's argument further to the right. So "`$ a b c`" would be parsed as `(:^+ $ [*_] a b)`, with `c` left for some operator to the left to grab. (We can force it gobble more, of course, by writing, e.g., `$ a (b c)`.) This contrasts with the way infix operators find their arguments. In "`$ a + b c`," the grouping depends on whether contiguity has higher left precedence than the right precedence of `<plus>`. Furthermore, in this context there must *be* a contiguity operator in the language, or an error will be signaled. Again, we can prevent the parser from looking for the contiguity operator after `b` by using parentheses: "`$ (a+b) c`".

The remaining type of syntactic token is `:suffix`, whose `def-syntactic` clause looks like this:

```
:suffix (:precedence prec
         [ :context :close ])
```

The `:context :close` should be used for lexemes that play no role except to close an open bracket. In some circumstances, Lexiparse can use this information to detect a bracket mismatch. You still have to specify the lexeme's left precedence. A suffix operator that isn't a closer requires no `:context` declaration, because all it can be is a unary operator with one argument, the operand to its left.

The only suffix operators in the Hobbs grammar are close brackets, which will be presented in the next section.

3.2 Bracketed Expressions

An open bracket is an operator that expects two things to its right: its "contents" followed by its "closer." It's important to realize that, except for its precedence, the syntactic information associated with an open bracket has *no effect* on how tokens are grouped, but only on what happens *after* they are grouped. The grouping depends on precedences alone. When trying to parse an expression beginning with an open bracket with right precedence R, the corresponding closer must have a left precedence that is less or equal to R. The left precedence of any operator before it reaches the close bracket must be $> R$. If these constraints are obeyed, then when the parser has gobbled up as much as possible to the right of the open, the closer is the next lexeme to be dealt with. E.g., when parsing "`( a * b + c )`", the attraction of `a` by `<*>` must be more than its attraction by the left parenthesis, as must the attraction of the parsetree for `a*b` by `<plus>`. In other words, the left precedence of any operator inside the brackets must be higher than the "inner" precedences of the brackets (the right precedence of the open and the left precedence of the close), except for "shielding" effects created by bracketed subexpressions.

Given these constraints, the "inner" precedences of brackets should be the lowest in the language, and their "outer" precedences among the highest. Although precedences can be any number, it is natural to make them non-negative integers, with the inner precedences of brackets set to 0, although sometimes there are reasons to set some of them a bit higher. (I'll discuss plausible values for precedence in section 6.)

Both left and right brackets can be infix lexemes. A left bracket is infix if a symbol (or some other expression) can occur to its left, as in `rec.fcn(a,b)`. It is rarer for a right bracket to be infix, but it is necessary for languages, such as XML, with "fat parentheses." In XML, the lexeme for "`</`," `left-slash`, closes an earlier occurrence of the lexeme for "`<`," `left-angle`, but it also has an argument to its right (a tag name).

The declared separator and closer tokens for an open bracket come into play after the stuff inside has been parsed. Let <⊂> be the open bracket, <⊃> be the close bracket (we'll pretend only one is possible), and (∼) be the stuff in between. There are several special situations, not mutually exclusive or exhaustive, to consider:

1. <⊃> is an infix operator, not a simple suffix operator.
2. (∼) has one of the separator tokens as its operator.
3. There is nothing between <⊂> and <⊃>.

To explain how situation 1 is dealt with, we have to distinguish two stages of bracket processing, *pre-close* and *post-close*, defined as the processing before and after branching on the fixity of <⊃>.

Situation 3 is detected when the very next lexeme after <⊂> is <⊃>, so that (∼) is empty. In this case, the result of the pre-close stage is the parsetree `(:^+ ⊂ [*_])`, which has no subtrees. If (∼) is nonempty, then situation 2 may obtain. Don't forget that (∼) is constructed from the lexeme stream using precedence relations only. If its operator is the separator for <⊂>, then (∼) is "flattened" before being incorporated into a parsetree. That is, if (∼) has structure `(:^+ *sep* ---*subs*---)`, then it is replaced by the list of *subs*, except that each element of *subs* is flattened as well.[5] I'll use the symbol (∼∼) to refer to the pre-close result, a list of one or more subtrees.

[5]If there are multiple separators, flattening takes into account only the separator at the

Once (∼∼) has been built, post-close processing depends on whether `<⊃>` is a simple `:close` operator or an infix operator (situation 1). In the former case, the closer is discarded, and the output for the entire expression is `(:^+ ⊂ ∼∼)`. In situation 1, the parser does what it would do given any operand (the pre-close output) followed by an infix operator. Because `<⊃>` has left precedence ≤ the right precedence of `<⊂>`, the `<⊃>` will become the operator for the final tree, whose structure will be

```
(:^+ ⊃ [_*]
      (:^+ ⊂ [*_] ∼∼)
      r)
```

where r is the right operand of `<⊃>`.

The Hobbs language has a comparatively simple bracketing system, although some of the tokens involved can't be described adequately until the next section.

```
<<Define hobbs-brackets
   (def-syntactic left-brace
         :prefix (:precedence 0 :context (:open comma right-brace)))

   (def-syntactic left-paren
         :prefix (:precedence 0 :context (:open \| right-paren))
         <<Insert: left-paren-infix (p. 14)>>
      <<Insert: left-paren-tidier (p. 22)>>)

   <<Insert: left-paren-checkers (p. 26)>>
   <<Insert: left-bracket (p. 28)>>

   (def-syntactic right-paren :suffix (:precedence 0 :context :close))

   (def-syntactic right-brace :suffix (:precedence 0 :context :close))

   (def-syntactic right-bracket :suffix (:precedence 0 :context :close))>>
```

3.3 Tidying

Precedences suffice for figuring out the boundaries between phrases. But for most languages beyond simple arithmetic there is more to syntax than that.

top of the tree. If `<left-paren>` allows either `<|>` or `<comma>` as a separator (with `<|>` having higher precedence), the characters "`f a,b c)`" are parsed as `(:^+ f a (:^+ | b c))`, not `(:^+ f a b c)`.

At the lowest level a language often has recursive expressions that require a type checker to verify, although this kind of checking is beyond the scope of the machinery to be described here.

At higher levels, one often finds idiosyncratic constructs which require particular substructures, and which themselves resist being incorporated into arbitrary larger expressions. If a language has a lexeme `define` for defining procedures, it usually wants to be in contexts like this one

```
define name (--parameters--){--body--};
```

where the three subfields require particular fillers. The *name* must be a symbol, and the *parameters* must satisfying the syntax for declaring variables. The *body* may, however, be drawn from a recursive grammar with few constraints.

In a traditional *phrase-structure* grammar[6] one expresses these regularities by the use of rewrite rules such as

```
definition ::= define name ...
```

At the top of the rule chain is a rule `program ::= ...` such that every legal program must be generable starting with this rule.

In Lexiparse, we do things "backwards." We first set the precedences of all lexemes to get the grouping right, then use matching rules to check that the material incoporated into a parsetree makes sense, and to tidy it up in ways that make life easier for later transformation steps. Finally, the `:top-tokens` argument to `def-grammar` specifies the operator(s) that play the role of *program* in my hypothetical phrase-structure grammar.

Before arriving at the top of the parsetree, Lexiparse transforms raw subtrees using the "tidiers" defined by the grammar. For example, a function application will initially get parsed as

(:^+ <left-paren> [_*_] f a_1 a_2 ...a_n)

Whereas a list might be parsed as

(:^+ <left-paren> [*_] a_1 a_2 ...a_n)

We would prefer to make the difference more visible by transforming the first into a parsetree with operator `<fun-app>`.

This job is done by a procedure described by the `:tidier` field of `def-syntactic`. The syntax of the `:tidier` argument (called a *tidier-spec* in section 3) is given by

[6] such as the half-assed grammar I'm using to explain the syntax of Lexiparse definitions.

```
:tidier
   [fcn-name | lambda-form
    | (...match-clause+...)
    | fcn-form ]
```

where

lambda-form `::= [ (function ...)  | (lambda ...)  ]`

and

match-clause `::= (:?  match-pattern ...)`
(See below)

and

fcn-form is a non-atomic form that must evaluate to a function or function name.

This procedure takes two arguments, a parsetree and a grammar, and returns one of three possible values:

- an improved version of the parsetree with rough edges smoothed off;
- `nil`, meaning, Leave the parsetree as it was;
- or a list of *defects,* which get attached to the parsetree. (A single defect is converted to a singleton list.) Defects will be explained below.

Tidiers are run bottom-up; parsetrees are tidied as soon as they are created. This means that tidiers can assume that all the subtrees of a parsetree have already been tidied.

Since many tidier functions rely heavily on pattern matching, I've introduced the *match-clause* notation as a shorthand for such a function. The shorthand depends on some notation from the YTools package, namely the `match-cond` macro:

```
(match-cond datum
   ---clauses---)
```

The *datum* is evaluated, and then the *clauses* are handled the same as in `cond`, except that any *match clause,* that is, one of the form

```
(:? pat
  ---body---)
```

is handled by matching the pattern *pat* against the datum. If the match succeeds, the *body* is executed and its value returned, and no further clauses are examined. (All the variables in the patterns of a `match-cond` are bound with scope equal to the `match-cond`, and initialized to `nil`.)

If a tidier is a list containing one or more match clauses `(...(:?` p_1 `...)  ...(:?` p_k `...)  ...)`, then the function produced is

```
  (lambda (this-ptree this-grammar)
      (match-cond this-ptree ...
          ...
      (:?  p_1 ...)
          ...
      (:?  p_k ...)
          ...))
```

The function normally returns a parsetree that is the "tidied" version of what it started with. The variable names `this-ptree` and `this-grammar` are as written, and can be used inside a tidier to refer to the parsetree being checked and the grammar being used to tidy it.[7]

In the case of the `<left-paren>` operator, we would write the rules thus:

```
<<Define left-paren-tidier

:.(def-syntactic left-paren \ldots .:
      :tidier
         ((:?  ?(:^+ left-paren :fixity :prefix ?@subs)
             !~(:^+ group ,@subs))
          (:?  ?(:^+ left-paren :fixity :infix ?name ?sub1 ?@others)
             !~(:^+ fun-app ,name ,sub1 ,@others))
          (:else (defect "Function " name
                       " must have at least one argument")))>>
```

The rules check the fixity of this occurrence of `<left-paren>` to decide whether it's part of a function application or a group of expressions. Once that's decided, the operator is changed to either `fun-app` or `group`, making the intended meaning more transparent to other transformations. The last clause generates a defect in the case where a function application is of the form f`()`. See below.

Most clauses in tidiers are of the form

(:? D C)

where D is a "deconstructor" pattern (containing question-marked match variables), and C is a "reconstructor," typically a generalization of backquote that builds new parsetrees using the values of the variables in D. (Let me make clear that C can be an arbitrary Lisp form, and, of course, a tidier need not consist of a set of match clauses in the first place.)

Deconstruction operates through pattern matching. A piece of a pattern starting with a question mark plays an active role (other data are passive

[7]Which may not be the grammar where the tidier was defined; see sect. 3.5.

and must be equal to the datum being matched). `?`v, where v is a name, matches anything and sets v to the thing it matched. `?,`v matches the current value of v. Every list can have at most one *segment variable* being matched against it, written `?@v`. This expression matches any sequence of elements (possibly of length 0), and sets v to it. Because only one segment is possible at a given level, there is never any question about what sequence to match. So the pattern `(d (a b ?@xx c) ?,xx)` matches `(d (a b p q r c) (p q r))` and sets `xx` to `(p q r)`. You can use comma and atsign together, with the obvious meaning. The constructs `?_` and `?@_` match any element or any segment without setting anything.

Question marks can be used to flag various special constructs, as explained in [McD03]. `?(:||` p_1 `...`p_k `[:&` p_0`])` matches a datum if any of the p_i match. Each p_i is tried in turn until one works. If the optional field `:&` p_0 is present, then p_0 must match the datum as well. For instance, `?(:|| (a b) (p q r))` matches both `(a b)` and `(p q r)`. `(a ?@(:|| (a b) (p q r) :& ?w))` matches any list that starts with `a` and continues with either `(a b)` or `(p q r)`. It binds `w` to the tail of the list. Similarly, `?(:&` p_1 `...`p_k`)` matches only if all the p_i match. A useful variant is `?(:+` p r_1 `...`r_k`)` which matches d if p matches it and the predicates r_i are all true of d. Another is `?(:~` D`)`, which matches if and only if D does not.

The code generated by the match macro does not backtrack to undo the effects of partial matches. The match of `?(:|| (?x a) (?y b))` against `(b b)` succeeds, and sets both `x` and `y` to `b`. An unset variable retains its initial `nil` value.

You can use the Tinkertoy symbol "`:^+`" to indicate parsetree structure. The pattern `?(:^+ ?op --`*`subpats`*`--)` matches a parsetree if `?op` matches the name of its operator, and the *subpats* match the subtrees. One delves into the subtree structures by simply using `?(:^+ ...)` on subpieces. Fixity is normally ignored, but can be included by writing "`:fixity` p" anywhere in the pattern. If p is a constant (i.e., one of `:prefix`, `:suffix`, or `:infix`), then "`:fixity` p" can be abbreviated as p itself, or `[*_]`, `[_*]`, or `[_*_]`.

On the construction side, we normally use an enhanced backquote to build new parsetrees, in a way analogous to the way we used the match syntax to dissect them. The expression `!~(...)` is a generalization of the usual backquote to allow us to build structure besides S-expressions. This is where the magic glyph `:^+` comes in again. In any `!~` expression, a list structure beginning `(:^+ ...)` will be interpreted as an instruction to build a parsetree, not a list structure. The first element will be the (name of the) operator and the remaining ones the subtrees. Fixity defaults to `:prefix`, but can be changed by writing, anywhere in the list, `...:fixity` v `...`, or just *v*, assuming *v* is a legal fixity designator (one of `:prefix`, `:suffix`, `:infix`, `[*_]`, `[_*]`, or `[_*_]`).

Just as any occurrence of `:^+` inside a `!~` expression is interpreted as a parsetree constructor, inside an expression of the form `?(:^+...)` an occurrence of such a a subexpression is treated as a parsetree *de*constructor, that is, it is treated as though it also had a question mark in front of it.

Although a tidier is expected to return a reconstructed parsetree, it may instead return a defect or list of defects. A defect is an object produced by a call to the macro `defect`. Its syntax is identical to the `out` construct of YTools. `(defect --`*aaa*`--)` constructs a "defect" object whose printed representation is `(out --`*aaa*`-)`. For instance, `(defect "Non-number: " n)`, when `n` = `five`, is an object whose printed representation is `#<Defect: Non-number: five>`. If a tidier returns any defects instead of a new parsetree, the old parsetree is labeled with those defects (which will be displayed by `parsetree-show`) (appendix B).

One can specify a tidier as an argument to `def-syntactic`, as shown above, or it can be done separately using the macro `def-tidier`:

```
(def-tidier toknane [:grammar gramname]
     [ :tidier fcn-form
       | fcn-name | lambda-form
       | (...match-clause+...)
       | (definer [:^] ---definition---) ])
```

where

definer is an operation such as `defun` for defining a function

(The other syntactic categories here are as in the definition of the `:tidier` argument to `def-syntactic`.)

The syntax of `def-tidier` is close to that of the `:tidier` argument to `def-syntactic`, with a couple of variations. One is that supplying a form that evaluates to an arbitrary function must be done with an explicit `:tidier` argument. Other forms are interpreted as being of the form `(`*definer* `:^...)`, just as for the `datafun` macro of YTools. A function is defined whose name is

k`-tidier/`*g*

where *k* is the *tokname* and *g* is the name of the grammar supplied either by a `:grammar` argument to `def-tidier` or by an enclosing `(with-grammar` *gramname* `...)`. This function then becomes the tidier for *k* in *g*.

Finally, if all this syntactic sugar seems too cloying, the function `(define-tidier` *k f* `[:grammar` *g*`])` can be used to make function *f* the tidier associated with the token of name *k* in grammar *g*, or the grammar declared by an enclosing `with-grammar`.

After all this explanation, I can finally present the tidier for quantifiers, a task postponed since section 3.2, page 19. The tidier just discards the structure separating the quantifier from the variables it binds, and makes the operator of any quantified expression be the new operator `<quant>`.

```
<<Define quant-tidier
   (defun quant-tidier (ptree _)
      (match-cond ptree
        (:?  ?(:^+ ?qtfier (:^+ comma ?@vars))
           !~(:^+ quant ,qtfier ,@vars))
        (:?  ?(:^+ ?qtfier (:^+ group (:^+ comma ?@vars)))
           !~(:^+ quant ,qtfier ,@vars))
        (:?  ?(:^+ ?qtfier ?v)
           !~(:^+ quant ,qtfier ,v))
        (:else (defect "Bogus quantifier structure")))) >>
```

(The underscore used as the second argument to `quant-tidier` indicates that that argument is to be ignored.)

3.4 Checking

Once a complete, tidied parsetree has been produced by the parser, it must be "checked." The operator at each node of the tree is associated with a list of *checkers*, each a procedure that returns a list of defects. Checking the tree means, for every subtree, running the checkers associated with its operator, and adding any defects produced to the defect list of the subtree. Checkers are run bottom-up, but they don't change the structure of the parsetree, so the only consequence of this order is that a checker can see the defects on the subtrees of the parsetree node it is checking.

A *checker*, is a procedure that takes the same two arguments as a tidier, a parsetree and a grammar, and returns a list of defects, hopefully empty. (As before, a single defect is converted to a list of defects.) One way to declare checkers is by using the `:checkers` argument of `def-syntactic`.

The syntax of the `:checkers` argument to `def-syntactic` is a list of *checkers-spec*s, described thus:

```
:checkers
   ( [ check-clause | (:up level check-clause*) ]*)
```

```
check-clause ::= fcn-name | lambda-form
               | fcn-form
               | match-clause
```

The syntactic constructs invoked here are just as for the `:tidier` argument.

The `:checkers` argument represents a list of $\langle l, c\rangle$ pairs, where c is a checker, and l is a *level*. Without the explicit `:up`, the level defaults to 0. For now, I'll focus on the level-0 case.

The checker c can be specified by name, by explicit `lambda`, or as a form that evaluates to it; or, in many cases, by a `match-cond` clause. A list of *match-clauses* `(:?` p e`)` is interpreted as a list of checkers, each of the form

```
(lambda (this-ptree this-grammar)
   (match-cond this-ptree ...
      (:?  p e)
      (t '())))
```

As with tidiers, the variables `this-ptree` and `this-grammar` may be used by the grammar writer to refer to the current parsetree and grammar.

However, a *match-clause* without the e part is interpreted as a pattern that must match the parsetree. That is, `(:?` p`)` is interpreted as

```
(lambda (this-ptree this-grammar)
   (match-cond this-ptree ...
      (:?  ?(?(:~ p) (defect "Not of form " p)))
      t '()))
```

Even though a list of checkers and a tidier can both be specified as a list of *match-clauses*, the meanings are different. To minimize confusion, one should put blank lines between checker clauses and omit them between tidier clauses, thus emphasizing that the former are lists of checker specifications, while the latter specifies a single tidier.

For example, the Hobbs grammar allows symbols as functions, but nothing more complex. That is, any parenthesized expression that is tidied into a `fun-app` must have a symbol as function. We don't attach this checker to `<left-paren>`, because the tidier for `<left-paren>` substituted the token `<fun-app>`:

```
<<Define left-paren-checkers
   (def-checkers fun-app
         (:?  ?(:^+ fun-app ?(:~ ?(:+ ?name is-Symbol)) ?@_)
            (defect "Function call with illegal function " name))) >>
```

The fact that the entire parsetree has been tidied before the checkers run allows one to think about the final expected pattern of trees and subtrees, and to neglect all the various intermediate forms that may have existed while tidying was in progress.

Analogously to `def-tidier`, there is a `def-checkers` macro that enables separate declaration of a token's checkers:

```
(def-checkers [:grammar gramname]
     [ check-clause | (:up level check-clause) ]*)
```

Except for the optional `:grammar` argument, its syntax is exactly the same as the `:checkers` argument to `def-syntactic`. There is also a function `(define-checkers` *k l* `[:grammar` *g*`])`, which finds the token with name *k* in the appropriate grammar (*g* or the grammar supplied by an enclosing `with-grammar`), and makes *l* into its checkers list. *l* should, of course, be a list of ⟨*level*, *checker*⟩ pairs.

No matter how the list of checkers is defined, it's important to realize that when a parsetree is to be checked, the parser runs, not just the checkers associated with its token in the current grammar, but also those associated with it in the grammar's `syn-parent`, the `syn-parent`'s `syn-parent`, and so forth. See section 4.1.

So far I have described only the behavior of level-0 checkers. A level-0 checker for token *tok* is applied to every parsetree whose operator is *tok*. A level-1 checker is applied to every parsetree one of whose subtrees has operator *tok*. I can't think of any reason to use levels higher than 1, but in general a level-i checker is applied to every parsetree that has a *tok*-headed sub^{i}tree. The defects produced are associated with the sub^{i}tree.

Along with the tidier for quantified expressions, presented at the end of section 3.3, we have a couple of checkers that make sure that the variables being bound are legal and that the body is a single expression: Because the tidier gave the operator `<quant>` to all quantified expressions, the checkers must be associated with that lexeme:

```
<<Define quant-checker
  (def-checkers quant
     (:?  ?(:^+ quant ?_ ?vars ?@_)
        (cond ((atom vars)
               (defect "Illegal bound variables in quantified expression "
                       vars))
              ((exists (v :in vars)
                  (not (is-Symbol v)))
               (defect "Quantified expression has illegal (non-Symbol)"
                      " variables:  " vars))))

     (:?  ?(:^+ quant ?_ ?_ ?body ?junk-h ?@junk-t)
        (defect "Quantified expression has too many expressions in"
        " body:  " `(,body ,junk-h ,@junk-t)))) >>
```

We can now specify the syntax of left-bracket in the Hobbs grammar, which is similar to that of left-paren, except that brackets allow quantifiers to their left.

```
<<Define left-bracket
   (def-syntactic left-bracket
                  :prefix (:precedence 0 :context (:open nil right-bracket))
                  :infix (:left-precedence 40 :right-precedence 0
                          :context (:open nil right-bracket))
      :tidier
         ((:?  ?(:^+ left-bracket :fixity :prefix ?@subs)
             !~(:^+ group ,@subs))
          (:?  ?(:^+ left-bracket :fixity :infix
                      ?(:^+ group
                            ?(:^+ ?(:& ?qtfier ?(:|| A E E!))
                                  ?vars))
                      ?@body)
             (match-cond vars
                 (:?  ?(:^+ comma ?@vl)
                    !~(:^+ ,qtfier ,vl ,@body))
                 (:else !~(:^+ ,qtfier ,vars ,@body))))
          (:else
             (let ((d (defect "Ill-formed quantified expression "
                              :% " Quantifier:  " qtfier " Vars:  " vars
                              :% " Body:  " body)))
                (dbg-save this-ptree)
                (breakpoint left-bracket-tidier
                   "Defect:  " d
                   :% " for ptree " this-ptree)
                d))))

   (def-checkers quant
       (:?  ?(:^+ quant ?_ ?_ ?body ?junk-h ?@junk-t)
          (defect "Quantified expression has too many expressions in"
                  " body:  " `(,body ,junk-h ,@junk-t)))

       (:?  ?(:^+ quant ?_
                   ?(:~ ?(:+ ?vars (\\ (vl) (<& is-Symbol vl))))
                   ?_)
```

```
            (defect "Quantified expression has illegal variables:  "
                    vars))) >>
```

(The notation `\\` is short for `lambda`.)

As I said, tidying and checking proceed bottom-up, so that the last node checked is the one at the very top of the parsetree. In addition to all the other tests, it is checked against the `:top-tokens` argument of `def-grammar`, which is of the form

```
    :top-tokens
        :all | (tok_1 ...tok_k)
        | (:all-but tok_1 ...tok_k)
```

If the value is `:all` (the default), then any symbol can appear at the top of the tree. If it is a list of token names, then only those symbols can appear. If it is of the form `(:all-but ...)`, then any symbol except the explicitly enumerated ones can appear at the top of the tree.

For the Hobbs grammar, we require top-level expressions to be either predications or quantified statements, by writing

```
<<Define hobbs-top-tokens
:.(def-grammar hobbsgram .:
                :top-tokens (fun-app quant)) >>
```

But I've left out one tricky issue. Suppose an operator has a level-1 checker, and it appears at the top of the tree. Do we just ignore the level-1 (and level-2, etc.) checkers? No. If there is a level-i checker for a top-level operator, then Lexiparse constructs a temporary, artificial "supertree" above the actual top node. This supertree has one subtree (the actual top node), and the operator `<^>`. Level-1 checkers for the actual top operator are run on this tree. If there are checkers at higher levels, the process is repeated for as many layers as are necessary.

Earlier I alluded to a hypothetical language in which all top-level expressions had to be of the form "`define ....`" We can make sure no other expressions can appear there by setting the `:top-tokens` field of the grammar to `(define)`. But suppose we want to forbid `define` from appearing in any other context. We can write the checker:

```
     (def-checkers define
        (:up 1 (:? ?(:~ (:^+ ^ ?@_))
                  (defect "Operator 'define' appears"
                          " below top of tree"))))
```

You don't have to define the token `<^>`, and in fact that would be a bad idea.
Here are a few built-in utilities that are useful in writing checkers:

- `(check-all` *v l g* `---`*checkers*`---)`: For each element *v* in list *l*, run all the *checkers* and return all the defects they produce. *g* is the local grammar, often not important. For example, to verify that all immediate subtrees of the current tree have an arithmetic operator, you can write the checker:

```
(:? ?(:^+ ?op ?@subs)
   (check-all sub subs this-grammar
        (:? ?(:^+ ?(:|| + - * /) ?@_))))
```

- `(occ-range` *trees min max pattern* `&key (lo-defect` d_{lo}`) (hi-defect` d_{hi}`))`: Given a list of parsetrees *trees*, produce a defect if the subset matching *pattern* is less than *min* or greater than *max*. To indicate "no upper bound," use `:infty` for *max*. The expression d_{lo} will be evaluated to yield the description of a defect in the case where the size is below *min*; and similarly for d_{hi} wrt *max*. The default values for `:lo-defect` and `:hi-defect` are statements that the numbers of elements matching the pattern are too high or too low.

- `(check-ancestors ---`*checkers*`---)` Run each checker on every ancestor of the current parsetree. Return all the defects they produce.

3.5 Local Grammars

Sometimes the syntax expected in one corner of a grammar is radically different from the syntax expected elsewhere. For example, suppose we want to parse a language in which variable declarations resemble this example:

```
(x, y - Integer name - String)
```

Clearly, `minus` has lower precedence than `comma`, so you get tree structures such as

```
(:^+ \| (:^+ minus (:^+ comma x y))
        (:^+ minus name))
```

But the arithmetic expression `f(x-y, z)` must parse as

```
(:^+ left-paren f (:^+ comma (:^+ minus x y)
                             name))
```

which would be tidied to

```
(:^+ fun-app f (:^+ minus x y)
               name)
```

if `comma` is the "separator" for `left-paren`. (See section 3.1.) Clearly, in arithmetic expressions `minus` has a higher precedence than `comma`.

The solution is to allow syntactic tokens to be associated with *local grammars* that prescribe different rules from normal. Suppose that our language allows variable declarations as the second argument to `define`, as in

```
define foo (x,y - Integer s - String) ...
```

and suppose that the grammar for variable declarations is named `var-decls`. We can declare `define` thus:

```
(def-syntactic define :reserved t
                      :prefix (:numargs 3
                               :local-grammar ((2 var-decls))))
```

The `:local-grammar` argument for a prefix operator is a list of *local-gram-specs*, each a pair (i g), where i is a legal argument position and g is a grammar name. As usual, if there's just one such spec, the outer layer of parens can be omitted. If $i = 1$, you can just write g.

For an infix operator, the `:local-grammar` argument, if present, is just the name of the grammar that should govern the syntax of the operator's second argument. (Lexiparse has an obvious left-to-right bias, and must have already parsed the first argument before the operator is seen.)

A local grammar is usually a subgrammar of the current grammar, that is, it inherits most of its definitions from the current grammar. The reason for and consequences of this design pattern are described in section 4.1.

4 Internalization and Inheritance

Internalization is the process of converting a parsetree into an application-dependent object. There is basically one way to do this, namely, to run a recursive procedure on parsetrees. Such a procedure need not have anything to do with grammars as such. Just write it, using the API for the `Parsetree` datatype described in section 3.

However, occasionally it makes sense to organize the internalization process using grammars. The hierarchy of grammars is a way of keeping track of different kinds of internalization associated with the same language. For instance, one might write three XML grammars with increasing degrees of validation:

1. The base grammar defines the syntax of XML without any internalization. It checks that brackets match, but not much else.

2. The next highest level checks the structure against a DTD — in other words, it implements a "validating" XML parser.

3. The next level converts XML to a data structure implementing the Document Object Model [Con04].

I'll explain how to do this below (sect. 4.1).

The *internalizer-spec* that is the value of the `:internalizer` argument to `def-syntactic` is any form that evaluates to a function of two or more arguments. If you call (`internalize` p g a_1 $\ldots a_k$), where p is a parsetree and g is a grammar, the internalizer associated with the operator of p in g will run, with the same arguments `internalize` was given. The function `internalize` does *not* try to internalize the subtrees of p; for that to happen, the internalizer must call `internalize` recursively. In this sense, internalization is "top-down."

For the Hobbs grammar, the built-in internalizers convert parsetrees to a Lisp-style notation, as used in Kif [GF94] and PDDL [McD98]. We start with the infix operator "v", used to indicate disjunction:

```
<<Define hobbs-internalizer
<<Insert:  functional-term-def (p. 33)>>

(with-grammar hobbsgram

   (def-internal v
       :internalizer
       (make-functional-term 'or))

   <<Insert:  many-other-functional-terms (p. 34)>>

   <<Insert:  hobbs-quantifiers (p. 34)>>
   <<Insert:  hobbs-parens (p. 35)>>
)>>
```

Because internalizers are relatively independent of the rest of the grammar, it often seems appropriate to use `def-internal` to define them instead of using the `:internal` argument to `def-syntactic`. For both contexts, the argument is just a term whose value is a function. In addition, the `def-internal` macro has a couple of other options:

```
(def-internal tokname [:grammar gramname]
    [ :internalizer fcn-form
      | fcn-name | lambda-form
      | (definer [:^] ---definition---) ])
```

where *definer* and the other syntactic variables are as for `def-tidier` and `def-checkers`.

In the case of "v", the `:internalizer` is a call to this function:

```
<<Define functional-term-def
;;;This function does the work for most of the internalizers.
;;; It just gloms the appropriate function onto the internalized args.  --
(defun make-functional-term (fcn)
   (\\ (pt g)
      `(,fcn ,@(mapcar (\\ (sub) (internalize sub g))
                       (Parsetree-subtrees pt))))) >>
```

`(make-functional-term 'or)` is an internalizer that tacks the symbol `or` onto the front of the internalized subtrees of the parsetree it is internalizing. So a parsetree with structure:

```
(:^+ or (a (:^+ and b c)))
```

will be converted to the list structure:

(or $I_{\mathtt{a}}$ $I_{\mathtt{(:\hat{}+ and b c)}}$)

where $I_{\mathtt{a}}$ and $I_{\mathtt{(:\hat{}+ and b c)}}$ are the internalizations of the two subtrees. I'm sure I'm not giving away any surprise ending when I reveal that the process eventually ends with the result

```
(or a (and b c))
```

As with tidiers and checkers, there is a straightforward function for defining internalizers:

(define-internalizer k f [:grammar g])

that associates f with the token named k as its internalizer. The optional `:grammar` argument works just as for `define-tidier` and `define-checkers`.

In this simple system, the internalizer takes no other arguments except the parsetree and the grammar. This internalizer does no error checking at all. I'll say more below about how it might have been made hairier, at least with regard to checking for unbound variables.

Most of the other tokens in the grammar are variations on the same theme, so much so that we can write a loop to define all their internalizers:

```
<<Define many-other-functional-terms
   (repeat :for ((tok+fcn :in '((and and) (not not) (greater >) (less <)
                                (geq >=) (leq =<) (equals =) (plus +)
                                (minus -) (times *) (divide /)
                                (right-arrow if) (double-arrow iff))))
      (define-internalizer
         (first tok+fcn)
         (make-functional-term (second tok+fcn))))
>>
```

The quantifiers are equally repetitive:

```
<<Define hobbs-quantifiers
   (repeat :for ((tok+quant :in '((A forall) (E exists) (E! exists!))))
      (define-internalizer
          (first tok+quant)
          (make-quant-internalizer (second tok+quant)))
    :where
      (:def make-quant-internalizer (q)
         (\\ (pt gram)
           `(,q ,@(list-internalize (Parsetree-subtrees pt) gram)))))

>>
```

The convenience function `(list-internalize` l g`)` returns a list of the internalized versions of the elements of l, all internalized with respect to the grammar g. (Here I use the `repeat` construct from [McD03], a less cluttered approach to iteration than the usual Lisp `loop`. The `:where-:def` clauses allows you to bind functions whose scope is the loop body.)

For a more realistic application, one would want to make sure that all variables that occur in a formula are bound by a quantifier. To do that, one would add an extra argument to the internalizers, call it `boundvars`, which, when processing expression E, would be a list of all the variables bound by quantifiers including E in their scope. Most internalizers would just pass the `boundvars` down, but the `quant` internalizers would add to it, and the internalizer for `fun-app` would check that any symbol occurring as an argument was an element of `boundvars`.

The internalizers for the brackets should be easy to understand at this point. Left braces denote sets; `{x, y, z}` → `(set x y z)`. A left paren gets handled differently depending on its fixity; by this point prefix parens have been renamed `group`, and infix parens renamed `fun-app`.

```
<<Define hobbs-parens
   (def-internalizer left-brace
       :internalizer
     (\\ (lbpt gram)
        `(set ,@(list-internalize (Parsetree-subtrees lbpt) gram))))

   (def-internal group
      (defun :^ (lppt g)
         (let ((subs (Parsetree-subtrees lppt)))
            (cond ((= (length subs) 1)
                   (internalize (first subs) g))
                  (t
                   (list-internalize subs g))))))

   (def-internal fun-app
      (defun :^ (lppt g)
         (let ((pre (first (Parsetree-subtrees lppt)))
               (args (list-internalize (rest (Parsetree-subtrees lppt)) g)))
            (cond ((is-Parsetree pre)
                   (let ((q (internalize pre g)))
                      (match-cond q
                         (:?  (?(:|| forall exists exists!  :& ?quant) ?vars)
                            `(,quant ,vars ,@args))
                         (t
                          `(,q ,@args)))))
                  (t `(,pre ,@args))))))

   (def-internal quant
      (defun :^ (qpt _)
         (match-cond qpt
            (:?  ?(:^+ quant ?qtfier ?vars ?body)
               `(,qtfier ,vars ,body))
            (:else (signal-problem quant-internalizer
                      "Malformed quantified parsetree fell through cracks:  "
                      qpt))))) >>
```

4.1 Inheritance

A grammar may inherit definitions from another grammar. In fact, you can control the inheritance of lexical, syntactic, and internalizing definitions from different parent grammars. In the `def-grammar` macro (section 1), there are three keyword arguments `:lex-parent`, `:syn-parent`, and `:int-parent`, each of which is a grammar name. (The `:parent` argument can be used to avoid writing a parent-grammar name more than once.)

Tokens are defined by name. If a parent grammar has a token `foo`, and the child grammar defines a token `foo`, their properties are merged when possible. The merge rules are as follows:

- For internalizers: The child internalizer shadows an internalizer for the same token in the parent, but within the child you can call

  ```
  (call-parent-internalizer tok tree gram)
  ```

  to let the shadowed internalizer run. It returns an internal data structure that can then be further elaborated by the child internalizer. Above I mentioned that one could build XML parsers with varying levels of validation. One way to do that would be to have the internalizers for the more stringent versions call the parent internalizers first, then follow up with further validations.

- For fixity information, specified by `:prefix`, `:infix`, or `:suffix` fields: if any of these is supplied for the token in the child grammar, then all grouping information in the parent is shadowed. If this information is missing in the child, then the inheritance pattern is more interesting. In this case, the fixity information is inherited in its entirety. The presence of a tidier in the child hides only the parent's tidier; and checkers specified for the child are *merged* with the checkers for the parent. (See below)

- For lexical definitions: Any specification of the lexical properties of a character in the child shadows whatever the parent has to say about it.

Checkers inherited from a syntactic parent of grammar G run before the checkers declared at G. However, since they all get run eventually, the only visible effect of this ordering is the order in which defects are listed, if there is more than one.

5 Getting, Installing, and Running Lexiparse

The Lexiparse system can be downloaded from my website, at

`http://www.cs.yale.edu/homes/dvm#software`.

You must first download the YTools package and install it. Then download the compressed tar file for Lexiparse. Uncompress and unpack the tar file to a convenient directory.

As explained at `http://www.cs.yale.edu/homes/dvm#ytsetup`, YTools requires that you bind two Lisp global variables, presumably in your Lisp initialization file. One of them, `ytools::ytload-directory*`, is the name of a directory containing the `.lmd` files that tell YTools how to install and load a software package. Find the file `lexiparse.lmd` in the Lexiparse directory, and move it to the ytload directory. Then execute `(yt-install :lexiparse)`. You will be asked a couple of questions about where to find the system and where to put binary files, and that will be that.

On future occasions, all you need to do is start Lisp and execute `(yt-load :lexiparse)`.

Once you've installed the system, there are various ways to use it. Suppose you've produced a grammar for a language. The grammar is an ordinary Lisp file, so you can compile and load it as you would any other Lisp file. The `(def-grammar` *N*`)` macro (section 1) defines as *N* as a global variable whose value is the grammar. It is also accessible as the value of `(grammar-with-name '`*N*`)`.

Once the grammar is loaded, it can be used by the following entry points to the parser:

`(string-syn-gen` *`string grammar`*`)` Given a string and a grammar, produce a generator of the parsetrees for the well-formed expressions in the *string*.

`(file-syn-gen` *`filename grammar`*`)` Like `string-syn-gen`, but it uses the contents of the given file as the string to parse.

`(parsetree-gen-show` *`parsetree-generator`*`)` Generate the parsetrees generated by *parsetree-generator*, and return a list of them. As they are produced, display them using `parsetree-show` (see appendix B).

`(chargen-to-stream` *`character-gen srm gram int`* `:int-output-fcn` *`iofcn`*`)` takes a character generator *character-gen* , an output stream *srm*, and a grammar *gram*, and writes to *srm* the entities parsed from *character-gen*. If the *int* argument is true, the entities are internalized and the results are written to the stream. If the keyword argument `:int-output-fcn` is supplied, it must be a function of two arguments (the second a stream), that is used to write the objects produced by the parsing process.

`(file-translate` *`in-file out-file gram int`* `:int-output-fcn` *`iofcn`*`)` reads and parses the contents of file *in-file*, and sends the results to *out-file*. All the other arguments are the same as those to `chargen-to-stream`.

`(parse1` *`string grammar`*`)` Extract from *string* and return a parsetree headed by one of the top-nodes of *grammar*. The parsetree need not cover the entire string, just some prefix of it.

`(string-parse-and-internalize` *`string grammar`*`)` is like `parse1`, except that it internalizes the resulting parsetree.

Besides the defects found by the tidiers and checkers in a grammar, the parser itself will attach defects to parsetrees. For example,

```
(parsetree-show (parse1 "p(aaa + * bbb)" hobbsgram))
```

will print

```
<fun-app> [*_]
   P
   <plus> [_*_]
      AAA
      <*> [_*_]
         nil
      #<Defect Operator <*> not allowed as prefix>
      BBB
```

If the global variable `break-on-defect*` is set to true, then the system will enter the debugger whenever a defect is attached to a parsetree.

6 Suggestions for Using Lexiparse

A parser written using Lexiparse doesn't look like the standard set of BNF-style productions as would be supplied to Yacc or a similar system. You need a little practice to learn how to read one. It helps if it was written in a clear way. Here are some hints on how to write readable grammars. First, be sure you're clear about how precedences work: The higher the precedence of an operator, the lower down it will appear in a parsetree (if the tree is drawn in the usual way, with the root at the top). Precedences are like the binding strengths among elementary particles. Tight-binding quarks are at the lowest level of an atom's organization, the loosely bound electrons are at the outermost level, and molecules are held together by even weaker forces.

In a traditional grammar, productions are usually written in a "top-down" way, starting with `S`→ ... and ending with `variable`→`name.name` or the like. To achieve the same effect in a precedence-based grammar, you put `S` in the `:top-tokens` list, then define it and other symbols with low precedence first. In this manual, I've arranged the pieces of the grammar in an abnormal order for expository purposes. It's probably worth comparing the complete grammar in its fully assembled order, which may be found at `http://www.cs.yale.edu/homes/dvm/papers/hobbsgram.lisp`.

Precedences are only the beginning of writing a Lexiparse grammar. Most of the action comes in the pattern checking and transformation done by the tidiers, checkers, and internalizers. These procedures operate on syntax trees using pattern matching, tree building, and occasional Lisp code. So the grammar ends up "feeling like" a set of Lisp macros. (Or perhaps Dylan is a nearer cousin [FEMW97].) The classic problem in macro writing is how the macro extracts the items it operates on from the surrounding text. Most solutions are quite stultifying.[8] In Lisp and Scheme the extraction is done using the ubiquitous parentheses, which spares the macro writer a lot of work. In Dylan, a flexible pattern language is superimposed on the underlying syntax; exactly how is not clear, but the idea is that if the pattern matches a piece of code, that piece of code is treated as a constituent in the parse tree. In any of these languages, the macro then rewrites the extracted subtree into a (more nearly[9]) executable code tree. The transformations are usually expressible as matching/rewrite rules, but, in Lisp at least, one has access to the entire language if necessary.

Hence, in reading a Lexiparse grammar, if the grammar writer has taken care to make explicit what context and pieces are legal for a tree headed by a given token, one good way to develop an overall picture of what the language looks like is to read the tidiers and checkers first, consulting the precedences only to verify that the subtrees the tidiers and checkers expect

[8]The "preprocessor" in C-like languages is typical; it allows you to define only constructs that look like function calls, and it provides string substitution as the only operation you can perform on the arguments.

[9]The output subtree and its parts are themselves subject to further macro expansion.

do indeed have operators with higher precedences than the current tree's operator.

In writing a Lexiparse grammar, one has a great deal of control over what to put in tidiers, checkers, and internalizers. One can put everything in the internalizer for a token, and generate Lisp errors in response to syntactic anomalies. But for someone reading the grammar, it makes more sense to put the same information into the checkers for the token. For one thing, each nugget of information is clearly displayed. For another, the defects produced by the checkers are attached to the parsetree being built in a perspicuous way.

Another issue is what precedences to assign. Lexiparse doesn't care, so long as they're (non-complex) numbers. But for clarity, here are some guidelines:

1. Precedences should be integers, to take advantage of the natural scaling effect imposed by the fact that integers can't be indefinitely magnified. The precedences 45 and 47 will be noted by someone reading the grammar as being "close together" — if all precedences are integers. But if the reader comes across a precedence of 46.3, all bets are off.

2. The natural lower bound on precedences is 0. As discussed above, this is the recommended precedence for the interior of brackets.

3. The `standard-arith` grammar is included in the distribution of Lexiparse. It defines lexemes for all the standard arithmetic and logical operators, with their usual precedences. The lowest precedence (except for the zeroes assigned to the interiors of parentheses) is the 100 assigned to `<comma>`. The highest, 200, is the left precedence of `<left-paren>`. The precedences of all the operators are evenly arranged between these two bounds, as shown in table 1.

You can declare your grammar to have `standard-arith` as a `syn-parent`. There's plenty of room to interleave your own operators between those provided by `standard-arith`. (You can also override the behavior of any operator, as explained in section 4.1.) The 200 level is not a ceiling; if you want `x.y(a, b)` to be parsed so that the left parenthesis "dominates" the expression `x.y`, i.e., appears above it in the parsetree, then create a token `<dot>` with lexical representation "`.`", and assign it precedence 210.

Finally, be prepared to get creative in your attempt to capture a syntax using precedence alone. Consider, for instance, the indispensable `if-then-else` statement. If the `else` is to be optional, one might want the syntax of `if` or

Token	*Character(s)*	*Precedence*
`<comma>`	`,`	100
`<or>`		120
`<and>`		130
`<not>`		140
`<greater>`	`>`	160
`<less>`	`<`	160
`<geq>`	`>=`	160
`<leq>`	`=<`	160
`<equals>`	`=`	160
`<plus>`	`+`	180
`<minus>`	`-`	180
`<times>`	`*`	190
`<divide>`	`/`	190

"*Character*" column is left blank if token is a reserved word.

Table 1: Precedences of the `standard-arith` grammar

`then` to check to see if it is there. Unfortunately, there is no way to do that using Lexiparse. Instead, you must make `else` an infix operator, so that `if` *a* `then` *b* `else` *c* gets parsed as

```
(:^+ if a (:^+ then (:^+ else b c)))
```

This makes no semantic sense, of course, but the tidier for `if` can fix it all up:

```
(def-tidier if
    ((:? ?(:^+ if ?test ?(:^+ then ?(:^+ else ?iftrue ?iffalse)))
        !~(:^+ if ,test ,iftrue ,iffalse))
     (:?  ?(:^+ if ?test ?(:^+ then ?iftrue))
        !~(:^+ if ,test ,iftrue null))
     (:else (defect "'if' " test " not followed by 'then"))))
```

Future work: Currently Lexiparse doesn't do a good job of keeping track of the context of syntactic errors. Its error messages can't tell the user which line of the input the error occurred. That could be fixed fairly easily.

Currently infix operators are allowed to have either no arguments to their right, or exactly one. The former class is just what I have been calling "suffix" operators. In fact, there are really only two kinds of operator, those that take no arguments to their left (prefix ops), and those that take one (infix ops). So there is a hole in the formalism: the case where an infix operator takes more than one argument to its right. This would be fairly easy to fix if there is a need.

Any grammar that does internalization should have an optional "leaf internalizer" that would be applied to the leaves of all parsetrees.

The number lexifier `lex-number` (sect. 2) doesn't recognize "scientific" notation such as `1.3E-9`. This should be fixed.

Limitations: Not all languages can be represented using a Lexiparse grammar, at least not in a legible way. It does best with languages whose major constructs are identified by a prefixed reserved word, such as `define`. Its left-right bias means that a language with suffix operators taking more than one argument is hard to handle. An expression such as "`x 5 + y *`" in such a language should be parsed as `(:^+ * (:^+ + x 5) y)`. But Lexiparse will be confused as soon as it sees `x` and `5` together. One would have to use the contiguity operator to allow the suffix operators `+` and `*` to take exactly one argument, and parse the expression as if it had the structure

```
   = (:^+ * (:^+ \| (:^+ + (:^+  \| x 5))
                   y))
```

Perhaps there is a way to implement infix ops that take more than one argument to their left. If so, then all operators would be characterized by how many arguments, zero or more, that they expected on their left and on their right.

Even more problematic are languages in which the precedence of operators can change at compile time, such as ML and Prolog. I can think of ways of handling this issue, but they all involve breaking through the abstractions provided by Lexiparse and performing unnatural acts on its internal organs.

Providing a Lexiparse grammar for the C family of languages (which includes C, C++, and Java) would be an interesting exercise, made challenging by the absence of reserved words in crucial places, such as declarators.

I doubt that Perl can be parsed using a Lexiparse grammar, but that's just a corollary of the general theorem that Perl can't be parsed using *any* grammar.

A Literate Programming

Literate programming is a documentation technique invented by Knuth [Knu84] in which the usual code-commenting conventions are turned inside out.

Rather than presenting a program + comments, one presents an essay + code. The essay appears in whatever expository order makes the most sense, and pieces of the code are exhibited in the essay at the point where it makes sense to discuss them. The entire program eventually appears, so that it can be extracted by a piece of software called the *tangler* into a compilable form.

A piece of code in the text is called a code *segment*. Inside a segment, wherever you see

<<*Insert: f*>>

you are to realize that to complete the code requires finding the segment with the name *f* and substituting it here. The segment may be defined anywhere in this file, but will usually come later than the point where it is inserted. The definition looks like this:

<<*Define f*
Text of segment >>

At the top level of the segment hierarchy are *file segments*, each of which specifies the skeleton of the contents of a code file. These are defined by the following notation, instead of the usual "*Define*":

<<*Define* `File` *filename*
Text of segment >>

Don't try too hard to reconstruct the entire grammar in your head by following such links. The point of literate programming is to allow you understand code segments in their "documentation context" instead of their "formal context." So try to understand what the segment means on its own terms. Any segments it points to will be dealt with later. If you want to see the output as a whole, look at the version output by the "tangler" that puts the program back into the form the compiler likes. (For the Hobbs grammar, this output is found in the file `hobbsgram.lisp`.)

Sometimes to make a segment intelligible we will duplicate a bit of its context. Any pieces of a segment that are surrounded by the characters "`:.`" on the left and "`.:`" on the right is to be interpreted as a "context establisher" that isn't "really there." For instance, if you see at one point

```
(foobalicious :left-frobboz (frobbistication)
              <<Insert right-foobal>>)
```

Then we might define the segment `right-foobal` thus:

```
<<Define:  right-foobal
 :.(foobalicious :left-frobboz ....:
                 :right-frobboz (rephlogistion))>>
```

The actual segment is just the string "`right-frobboz (rephlogistion))`," but we repeat the outer layer of parens to remind ourselves what the string is embedded in.

B Lisp Utilities for Use with Lexiparse

A Lexiparse grammar is essentially a set of Lisp data structures and programs. To some extent one can do business entirely in a Lisp subset dedicated entirely to the parsing task. But inevitably one will find oneself having to deal with bits of Lisp code here and there. At that point it would be helpful to know the built-in utilities for dealing with parsetrees and grammars. Here's a list, in no particular order:

- `(string-gen` S`)` is a generator of the characters in S.
- `(file-gen` *`pathname`*`)` is a generator of the characters in the file *pathname*.
- `(Parsetree-op-token` T G`)` returns the token in grammar G corresponding to the operator of parsetree T.
- `(Parsetree-opname` T`)` returns the name of the operator of T.
- `(->token` S g`)` returns the token in grammar g whose name is the symbol S.
- `(tokname` t`)` is the name of the token t.
- `(Parsetree-defects` T`)` returns the defects checkers have found for T.
- `(subtree-is-defective` T`)` tests whether T or any of its descendents has any defects. (It works for any parsetree, in spite of the word "subtree" in its name.)
- `(Parsetree-defective-subtrees` T`)` returns the defective subtrees of T, i.e., the subtrees that have defects or have descendents with defects.

- `(parsetree-show` T `&optional` s d`)` displays parsetree T as a hierarchy, with subtrees indented and printed on successive rows beneath their parent. s is the stream the output is directed to (default `*standard-output*`). d is the maximum depth to display to.
- `show-parsetree-fixity*`: global Boolean variable; if set to false, the fixity of a parsetree is not printed (even by `parsetree-show`).

References

[Con04] World-Wide-Web Consortium. *Document Object Model (DOM) Technical Reports*. 2004.

[FEMW97] Neal Feinberg, Sonya E.Keene, Robert O. Mathews, and P. Tucker Withington. *Dylan Programming*. Addison-Wesley, 1997.

[GF94] Michael R. Genesereth and Richard E. Fikes. *Knowledge Interchange Format Version 3.0 Reference Manual*. 1994.

[HFA$^+$02] Jerry R. Hobbs, George Ferguson, James Allen, Richard Fikes, Pat Hayes, Drew McDermott, Ian Niles, Adam Pease, Austin Tate, Mabry Tyson, and Richard Waldinger. *DAML Ontology of Time*. 2002.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[McD98] Drew McDermott. The planning domain definition language manual. Technical Report 1165, Yale Computer Science, 1998. (CVC Report 98-003) `/mcdermott/software/pddl.tar.gz`.

[McD03] Drew McDermott. *The YTools Manual*. 2003.

[Pra76] Vaughn Pratt. Semantical considerations on floyd-hoare logic. In *Proc. IEEE Symp. on Foundations of Comp. Sci*, volume 17, pages 102–121, 1976.