Lexer and Parser Generators in Scheme

Scott Owens Matthew Flatt Olin Shivers Benjamin McMullan
University of Utah Georgia Institute of Technology

Abstract

The implementation of a basic LEX-style lexer generator or YACC-style parser generator requires only textbook knowledge. The implementation of practical and useful generators that cooperate well with a specific language, however, requires more comprehensive design effort. We discuss the design of lexer and parser generators for Scheme, based on our experience building two systems. Our discussion mostly centers on the effect of Scheme syntax and macros on the designs, but we also cover various side topics, such as an often-overlooked DFA compilation algorithm.

1 Introduction

Most general-purpose programming systems include a lexer and parser generator modeled after the design of the LEX and YACC tools from UNIX. Scheme is no exception; several LEX- and YACC-style packages have been written for it. LEX and YACC are popular because they support declarative specification (with a domain-specific language), and they generate efficient lexers and parsers. Although other parsing techniques offer certain advantages, LEX- and YACC-style parsing remains popular in many settings. In this paper, we report on the design and implementation of LEX- and YACC-style parsers in Scheme. Scheme's support for extensible syntax makes LEX- and YACC-style tools particularly interesting.

- Syntax allows the DSL specifications to reside within the Scheme program and to cooperate with the programming environment. We can also lift Scheme's syntactic extensibility into the DSL, making it extensible too.
- Syntax supports code generation through a tower of languages. Breaking the translation from grammar specification to Scheme code into smaller steps yields a flexible and clean separation of concerns between the levels.

Additionally, lexer and parsers are examples of language embedding in general, so this paper also serves as an elaboration of the "little languages" idea [13].

Fifth Workshop on Scheme and Functional Programming. September 22, 2004, Snowbird, Utah, USA.

We base our discussion on two parsing tools, PLT and GT, and present specific design notes on both systems along with discussion on why certain ideas work well, and how these systems differ from others. The PLT system most clearly demonstrates the first point above. PLT's novel features include an extensible regular expression language for lexing and a close interaction with the DrScheme programming environment. The GT system most clearly illustrates the second point. In GT, a parser's context-free grammar is transformed through a target-language independent language and a push-down automaton language before reaching Scheme, with potential for optimization and debugging support along the way.

2 Background

We briefly describe the essential design of LEX and YACC and how it can fit into Scheme. We also discuss how the existing LEX- and YACC-like systems for Scheme fit into the language.

2.1 LEX and YACC

A text processor is constructed with LEX and YACC by specifying a lexer that converts a character stream into a stream of tokens with attached values, and by specifying a parser that converts the token/value stream into a parse tree according to a context-free grammar (CFG). Instead of returning the parse tree, the parser performs a single bottom-up computation pass over the parse tree (synthesized attribute evaluation) and returns the resulting value, often an abstract-syntax tree (AST). LEX generates a lexer from a regexp DSL, and YACC generates a parser from a CFG DSL.

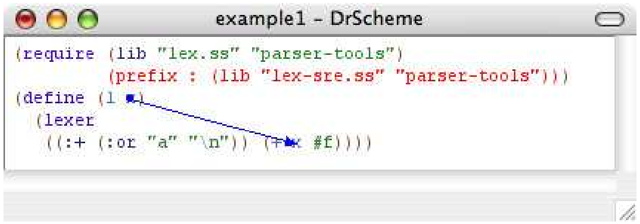
$$\begin{array}{ccccccc} \textit{regexps} \longrightarrow & \text{LEX} & & \text{YACC} & \longleftarrow \textit{CFG} \\ & \downarrow & & \downarrow & \\ \textit{chars} \longrightarrow & \text{lexer} & \stackrel{\textit{tokens}}{\longrightarrow} & \text{parser} & \longrightarrow \textit{AST} \end{array}$$

From a programmer's perspective, building a text processor takes three steps:

- Creation of regular expressions (regexps) that describe the token structure. These are typically defined outside the lexer with a regexp abbreviation facility.
- Creation of the lexer by pairing the regexps with code to generate tokens with values.
- Creation of the parser with a CFG that describes the syntax, using token names as non-terminals. Attribute evaluation code is directly attached to the CFG.

A lexer is occasionally useful apart from a parser and can choose to produce values other than special token structures. Similarly, a

Figure 1 Lexical scope in a lexer



parser can operate without a lexer or with a hand-written lexer that returns appropriate token structures. Nevertheless, LEX and YACC (and the lexers and parsers they generate) are used together in most cases.

Operationally, LEX converts the regexps into a deterministic finite automaton (DFA) and YACC converts the CFG into a push-down automaton (PDA). These conversions occur at lexer and parser generation time. The DFA can find a token in linear time in the length of the input stream, and the PDA can find the parse tree in linear time in the number of tokens. The transformation from regexps and CFG can be slow (exponential in the worst case), but performing these computations once, at compile time, supports deployment of fast text-processing applications.

2.2 The Scheme Way

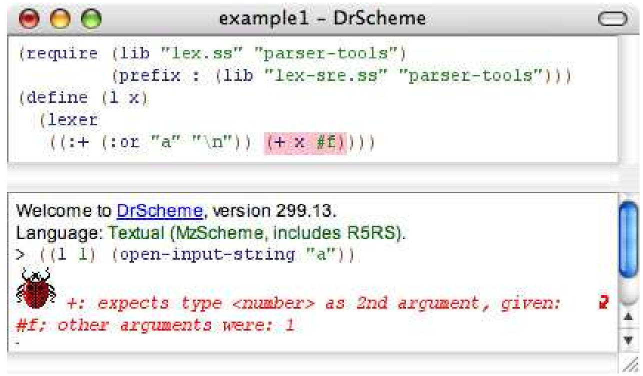
UNIX LEX and YACC operate in a file-processing batch mode. They read a lexer or parser specification from an input file and write a program to an output file. With batch compiled programming languages, *e.g.*, C or Java, this is the best that can be done. The build system (such as a makefile) is told how to convert the specification file into a code file, and it does so as needed during batch compilation.

The file-processing approach does not fit naturally into Scheme's compilation model. Instead of using an external batch compilation manager, most Scheme programs rely on a compilation strategy provided by the language implementation itself. The simplest way to cause these compilation managers to execute a Scheme program is to package it in a macro. The compilation manager then runs the program while it macro-expands the source. Specifically, when a lexer or parser generator is tied into the Scheme system via a macro, the macro expander invokes the regexp or grammar compiler when the internal compilation system decides it needs to. Each of the PLT and GT parser tools syntactically embeds the lexer or parser specification inside the Scheme program using lexer and parser macros. This solution easily supports LEX- and YACC- style pre-computation without breaking Scheme's compilation model.

With a macro-based approach, a lexer or parser specification can appear in any expression position. Hygiene then ensures that variables in embedded Scheme expressions refer to the lexically enclosing binding (see Figure 1). Furthermore, the embedded code automatically keeps source location information for error reporting, if the underlying macro expander tracks source locations as does PLT Scheme's (see Figure 2). A stand-alone specification achieves neither of these goals easily.

Source-location tracking and lexical scoping lets refactoring tools, such as DrScheme's Check Syntax, assist programmers with their

Figure 2 Source highlighting of a runtime error



parser and lexer actions. Check Syntax draws arrows between statically apparent binding and bound variable occurrences and can α-rename these variables. It also provides assistance with the declarative part of a parser specification, correlating non-terminals on the right and left of a production and correlating uses of `$n` attribute references with the referent terminal or non-terminal.

2.3 Previous Work

We categorize the many lexer and parser generators written in Scheme as follows: those that do not use s-expression syntax, those that use s-expressions but do not provide a syntactic form for integrating the specifications with Scheme programs, and those that do both.

Danny Dubé's SILex lexer generator [6] fits in the first category, closely duplicating the user interface of LEX. Mark Johnson's LALR parser generator [10] and the SLLGEN system [9] fall into the second category. Both provide a function that consumes a list structure representing a grammar and returns a parser. With Scheme's quasiquote mechanism, programmers can write the grammar in s-expression format and generate grammars at run time. This approach sacrifices the efficiency of computing the parser from the grammar at compile time and also hampers compile-time analysis of the CFG and attached code.

Dominique Boucher's original lalr-scm system [3] falls into the second category, encoding the CFG in an s-expression. It uses DeRemer and Pennello's LALR algorithm [5] to process the grammar and, unlike previous tools, supports ahead-of-time compilation of the CFG. However, it does not integrate specifications into Scheme via macros, instead it provides a compiler that maps CFG s-expressions to a parse table. The table is printed out and incorporated into source code along with an interpreter to drive the parser. In this sense, the parser development cycle resembles YACC's. Boucher's original implementation is also missing some important functional elements, such as associativity declarations to resolve shift/reduce ambiguities.

Design particulars aside, Boucher's source code was influential, and his complete, debugged, and portable LALR implementation provided a foundation for several later efforts in our third category. Serrano's Bigloo Scheme system [11] incorporated Boucher's implementation, with extensions. Critically, Bigloo uses macros to embed the parser language directly into Scheme. (Bigloo also supports embedded lexers.) Shivers and his students subsequently adopted Serrano's code at Georgia Tech for compiler work, and then massively rewrote the code (for example, removing global-variable dependencies and introducing record-based data structures for the ASTs) while implementing the GT parser design described in this paper. Boucher has addressed the above concerns, and the current lalr-scm system supplies a form for incorporating a CFG in a program.

Sperber and Thiemann's Essence parser generator [16] also falls into the third category, using an embedded s-expression based CFG definition form. Instead of a YACC-style CFG to PDA compilation step, Essence uses partial evaluation to specialize a generic LR parser to the given CFG. The partial evaluation technology removes the need for a separate compilation step, while ensuring performance comparable to the YACC methodology. Our use of macros lets us take a compilation-based approach to implementation—a simpler and less exotic technology for achieving performance.

3 Regular Expressions

Defining regular expressions that match the desired tokens is the first step in creating a lexer, but regexps are also used as a pattern language in many other text processing tools. For example, programming languages often support regular expression matching as a computational device over strings. Hence we first consider regexps by themselves.

3.1 Notational Convenience

Many regexp matching libraries use the POSIX regexp syntax embedded in a string. This approach requires the insertion of escape characters into the POSIX syntax (a veritable explosion of \ characters, since \ is used as the escape character in both POSIX and Scheme). An s-expression based regexp language fits more naturally into Scheme and can still include a form for string-embedded POSIX syntax, if desired.

SREs [14], Bigloo Regular Grammars [11, section 9], and PLT lexer regexps all use s-expression syntax. The SRE notation is oriented toward on-the-fly regexp matching functions and was developed for the scsh systems programming environment [12, 15]. Bigloo Regular Grammars are designed for lexer specification, as are the PLT lexer regexps. The PLT Scheme lexer generator uses the syntax of Figure 3,[1] and SREs use the syntax of Figure 4. Bigloo uses notation similar to SREs without the dynamic unquote operation.

3.2 Abstraction

To avoid unnecessary duplication of regular expressions, a regexp language should support abstraction over regular expressions. Consider the R^{5}RS specification of numbers:

$$\langle \texttt{uinteger}R\rangle \to \langle \texttt{digit}R\rangle^{+}\texttt{\#}^{*}$$

This example suggests naming a regexp, such as *digit8* for the digits `0` to `7`, and building a regexp producing function *uinteger* that takes in, for example, *digit8* and produces the regexp *uinteger8*.

[1] The regexp language described in Figure 3 is new to version 299.13 of PLT Scheme. The syntax of versions 20x does not support the definition of new forms and is incompatible with other common s-expression notations for regexps.

In systems that support runtime regexp generation, the abstraction power of Scheme can be applied to regexps. String-based regexps support run-time construction through direct string manipulation (*e.g.*, `string-append`). The SRE system provides constructors for SRE abstract syntax, allowing a Scheme expression to directly construct an arbitrary SRE. It also provides the `(rx sre ...)` form which contains s-expressions compiled according to the SRE syntax. Think of `rx` in analogy with `quasiquote`, but instead of building lists from s-expressions, it builds regexps from them. The `unquote` form in the SRE language returns to Scheme from SRE syntax. The Scheme expression's result must be a regexp value (produced either from the AST constructors, or another `rx` form). The R^{5}RS example in SRE notation follows.

```
(define (uinteger digit)
  (rx (: (+ ,digit) (* "#"))))
(define (number digit)
  (rx (: (? "-") ,(uinteger digit)
         (? ".") (? ,(uinteger digit)))))
(define digit2 (rx (| "0" "1")))
(define digit8 (rx (| "0" ... "7")))
(define number2 (number digit2))
(define number8 (number digit8))
```

A lexer cannot use the `unquote` approach, because a lexer must resolve its regexps at compile time while building the DFA. Thus, PLT and Bigloo support static regexp abstractions. In both systems, the regexp language supports static reference to a named regexp, but the association of a regexp with a name is handled outside of the regexp language. In Bigloo, named regexp appear in a special section of a lexer definition, as in LEX. This prevents sharing of regexps between lexers. In PLT, a Scheme form `define-lex-abbrevs` associates regexps with names. For example, consider the `define-lex-abbrevs` for a simplified *uinteger2*:

```
(define-lex-abbrevs
  (digit2 (union "0" "1"))
  (uinteger2 (repetition 1 +inf.0 digit2)))
```

Each name defined by `define-lex-abbrevs` obeys Scheme's lexical scoping and can be fully resolved at compile time. Thus, multiple PLT lexers can share a regexp.

To support the entire R^{5}RS example, the PLT system uses macros in the regexp language. A form, `define-lex-trans`, binds a transformer to a name that can appear in the operator position of a regexp. The regexp macro must return a regexp, as a Scheme macro must return a Scheme program. The system provides libraries of convenient regexp forms as syntactic sugar over the parsimonious built-in regexp syntax. Of course, programmers can define their own syntax if they prefer, creating regexp macros from any function that consumes and produces syntax objects [7][8, section 12.2] that represent regexps.

Using the SRE operator names, the R^{5}RS example becomes:

```
(define-lex-trans uinteger
  (syntax-rules ()
    ((_ digit) (: (+ digit) (* "#")))))
(define-lex-trans number
  (syntax-rules ()
    ((_ digit)
     (: (? "-") (uinteger digit)
        (? ".") (? (uinteger digit))))))
```

Figure 3 The PLT lexer regular-expression language

```
re ::= ident                              ; Bound regexp reference
     | string                             ; Constant
     | char                               ; Constant
     | (repetition lo hi re)              ; Repetition. hi=+inf.0 for infinity.
     | (union re ...)                     ; General set
     | (intersection re ...)              ;    algebra on
     | (complement re)                    ;    regexps
     | (concatenation re ...)             ; Sequencing
     | (char-range char char)             ; Character range
     | (char-complement re ...)           ; Character set complement, statically restricted to 1-char matches
     | (op form ...)                      ; Regexp macro
lo ::= natural number
hi ::= natural number or +inf.0
op ::= identifier
```

Figure 4 The SRE regular-expression language (some minor features elided)

```
re ::= string                ; Constant
     | char                  ; Constant
     | (** lo hi re ...)     ; Repetition. hi=#f for infinity.
     | (* re ...)            ; (** 0 #f re ...)
     | (+ re ...)            ; (** 1 #f re ...)
     | (? re ...)            ; (** 0 1 re ...)
     | (string)              ; Elements of string as char set
     | (: re ...)            ; Sequencing
     | (| re ...)            ; Union
     | (& re ...)            ; Intersection, complement, and difference
     | (~ re)                ;    statically restricted
     | (- re re ...)         ;    to 1-char matches
     | (/ char ...)          ; Pairs of chars form ranges
     | (submatch re ...)     ; Body is submatch
     | ,exp                  ; Scheme exp producing dynamic regexp
     | char-class            ; Fixed, predefined char set
lo ::= natural number
hi ::= natural number or #f
char-class ::= any | none | alphabetic | numeric | ...
```

```
(define-lex-abbrevs
  (digit2 (| "0" "1"))
  (digit8 (| "0" ... "7"))
  (number2 (number digit2))
  (number8 (number digit8)))
```

The `define-lex-trans` and `define-lex-abbrevs` forms are macro-generating macros that define each given name with a `define-syntax` form. The regexp parser uses `syntax-local-value` [8, section 12.6] to locates values for these names in the expansion environment. Unfortunately, many common regexp operator names, such as $+$ and $*$, conflict with built-in Scheme functions. Since Scheme has only one namespace, some care must be taken to avoid conflicts when importing a library of regexp operators, *e.g.*, by prefixing the imported operators with `:` using the prefix form of `require` [8, section 5.2].

3.3 Static Checking

Both the SRE system and the PLT lexer generator statically check regexps. The SRE language supports a simple type inference mechanism that prevents character set operations, such as intersection (`&`), from being misapplied to regexps that might accept more or less than a single character. This system has two types: `1` and `n`.

Figure 5 Illustrative fragment of SRE type system

`T ::= 1 | n`

$$\frac{}{\vdash char : \mathtt{1}} \qquad \frac{\vdash re_1 : t_1 \cdots \vdash re_m : t_m}{\vdash (*\ re_1 \ldots re_m) : \mathtt{n}}$$

$$\frac{\vdash re_1 : \mathtt{1} \cdots \vdash re_m : \mathtt{1}}{\vdash (|\ re_1 \ldots re_m) : \mathtt{1}} \qquad \frac{\vdash re_1 : \mathtt{1} \cdots \vdash re_m : \mathtt{1}}{\vdash (\&\ re_1 \ldots re_m) : \mathtt{1}}$$

$$\frac{\vdash re_1 : t_1 \cdots \vdash re_m : t_m}{\vdash (|\ re_1 \ldots re_m) : \mathtt{n}}$$

Intuitively, a regexp has type `1` iff it must match exactly one character and type `n` if it can match some other number of characters. Regexps with misapplied character set operations have no type.

Figure 5 presents the type system for $*$, `&`, `|`, and *char* SREs—the rules for the polymorphic `|` are most interesting. The macro that processes SREs, `rx`, typechecks regexps that contain no `,exp` forms. For dynamic regexps, it inserts a check that executes when the regexp is assembled at run time.

The PLT lexer regexp language also check character set operations. Instead of using a separate typechecking pass, it integrates the computation with the regexp parsing code. Not only must the lexer generator reject mis-applications of character set primitives, but it must internally group character set regexps into a specialized character set representation. In other words, `(| "a" (| "b" "c"))` is internally represented as `(make-char-set '("a" "b" "c"))`.[2] The DFA-construction algorithm usually operates on only a few characters in each set, whereas it considers each character in a union regexp individually. Thus the character set grouping yields a requisite performance enhancement.

3.4 Summary

Even though a lexer generator must resolve regular expressions statically, its regexp language can still support significant abstractions. Syntactic embedding of the regexp language, via macros, is the key technique for supporting programmer-defined regexp operators. The embedded language can then have static semantics significantly different from Scheme's, as illustrated by the regexp type system.

4 Lexer Generator

After defining the needed regexps, a programmer couples them with the desired actions and gives them to the lexer generator. The resulting lexer matches an input stream against the supplied regexps. It selects the longest match starting from the head of the stream and returns the value of the corresponding action. If two of the regexps match the same text, the topmost action is used.

A lexer is expressed in PLT Scheme with the following form:

```
(lexer (re action) ...)
```

The `lexer` form expands to a procedure that takes an input-port and returns the value of the selected action expression.

The PLT lexer generator lacks the left and right context sensitivity constructs of LEX. Neither feature poses a fundamental difficulty, but since neither omission has been a problem in practice, we have not invested the effort to support them. Cooperating lexers usually provide an adequate solution in situations where left context sensitivity would be used (encoding the context in the program counter), and right context sensitivity is rarely used (lexing Fortran is the prototypical use). The Bigloo lexer supports left context sensitivity, but not right.

4.1 Complement and Intersection

In Section 3.2, we noted that a regexp language for lexers has different characteristics than regexp languages for other applications in that it must be static. In a similar vein, lexer specification also benefits from complement and intersection operators that work on all regexps, not just sets of characters. The PLT lexer generator supports these, as does the lexer generator for the DMS system [2].

Intersection on character sets specializes intersection on general regexps, but complement on character sets is different from complement on general regexps, even when considering single character regexps. For example, the regexp `(char-complement "x")` matches any single character except for `#\x`. The regexp `(complement "x")` matches any string except for the single character string `"x"`, including multiple character strings like `"xx"`.

[2] To handle the large character sets that can arise with Unicode codepoints as characters, the character set representation is actually a list of character intervals.

The following regexp matches any sequence of letters, except for those that start with the letters b-a-d (using a SRE-like sugaring of the PLT regexp syntax with & generalized to arbitrary regexps).

```
(& (+ alphabetic)
   (complement (: "bad" any-string)))
```

The equivalent regexp using only the usual operators (including intersections on character sets) is less succinct.

```
(| (: (& alphabetic (~ "b"))
      (* alphabetic))
   (: "b" (& alphabetic (~ "a"))
          (* alphabetic))
   (: "ba" (& alphabetic (~ "d"))
           (* alphabetic)))
```

The formal specification more closely and compactly mirrors the English specification when using complementation and intersection. We have used this idiom to exclude certain categories of strings from the longest-match behavior of the lexer in specific cases.

As another example, a C/Java comment has the following structure: /* followed by a sequence of characters not containing */ followed by */. Complementation allows a regexp that directly mirrors the specification.

```
(: "/*"
   (complement (: any-string "*/" any-string))
   "*/")
```

The regexp `(: any-string "*/" any-string)` denotes all strings that contain */, so `(complement (: any-string "*/" any-string))` denotes all strings that do not contain */. Notice that `(complement "*/")` denotes all strings except the string `"*/"` (including strings like `"a*/"`), so it is not the correct expression to use in the comment definition. For a similar exercise, consider writing the following regexp without complement or intersection.

```
(& (: (* "x") (* "y"))
   (: any-char any-char any-char any-char))
```

4.2 Lexer Actions

A lexer action is triggered when its corresponding regexp is the longest match in the input. An action is an arbitrary expression whose free variables are bound in the context in which the lexer appears. The PLT lexer library provides several variables that let the action consult the runtime status of the lexer.

One such variable, *input-port*, refers to the input-port argument given to the lexer when it was called. This variable lets the lexer call another function (including another lexer) to process some of the input. For example,

```
(define l
  (lexer
    ((+ (or comment whitespace))
     (l input-port))
    ...))
```

instructs the lexer to call *l*, the lexer itself, when it matches whitespace or comments. This common idiom causes the lexer to ignore whitespace and comments. A similar rule is often used to match string constants, as in

```
(#\" (string-lexer input-port))
```

where *string-lexer* recognizes the lexical structure of string constants.

The *lexeme* variable refers to the matched portion of the input. For example,

```
(number2 (token-NUM (string->number lexeme 2)))
```

converts the matched number from a Scheme string into a Scheme number and places it inside of a token.

A lexer often needs to track the location in the input stream of the tokens it builds. The *start-pos* and *end-pos* variables refer to the locations at the start of the match and the end of the match respectively. A lexer defined with `lexer-src-pos` instead of `lexer` automatically packages the action's return value with the matched text's starting and ending positions. This relieves the programmer from having to manage location information in each action.[3]

4.3 Code Generation

Most lexer generators first convert the regexps to a non-deterministic finite automaton (NFA) using Thompson's construction [18] and then use the subset construction to build a DFA, or they combine these two steps into one [1, section 3.9]. The naive method of handling complement in the traditional approach applies Thompson's construction to build an NFA recursively over the regexp. When encountering a complement operator, the subset construction is applied to convert the in-progress NFA to a DFA which is then easily complemented and converted back to an NFA. Thompson's construction then continues. We know of no elegant method for handling complement in the traditional approach. However, the DMS system [2] uses the NFA to DFA and back method of complement and reports practical results.[4]

The PLT lexer generator builds a DFA from its component regular expressions following the derivative based method of Brzozowski [4]. The derivative approach builds a DFA directly from the regexps, and handles complement and intersection exactly as it handles union.

Given a regular expression r, the *derivative* of r with respect to a character c, $D_c(r)$, is $\{s \mid r \text{ matches } cs\}$. The derivative of a regexp can be given by another regexp, and Brzozowski gives a simple recursive function that computes it. The DFA's states are represented by the regexps obtained by repeatedly taking derivatives with respect to each character. If $D_c(r) = r'$, then the state r has a transition on character c to state r'. Given r and its derivative r', the lexer generator needs to determine whether a state equivalent to r' already exists in the DFA. Brzozowski shows that when comparing regexps by equality of the languages they denote, the iterated derivative procedure constructs the minimal DFA. Because of the complexity of deciding regular language equality, he also shows that the process will terminate with a (not necessarily minimal) DFA if regexps are compared structurally, as long as those that differ only by associativity, commutativity and idempotence of union are considered equal.

[3]The `return-without-pos` variable lets src-pos lexers invoke other src-pos lexers without accumulating multiple layers of source positioning.

[4]Michael Mehlich, personal communication

A few enhancements render Brzozowski's approach practical. First, the regexp constructors use a cache to ensure that equal regexps are not constructed multiple times. This allows the lexer generator to use *eq?* to compare expressions during the DFA construction. Next, the constructors assign a unique number to each regexp, allowing the sub-expressions of a union operation to be kept in a canonical ordering. This ordering, along with some other simplifications performed by the constructors, guarantees that the lexer generator identifies enough regexps together that the DFA building process terminates. In fact, we try to identify as many regexps together as we can (such as by canceling double complements and so on) to create a smaller DFA.

With modern large characters sets, we cannot efficiently take the derivative of a regexp with respect to each character. Instead, the lexer generator searches through the regexp to find sets of characters that produce the same derivative. It then only needs to take one derivative for each of these sets. Traditional lexer generators compute sets of equivalent characters for the original regexp. Our derivative approach differs in that the set is computed for each regexp encountered, and the computation only needs to consult parts of the regexp that the derivative computation could inspect.

Owens added the derivative-based lexer generator recently. Previously, the lexer generator used a direct regexp to DFA algorithm [1, section 3.9] (optimized to treat character sets as single positions in the regexp). Both algorithms perform similarly per DFA state, but the derivative-based algorithm is a much better candidate for elegant implementation in Scheme and may tend to generate smaller DFAs. On a lexer for Java, both algorithms produced (without minimization) DFAs of similar sizes in similar times. On a lexer for Scheme, the Brzozowski algorithm produced a DFA about $\frac{2}{5}$ the size (464 states vs. 1191) with a corresponding time difference.

4.4 Summary

Embedding the lexer generator into Scheme places the action expressions naturally into their containing program. The embedding relies on hygienic macro expansion. To support convenient complement and intersections, we moved the lexer generator from a traditional algorithm to one based on Brzozowski's derivative. Even though the derivative method is not uniquely applicable to Scheme, we found it much more pleasant to implement in Scheme than our previous DFA generation algorithm.

5 Parser Generators

A parser is built from a CFG and consumes the tokens supplied by the lexer. It matches the token stream against the CFG and evaluates the corresponding attributes, often producing an AST.

5.1 Grammars

A CFG consists of a series of definitions of *non-terminal* symbols. Each definition contains the non-terminal's name and a sequence of terminal and non-terminal symbols. Uses of non-terminals on the right of a definition refer to the non-terminal with the same name on the left of a definition. A *terminal* symbol represents an element of the token stream processed by the parser.

A parser cannot, in general, efficiently parse according to an arbitrary CFG. The bottom-up parsers generated by YACC use a lookahead function that allows them to make local decisions during parsing and thereby parse in linear time. Lookahead computation has ambiguous results for some grammars (even unambiguous ones), but the LALR(1) lookahead YACC uses can handle grammars for most common situations. Precedence declarations allow the parser to work around some ambiguities. Both the PLT and GT tools follow YACC and use LALR(1) lookahead with precedences.

5.2 Tokens

A parser is almost completely parametric with respect to tokens and their associated values. It pushes them onto the value stack, pops them off it, and passes them to the semantic actions without inspecting them. The parser *only* examines a token when it selects shift/reduce/accept actions based on the tokens in the input stream's lookahead buffer. This is a control dependency on the token representation because the parser must perform a conditional branch that depends on the token it sees.

Nevertheless, most parser generators, including the PLT system, enforce a specific token representation. The PLT system abstracts the representation so that, were it to change, existing lexer/parser combinations would be unaffected. The GT system allows the token representation to be specified on a parser-by-parser basis.

5.2.1 Tokens in GT

The GT parser tool is parameterized over token branch computation; it has no knowledge of the token representation otherwise. The GT parser macro takes the name of a `token-case` macro along with the CFG specification. The parser generator uses the `token-case` macro in the multi-way branch forms it produces:

```
(token-case token-exp
  ((token ...) body ...)
  ...
  (else body ...))
```

The Scheme expression *token-exp* evaluates to a token value, and the *token* elements are the token identifiers declared with the CFG. Scheme's macro hygiene ensures that the identifiers declared in CFG token declarations and the keys recognized by the `token-case` macro interface properly.

The `token-case` macro has a free hand in implementing the primitive token-branch computation. It can produce a type test, if tokens are Scheme values such as integers, symbols, and booleans; extract some form of an integer token-class code from a record structure, if tokens are records; or emit an immediate jump table, if tokens are drawn from a dense space such as the ASCII character set.

The `token-case` branch compiler parameterizes the CFG to Scheme compiler. This ability to factor compilers into components that can be passed around and dropped into place is unique to Scheme. Note, also, that this mechanism has nothing to do with core Scheme *per se*. It relies only on the macro technology which we could use with C or SML, given suitable s-expression encodings.

5.2.2 Tokens in PLT

The PLT parser generator sets a representation for tokens. A token is either a symbol or a token structure containing a symbol and a value, but this representation remains hidden unless the programmer explicitly queries it. A programmer declares, outside of any parser or lexer, the set of valid tokens using the following forms for tokens with values and without, respectively.

```
(define-tokens group-name (token-name ...))
(define-empty-tokens group-name
  (token-name ...))
```

A parser imports these tokens by referencing the group names in its `tokens` argument. The parser generator statically checks that every grammar symbol on the right of a production appears in either an imported token definition or on the left of a production (essentially a non-terminal definition). DrScheme reports violations in terms of the CFG, as discussed in Section 6.2.

The token-declaration forms additionally provide bindings for token-creation functions that help ensure that the lexer creates token records in conjunction with the parser's expectations. For example, `(token-x)` creates an empty token named `x`, and `(token-y val)` creates a non-empty token named `y`. Thus the single external point of token declaration keeps the token space synchronized between multiple lexers and parsers.

5.3 Parser Configuration

A parser specification contains, in addition to a CFG, directives that control the construction of the parser at an operational level. For example, precedence declarations, in the GT `tokens` and PLT `precs` forms, resolve ambiguities in the CFG and in the lookahead computation.

The PLT `start` form declares the non-terminal at the root of the parse tree. When multiple `start` non-terminals appear, the parser generator macro expands into a list containing one parser per `start` non-terminal. Multiple `start` symbols easily allow related parsers to share grammar specifications. (Most other parser generators do not directly support multiple `start` symbols and instead require a trick, such as having each intended `start` symbol derive from the real `start` symbol with a leading dummy terminal. The lexer produces a dummy terminal to select the desired `start` symbol [17, section 10].)

The PLT system `end` form specifies a set of distinguished tokens, one of which must follow a valid parse. Often one of these tokens represents the end of the input stream. (Other parser generators commonly take this approach.) In contrast, GT's `accept-lookaheads` clause supports k-token specifications for parse ends. Thus nothing in GT's CFG language is specifically LALR(1); it could just as easily be used to define an LR(k) grammar, for $k > 1$. Although the current tools only process LALR(1) grammars, the CFG language itself allows other uses.

GT's CFG language makes provision for the end-of-stream (eos) as a primitive syntactic item distinct from the token space. An `accept-lookaheads` specification references eos with the `#f` literal, distinguishing the concept of end-of-stream (absence of a token; a condition of the stream itself) from the set of token values. This was part of clearly factoring the stream representation (*e.g.*, a list, a vector, an imperative I/O channel) from the token representation (*e.g.*, a record, a character, a symbol) and ensures that the token space is not responsible for encoding a special end-of-stream value.

Figure 6 The PLT parser language

```
parser ::= (parser-clause ...)

parser-clause ::= (start nterm ...)                  ; Starting non-terminals
                | (end term ...)                     ; Must follow parse only
                | (tokens token-group ...)           ; Declare tokens
                | (error scheme-exp)                 ; Called before error correction
                | (grammar (nterm rhs ...) ...)      ; Defines the grammar
                | (src-pos)                          ; Optional: Automatic source locationing
                | (precs (prec term ...) ...)        ; Optional: Declare precedences
                | (debug filename)                   ; Optional: print parse table & stack on error
                | (yacc-output filename)             ; Optional: Output the grammar in YACC format
                | (suppress)                         ; Optional: Do not print conflict warnings

rhs      ::= ((gsym ...) [term] action)   ; Production with optional precedence tag
prec     ::= left | right | nonassoc      ; Associativity/precedence declarator
gsym     ::= term | nterm                 ; Grammar symbol
action   ::= scheme-exp                   ; Semantic action
filename ::= string
term, nterm, token-group ::= identifier
```

5.4 Attribute Computation

Each production in the grammar has an associated expression that computes the attribute value of the parse-tree node corresponding to the production's left-hand non-terminal. This expression can use the attribute values of the children nodes, which correspond to the grammar symbols on the production's right side. In YACC, the variable *$n* refers to the value of the n[th] grammar symbol.

The PLT system non-hygenically introduces *$n* bindings in the attribute computations. Tokens defined with `define-empty-tokens` have no semantic values, so the parser form does not bind the corresponding *$n* variables in the semantic actions. The parser generator thereby ensures that a reference to a *$n* variable either contains a value or triggers an error.

Instead of requiring the *$n* convention, the GT design places no restrictions on the variables bound to attribute values. For example, the subtraction production from a simple calculator language,

```
(non-term exp
  ...
  (=> ((left exp) - (right exp))  (- left right))
  ...)
```

hygienically introduces the `left` and `right` bindings referenced from the attribute computation, `(- left right)`. A grammar symbol without enclosing parentheses, such as `-`, specifies no binding, indicating to downstream tools that the token's semantic value may be elided from the value stack when it is shifted by the parser. (Essentially, the empty-token specification shows up in the CFG specification.)

As a convenience syntax, if the variable is left unspecified, as in

```
(=> ((exp) - (exp))  (- $1 $3))
```

then the *$n* convention is used. This unhygienic bit of syntactic sugar is convenient for hand-written parsers, while the explicit-binding form provides complete control over variable binding for hygienic macros that generate CFG forms.

For further convenience, the `implicit-variable-prefix` declaration can override the *$* prefix. Thus, a handwritten parser can arrange to use implicitly-bound variables of the form `val-1`, `val-2`, ..., with the declaration

```
(implicit-variable-prefix val-)
```

The *$n* notation is unavailable in the Bigloo parser generator. Instead, the grammar symbol's name is bound to its value in the action. Because the same grammar symbol could appear more than once, the programmer can choose the name by appending it to the grammar symbol's name with the `@` character in-between. The Bigloo design provides more naming naming control than the PLT system, but no more control over hygiene. Additionally, it can lead to confusion if grammar symbols or attribute bindings already contain `@`.

5.5 Code Generation

Although the PLT and GT parser generators are based on YACC's design, both use a syntactic embedding of the parser specification into Scheme, much as PLT's lexer generator does. In the PLT system, a programmer writes a parser by placing a specification written in the language shown in Figure 6 inside of a `(parser ...)` form. The `(parser ...)` form compiles the grammar into a parse table using LALR(1) lookahead and supplies an interpreter for the table. These two elements are packaged together into a parser function. The GT parser system uses the language in Figure 7 for its CFG specifications. As in the PLT system, a CFG specification is placed inside of a macro that compiles the CFG form into the target language. However, the GT design provides a much wider range of implementation options than the PLT and other systems.

The GT system factors the parser tool-chain into multiple languages. The programmer writes a parser using the CFG language and the parser generator compiles it to a Scheme implementation in three steps. It transforms the CFG into a TLI (for "target-language independent") specification which it then expands to an equivalent parser in a push-down automata (PDA) language which it finally compiles into Scheme. The continuation-passing-style (CPS) macro `(cfg->pda` *cfg form* `...)` packages up the LALR com-

Figure 7 The GT CFG language

```
cfg ::= (clause ...)

clause ::= (tokens token-decl ...)                       ; Declare tokens and precedence tags
         | (non-term nterm rhs ...)                      ; Declare a non-terminal
         | (accept-lookaheads lookahead ...)             ; Must come after parse
         | (error-symbol ident [semantic-value-proc])    ; Error-repair machinery
         | (no-skip token ....)                          ; Error-repair machinery
         | (implicit-variable-prefix ident)              ; Defaults to $
         | (allowed-shift/reduce-conflicts integer-or-false)

token-decl ::= token
             | (non token ...)      ; Non-associative tokens
             | (right token ...)    ; Right-associative tokens
             | (left token ...)     ; Left-associative tokens

rhs ::= (=> [token] (elt ...) action)   ; Production w/optional precedence tag

elt ::= symbol            ; Symbol w/unused semantic value
      | (var symbol)      ; Symbol binding semantic value to var
      | (symbol)          ; Symbol w/implicitly bound semantic value

lookahead ::= (token ...[#f])   ; #f marks end-of-stream.
action ::= scheme-exp
symbol, nterm, token, var ::= ident
```

piler machinery and performs the first two steps. It expands to (*form* ... *pda*), where *pda* is the PDA program compiled from the original *cfg* form. The macro `pda-parser/imperative-io` takes a PDA program, along with the `token-case` macro other relevant forms specifying the input-stream interface, and expands it into a complete Scheme parser. An alternate macro, `pda-parser/pure-io` maps a PDA program to Scheme code using a functional-stream model; it is intended for parsing characters from a string, or items from a list. The main parser macro simply composes the `cfg->pda` macro with one of the PDA-to-Scheme macros to get a Scheme parser; this is a three-line macro.

Exporting the PDA language lets the system keep the token-stream mechanism abstract throughout the CFG-to-PDA transformation. The two PDA-to-Scheme macros each provide a distinct form of token-stream machinery to instantiate the abstraction. In contrast, the PLT system fixes the token-stream representation as a function of no arguments that returns a token. Successive calls to the function should return successive tokens from the stream.

5.5.1 The TLI language

GT system was designed to be target-language neutral. That is, to specify a parser in C instead of in Scheme using the CFG language, we would only need an s-expression concrete grammar for C in which to write the semantic actions. This means that the CFG-processing tools for the GT system are also independent of the target language and the language used for the semantic actions. Note that Scheme creeps out of the semantic actions and into the rest of the grammar language in only one place: the variable-binding elements of production right-hand sides. These variables (such as the `left` and `right` variables bound in the above example) are Scheme constructs.

To excise this Scheme dependency, the GT system defines a slightly lower-level language than the CFG language defined in Figure 7. The lower-level language (called the TLI language) is identical to the main CFG language, except that (1) the `implicit-variable-prefix` clause is removed (having done its duty during the CFG-to-TLI expansion), and (2) variable binding is moved from the production rhs to the semantic-action expression. In the TLI language, the example above is rewritten to

```
(=> ((exp) - (exp))                          ; Grammar
    (lambda (left right) (- left right)))   ; Scheme
```

As in the the main CFG language, parentheses mark grammar symbols whose semantic values are to be provided to the semantic action. The TLI language is *completely* independent of the target language, except for the semantic actions. In particular, TLI has nothing to do with Scheme at all. This means that the CFG can be compiled to its push-down automaton (PDA) with complete indifference to the semantic actions. They pass through the LALR transformer unreferenced and unneeded, to appear in its result. Because the TLI language retains the information about which semantic values in a production are needed by the semantic action, optimizations can be performed on the parser in a target-language independent manner, as we will see below.

5.5.2 The PDA language

A PDA program (see Figure 8) is primarily a set of *states*, where each state is a collection of shift, reduce, accept and goto *actions*. Shift, reduce and accept actions are all guarded by *token lookahead* specifications that describe what the state of the token stream must be in order for the guarded action to fire. A non-terminal symbol guards a goto action. Reduce actions fire named *rules*, which are declared by `rule` clauses; a reduction pops semantic values off the value stack and uses its associated semantic action to compute the replacement value.

The PDA design contains several notable elements. The lookahead syntax allows for k-token lookahead, for $k = 0, 1$ and greater, so the PDA language supports the definition of LR(k) parsers (al-

Figure 8 The PDA language

```
pda ::= (pda-clause ...)

pda-clause ::= (comment form ...)                                      ; Ignored
             | (tokens token ...)                                      ; Declare tokens
             | (state state-name action ...)
             | (rule rule-name non-term bindings semantic-action)
             | (error-symbol ident [semantic-value-proc])              ; Error-repair machinery
             | (no-skip token ...)                                     ; Error-repair machinery

action ::= (comment form ...)                ; Ignored
         | (shift lookahead state-name)      ; Shift, reduce & accept
         | (reduce lookahead rule-name)      ;    actions all guarded
         | (accept lookahead)                ;    by token-lookaheads.
         | (goto non-term state-name)        ; Goto action guarded by non-terminal
         | (error-shift ident state-name)    ; Error-repair machinery

lookahead ::= (token ...[#f])   ; #f marks end-of-stream
bindings  ::= (boolean ...)     ; #f marks a value not passed to semantic action
state-name, rule-name ::= ident
token, non-term ::= ident
```

though our tools only handle $k \leq 1$). As action-selection is order-dependent, the zero-token lookahead `()` is useful as a default guard.

Also notable, the *bindings* element of the rule form is a list of boolean literals, whose length determines how many semantic values are popped off the value stack. Only values tagged with a `#t` are passed to the semantic action; values tagged with a `#f` are discarded. As an example, the reduction rule

```
;;;  ifexp ::= if <exp> then <stmt> else <stmt> fi
(rule r7 ifexp (#t   #t   #f   #t   #f   #t   #t)
  (lambda (iftok exp stmt1 stmt2 fitok)
    (make-ifexp exp stmt1 stmt2
       (token:leftpos  iftok)     ;Position-tracking
       (token:rightpos fitok)))) ;machinery
```

specifies a rule that will pop seven values off the stack, but only pass five of them to the semantic action. Thus, the semantic action is a Scheme procedure that takes only five arguments, not seven.

The bindings element allows a PDA optimizer, via static analysis, to eliminate pushes of semantic values that will ultimately be discarded by their consuming reductions—in effect, useless-value elimination at the PDA level. The *bindings* form specifies the local data dependencies of the semantic actions. This key design point allows data-flow analysis of the PDA program *without requiring any understanding of the language used to express the semantic action*, which in turn supports strong linguistic factoring. The semantic action s-expression could encode a C statement, or an SML expression just as easily as a Scheme expression; a PDA optimizer can analyse and transform a PDA program with complete indifference.

A great allure of PDA computation is its sub-Turing strength, which means that we have a much easier time analyzing PDA programs than those written in a Turing-equivalent language. The moral might be: always use a tool small enough for the job. We have designed and are currently implementing a lower-level PDA0 language, which allows source-to-source optimizations such as non-terminal folding, control- and data-flow analysis, and dead-state elision. This has the potential to make very lightweight parsing practical, *i.e.*, parsers that parse all the way down to individual characters, yet still assemble tokens at lexer speeds. Again, this can all be provided completely independent of the eventual target language by defining CPS macros that work strictly at the PDA0 level.

Factoring out the PDA as a distinct language also supports multiple producers as well as multiple consumers of PDA forms. One could implement SLR, canonical LR and other CFG processors to target the same language, and share common back end.

5.6 Summary

Although the PLT and GT parser generators follow the general design of YACC, both systems syntactically embed parser specifications in Scheme. The embedding benefits the PLT parser generator in the same way it benefits the PLT lexer generator, whereas the GT system takes advantage of the syntactic embedding to maximize flexibility. In GT, the token structure is specified on a per-parser basis through a `token-case` macro, avoiding any commitment to a particular lexer/parser interface. (The PLT token strategy could be implemented as a `token-case` macro.) Furthermore, the GT system provides complete freedom over naming in the grammar and attributes, without compromising hygiene. We think GT's attribute naming system is superior to other Scheme parser generators, including Bigloo's and PLT Scheme's. By using a language tower, the GT system can isolate details of one level from the others. This allows, for example, easily switching between multiple PDA implementations and token stream representations with the same CFG. The separation of the token and end-of-stream representations supports the use of different kinds of token-stream representations.

6 Taking Full Advantage of Syntax

As we have seen, syntactic embeddings of lexer and parser specifications allow the lexer and parser generator to perform the translation to DFA and PDA at compile time. The syntactic approach also supports the debugging of parser specifications and lets the program development environment operate on them.

6.1 Debugging Parsers

Static debugging of an LR parser has two components: detecting malformed grammars and semantic actions, and detecting grammars that do not conform to the requirements of the parsing methodology in use. The PLT system helps programmers with the former kinds of error using the techniques mentioned in Section 5.2 and Section 6.2. The GT system's multi-level design gives programmers an elegant way of approaching the latter kinds of problems.

Most parsing methodologies (including LL(k), LR(k), and LALR) cannot handle all unambiguous CFGs, and each builds ambiguous PDAs on some class of unambiguous CFGs. Analyzing and fixing these grammars necessarily requires an examination of the item sets associated with the conflicted states of the broken PDA—they must be debugged *at the PDA level*. In most systems, including YACC and the PLT system, these errors are debugged by printing out and then carefully studying a report of the grammar's ambiguous characteristic finite-state automaton (CFSA), which is essentially the program defining the PDA.

An ambiguous PDA has multiple shift/reduce/accept transitions guarded by the same lookahead, so the check for bad grammars occurs statically in the PDA-to-Scheme macro. Because the GT parser system factors the parser tool chain into multiple language levels, the report machinery comes for free: the PDA program is the report. Since the LALR compiler is exported as a CPS macro, using `quote` for the syntactic continuation shifts from language-level to data structure. That is, this Scheme form `(cfg->pda` *cfg* `quote)` expands to `(quote` *pda*`)` so the the following expression produces an error report.

```
(pretty-print (cfg->pda cfg quote))
```

The PDA language includes a `comment` clause for the LALR compiler to record the item-set information for each state. This information is critical for human understanding of the PDA. An example of a state generated by `cfg->pda` is

```
(state s15
  (comment (items (=> exp (exp () divide exp))
                  (=> exp (exp () times exp))
                  (=> exp (exp minus exp ()))
                  (=> exp (exp () minus exp))
                  (=> exp (exp () plus exp))))
  (reduce (r-paren) r11)
  (reduce (semicolon) r11)
  (comment (reduce (times) r11))
  (shift (times) s11)
  (comment (reduce (divide) r11))
  (shift (divide) s12)
  (comment (reduce (plus) r11))
  (shift (plus) s13)
  (comment (reduce (minus) r11))
  (shift (minus) s14)
  (reduce (#f) r11))
```

A comment clause lists the kernel set of the state's items. The item comments are grammar productions with `()` allowed on the right-hand sides to mark the item cursor. (We did not use the traditional dot marker '.' for obvious reasons.) The LALR compiler comments out ambiguous actions that are resolved by precedence, associativity, or the allowed-conflict-count declaration. State s15 has four of these. Had one of them not been resolved by the LALR macro, the resulting PDA would be ambiguous, causing the PDA macro to report a static error that the programmer would have to debug.

The GT tools also have small touches to help the programmer focus in on the problem states. The LALR compiler leaves a simple report in a comment at the top of the PDA form listing the names of all conflicted states, *e.g.*,

```
(comment (conflict-states s41 s63 s87))
```

The GT tools also provide a PDA analyser that filters a PDA and produces a reduced PDA that containing only the ambiguous states of the original program. Because we can so trivially render the PDA as a Scheme s-expression, it is easy to comb through a PDA or otherwise interactively manipulate it using the usual suite of Scheme list-processing functions such as `filter`, `fold`, `map`, `any` and so forth—a luxury not afforded to YACC programmers.

Placing PDA static-error detection in the PDA tools, where it belongs, has another benefit. Since the LALR compiler will happily produce an ambiguous PDA, we could produce a generalized LR (GLR) parser simply by implementing a nondeterministic PDA as a distinct macro from the current PDA-to-Scheme one. It would compose with the current `cfg->pda` macro, handling ambiguous grammars without complaint allowing reuse of the complex LALR tool with no changes.

6.2 Little Languages and PDEs

The DrScheme program development environment has several features that display feedback directly on a program's source. Specifically, DrScheme highlights expressions that have caused either a compile-time or run-time error, and the Check Syntax tool draws arrows between binding and bound variables. Check Syntax inspects fully expanded Scheme source code to determine arrow placements. The action and attribute expressions inside the PLT `lexer` and `parser` forms appear directly in the expanded code with their lexical scoping and source location information intact, so that Check Syntax can draw arrows, and DrScheme can highlight errors as demonstrated in Figures 1 and 2 in Section 2.2.

The `lexer` and `parser` forms expand into DFA and parse tables, leaving out the source regular expression and CFG specifications. Thus, DrScheme requires extra information to fully support these forms. Run-time error highlighting is not an issue, because the regexp or grammar itself cannot cause a runtime error. The `lexer` and `parser` forms directly signal compile-time errors (*e.g.*, for an unbound regexp operator or terminal), including the source location of the error, to DrScheme. As they parse the input regexp or grammar expression, each sub-expression (as a syntax object) contains its source location, so they can conveniently signal such errors.

To inform Check Syntax of dependencies in the grammar, the `parser` form emits a dummy `let` form as dead code, along with the parse table and actions. The `let` includes a binding for each non-terminal and token definition, and its body uses each grammar symbol that occurs on the right of a production. The `let` introduces a new scope for all of the non-terminals and tokens, ensuring that they do not interfere with outside identifiers of the same name. The `parser` form generates the following `let` for the example in Figure 9.

```
(let ((exp void)
      (NUM void)
      (− void)
      (EOF void))
  (void NUM exp − exp))
```

Figure 9 Locating the uses of a token and a non-terminal

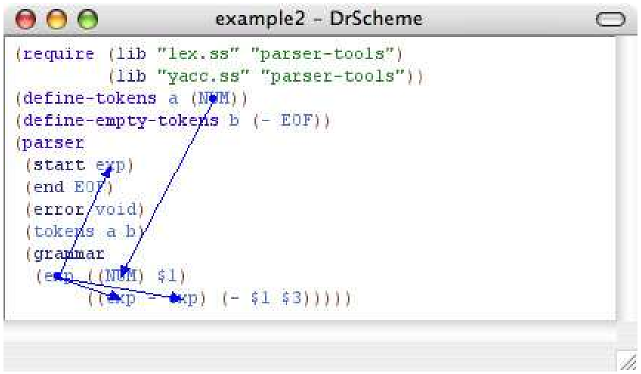
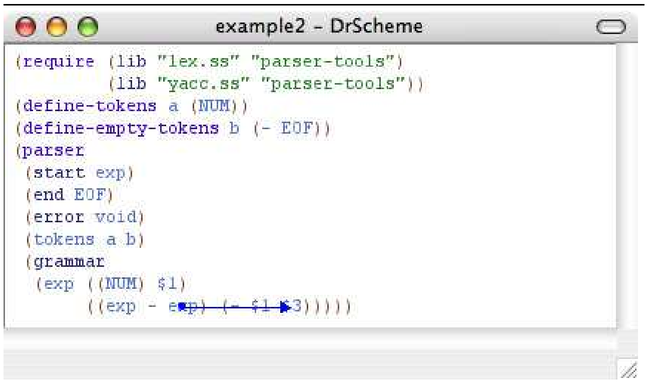


Figure 10 Correlating an action with the grammar



We use a different approach for the situation shown in Figure 10. The parser generator wraps the action with a `lambda` that binds the *$3*. To cause Check Syntax to draw an arrow, the `lambda`'s *$3* parameter uses the source location of the referent grammar symbol. With a GT-style hygienic naming option, we would use the identifier supplied with the grammar symbol in the `lambda` instead, and Check Syntax could then draw the arrow appropriately to the binder. Furthermore, α-renaming could be used to change the name. This illustrates that hygienic macros interact more naturally with programming tools, and not just with other macros.

Like any compiler, a macro that processes an embedded language must respect that language's dynamic semantics by generating code that correctly executes the given program. Also like any compiler, the macro must implement the language's static semantics. It can do this by performing the requisite static checking itself, as in the SRE type system and the PLT `parser` form's check for undefined grammar symbols, or it can arrange for statically invalid source programs to generate statically invalid target programs. In this case, the macro effectively re-uses the target language's static checking. This is how the `parser` form handles unbound *$n* identifiers, by letting Scheme's free variable detection catch them. Even for the static properties checked directly by the macro, it might need to emit annotations (such as the `let` mentioned above) to preserve static information for tools like Check Syntax.

7 References

[1] A. A. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] I. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program transformations for practical scalable software evolution. In *International Conference on Software Engineering*, 2004.

[3] D. Boucher. A portable and efficient LALR(1) parser generator for Scheme. `http://www.iro.umontreal.ca/~boucherd/Lalr/documentation/lalr.html`.

[4] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, October 1964.

[5] F. DeRemer and T. Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Trans. Program. Lang. Syst.*, 4(4):615–649, 1982.

[6] D. Dubé. SILex. `http://www.iro.umontreal.ca/~dube/`.

[7] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1993.

[8] M. Flatt. *PLT MzScheme: Language Manual*, 2004. `http://download.plt-scheme.org/doc/mzscheme/`.

[9] D. P. Friedman, M. Wand, and C. P. Haynes. *Essentials of Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2001.

[10] M. Johnson. `http://cog.brown.edu:16080/~mj/Software.htm`.

[11] M. Serrano. *Bigloo: A "practical Scheme compiler"*, 2004. `http://www-sop.inria.fr/mimosa/fp/Bigloo/doc/bigloo.html`.

[12] O. Shivers. A scheme shell. *Higher-order and Symbolic Computation*. to appear.

[13] O. Shivers. A universal scripting framework, or lambda: the ultimate "little language". In *Concurrency and Parallelism, Programming, Networking, and Security*, volume 1179 of *Lecture Notes in Computer Science*, pages 254–265. Springer, 1996.

[14] O. Shivers. The SRE regular-expression notation. `http://www.cs.gatech.edu/~shivers/sre.txt`, 1998.

[15] O. Shivers and B. Carlstrom. The scsh manual. `ftp://www-swiss.ai.mit.edu/pub/su/scsh/scsh-manual.ps`.

[16] M. Sperber and P. Thiemann. Generation of LR parsers by partial evaluation. *ACM Trans. Program. Lang. Syst.*, 22(2):224–264, 2000.

[17] D. R. Tarditi and A. W. Appel. *ML-Yacc User's Manual: Version 2.4*, 2000. `http://www.smlnj.org/doc/ML-Yacc/`.

[18] K. Thompson. Programming Techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, June 1968.