Automatic construction of parse trees for lexemes [*]

Danny Dubé
Université Laval
Quebec City, Canada
Danny.Dube@ift.ulaval.ca

Anass Kadiri
ÉPITA
Paris, France
Anass.Kadiri@gmail.com

Abstract

Recently, Dubé and Feeley presented a technique that makes lexical analyzers able to build parse trees for the lexemes that match regular expressions. While parse trees usually demonstrate how a word is generated by a context-free grammar, these parse trees demonstrate how a word is generated by a regular expression. This paper describes the adaptation and the implementation of that technique in a concrete lexical analyzer generator for Scheme. The adaptation of the technique includes extending it to the rich set of operators handled by the generator and reversing the direction of the parse trees construction so that it corresponds to the natural right-to-left construction of the lists in Scheme. The implementation of the adapted technique includes modifications to both the generation-time and the analysis-time parts of the generator. Uses of the new addition and empirical measurements of its cost are presented. Extensions and alternatives to the technique are considered.

Keywords Lexical analysis; Parse tree; Finite-state automaton; Lexical analyzer generator; Syntactic analysis; Compiler

1. Introduction

In the field of compilation, more precisely in the domain of syntactic analysis, we are used to associate the notion of parse tree, or derivation tree, to the notion of context-free grammars. Indeed, a parse tree can be seen as a demonstration that a word is generated by a grammar. It also constitutes a convenient structured representation for the word. For example, in the context of a compiler, the word is usually a program and the parse tree (or a reshaped one) is often the internal representation of the program. Since, in many applications, the word is quite long and the structure imposed by the grammar is non-trivial, it is natural to insist on building parse trees.

However, in the related field of lexical analysis, the notion of parse trees is virtually inexistent. Typically, the theoretical tools that tend to be used in lexical analysis are regular expressions and finite-state automata. Very often, the words that are manipulated are rather short and their structure, pretty simple. Consequently, the notion of parse trees is almost never associated to the notion of lexical analysis using regular expressions. However, we do not necessarily observe such simplicity in all applications. For instance, while numerical constants are generally considered to be simple lexical units, in a programming language such as Scheme [9], there are integers, rationals, reals, and complex constants, there are two notations for the complex numbers (rectangular and polar), there are different bases, and there are many kinds of prefixes and suffixes. While writing regular expressions for these numbers is manageable and matching sequences of characters with the regular expressions is straightforward, extracting and interpreting the interesting parts of a matched constant can be much more difficult and error-prone.

This observation has lead Dubé and Feeley [4] to propose a technique to build parse trees for lexemes when they match regular expressions. Until now, this technique had remained paper work only as there was no implementation of it. In this work, we describe the integration of the technique into a genuine lexical analyzer generator, SILex [3], which is similar to the Lex tool [12, 13] except that it is intended for the Scheme programming language [9]. In this paper, we will often refer to the article by Dubé and Feeley and the technique it describes as the “original paper” and the “original technique”, respectively.

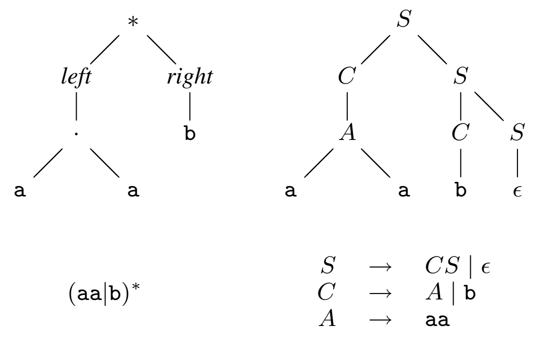
Sections 2 and 3 presents summaries of the original technique and SILex, respectively. Section 4 continues with a few definitions. Section 5 presents how we adapted the original technique so that it could fit into SILex. This section is the core of the paper. Section 6 quickly describes the changes that we had to make to SILex to add the new facility. Section 7 gives a few concrete examples of interaction with the new implementation. The speed of parse tree construction is evaluated in Section 8. Section 9 is a brief discussion about related work. Section 10 mentions future work.

2. Summary of the construction of parse tree for lexemes

Let us come back to the original technique. We just present a summary here since all relevant (and adapted) material is presented in details in the following sections.

The original technique aims at making lexical analyzers able to build parse trees for the lexemes that they match. More precisely, the goal is to make the *automatically generated* lexical analyzers able to do so. There is not much point in using the technique on analyzers that are written by hand. Note that the parse trees are those for the lexemes, not those for the regular expressions that match the latter. (Parse trees for the regular expressions themselves can be obtained using conventional syntactic analysis [1].) Such a parse tree is a demonstration of how a regular expression generates a word, much in the same way as a (conventional) parse tree demonstrates how a context-free grammar generates a word. Figure 1 illustrates what the parse trees are for a word `aab` that is generated by both a

[*] This work has been funded by the National Sciences and Engineering Research Council of Canada.



$(\texttt{aa}|\texttt{b})^*$

$$\begin{array}{lcl} S & \rightarrow & CS \mid \epsilon \\ C & \rightarrow & A \mid \texttt{b} \\ A & \rightarrow & \texttt{aa} \end{array}$$

Figure 1. Parse trees for a word `aab` that is generated by a regular expression and a context-free grammar.

regular expression and an equivalent context-free grammar. While the parse tree on the right-hand side needs no explanation, the other one may seem unusual. It indicates the following: the Kleene star has been used for 2 iterations; in the first iteration, the *left*-hand side alternative has been selected and, in the second one, it was the *right*-hand side alternative; the sub-tree for `aa` is a concatenation (depicted by the implicit $\cdot$ operator). Note that the *left* and *right* labels are used for illustration purposes only. In general, any number of alternatives is allowed and the labels are numbered.

One might wonder why parse trees should be built for lexemes. Typically, compiler front-end implementors tend to restrict the lexical elements to relatively simple ones (e.g. identifiers, literal character constants, etc.). Even when more "complex" elements such as string constants are analyzed, it is relatively easy to write a decoding function that extracts the desired information from the lexemes. When some elements are genuinely more complex, their treatment is often deferred to the syntactic analysis. However, there are cases where the nature of the elements is truly lexical and where these are definitely *not* simple. In the introduction, we mentioned the numerical constants in Scheme. These are definitely lexical elements (no white space nor comments are allowed in the middle of a constant), yet their lexical structure is quite complex. In Section 7, we illustrate how one can benefit from obtaining parse trees for Scheme numerical constants. Moreover, it is a "chicken-and-egg" kind of issue since, by having more powerful tools to manipulate complex lexical elements, implementors may choose to include a wider variety of tasks as part of the lexical analysis phase.

The idea behind the technique described in the original paper is pretty simple. Because the automatically generated lexical analyzers are usually based on finite-state automata, the technique is based on automata too, but with a simple extension. The augmented automata are built using straightforward structural induction on the regular expressions to which they correspond. The addition to the automata consists only in putting *construction commands* on some arcs of the automata. The purpose of the construction commands is simple: let r be a regular expression, $A(r)$, the corresponding automaton, and w, a word; if a path P traverses $A(r)$ and causes w to be consumed, then the sequence of construction commands found along P forms a "recipe" that dictates how to build a parse tree t which is a demonstration that r generates w.

The automata that are built using the original technique are non-deterministic. It is well-known that performing lexical analysis using non-deterministic finite-state automata (NFA) is generally slower than using deterministic finite-state automata (DFA). Consequently, conversion of the NFA into DFA is desirable.

The augmented NFA can indeed be converted into DFA. However, note that a path taken through a DFA while consuming some word has little to do with the corresponding path(s) in the NFA, because of the presence of ϵ-transitions and arbitrary choices featured by the latter. Since one needs the construction commands of the NFA to build a parse tree, there must exist a mechanism that allows one to recover a path through the NFA from a path through the DFA. The technique proposes a mechanism that is implemented using three tables that preserve the connection between the DFA and the NFA. By making a series of queries to these tables, one is able to efficiently convert a path through the DFA into a corresponding path through the NFA. The path through the NFA trivially can be translated into a sequence on commands that explain how to build a parse tree. To summarize, the process of recognizing a lexeme and building a parse tree for it consists in identifying the lexeme using the DFA in the usual way while taking note of the path, recovering the path through the NFA, and then executing the sequence of commands.

The technique presented in the original paper deals only with the most basic regular operators: concatenation, union, and the Kleene star. Two distinct representations for the parse trees are introduced: the internal representation and the external one. The first one manipulates the trees as data structures. The second one manipulates them under their *printed* form, i.e. as words. Since the current paper is about lexical analyzers, we only consider the internal representation of parse trees. Finally, the original paper presents how one can obtain the complete set of parse trees for a word w that matches a regular expression r. Indeed, as shown below, the parse tree needs not be unique. In fact, there can be huge numbers (even an infinity) of parse trees, in some cases. Consequently, sets of parse trees are always represented under an implicit form only. We consider complete sets of parse trees to be mainly of theoretical interest and the current paper only considers the construction of a single parse tree for any lexeme.

3. SILex: a lexical analyzer generator for Scheme

SILex has originally been designed to be similar to the original Lex tool for the C language. In particular, the syntax of the regular expressions and the set of operators are the same. However, the actions that specify how to react to the recognition of a lexeme must be written in Scheme as expressions. In SILex, the actions *return* tokens while, in Lex, the actions produce tokens using a mixture of a returned value and side-effects. The third part of the specification files for Lex, which contains regular C code, does not have a counterpart in SILex. Consequently, the specification files for SILex include the part for the definition of macros (shorthands for regular expressions) and the part for the rules. SILex offers various services: many lexical analyzers may be used to analyze the same input; counters are automatically updated to indicate the current position inside of the input; the DFA can be represented using ordinary (compact) or portable tables, or can be directly implemented as Scheme code.

3.1 Lexemes

We describe the set $\mathcal{R}$ of regular expressions supported by SILex. All regular expressions in $\mathcal{R}$ are presented in Figure 2. Each kind of regular expression is accompanied by a short description and its language. We use Σ to denote the set of characters of the Scheme implementation at hand (e.g. the ASCII character set). The language of a regular expression r is denoted by $L(r)$. In the figure, c ranges over characters ($c \in \Sigma$), i and j ranges over integers ($i, j \in \mathbb{N}$), *spec* denotes the specification of the contents of a character class, C ranges over character classes ($C \subseteq \Sigma$), and v ranges over strings ($v \in \Sigma^*$). All variables r and r_i are assumed to be in $\mathcal{R}$. Finally, $\rho_L : \mathcal{R} \times \mathbb{N} \times (\mathbb{N} \cup \{\infty\}) \rightarrow 2^{\Sigma^*}$ is a repetition function defined

DESCRIPTION	REGULAR EXPRESSION	LANGUAGE
Ordinary character	c	$\{c\}$
Any character	.	$\Sigma - \{\textit{newline character}\}$
Newline character	`\n`	$\{\textit{newline character}\}$
Character by code	$\backslash i$	$\{\textit{character of code } i\}$
Quoted character	$\backslash c$	$\{c\}$
Character class	$[\textit{spec}]$	$C \subseteq \Sigma$
Literal string	$"v"$	$\{v\}$
Parenthesized expression	(r)	$L(r)$
Kleene closure	r^*	$\rho_L(r, 0, \infty)$
Positive closure	r^+	$\rho_L(r, 1, \infty)$
Optional expression	$r^?$	$\rho_L(r, 0, 1)$
Fixed repetition	$r\{i\}$	$\rho_L(r, i, i)$
At-least repetition	$r\{i,\}$	$\rho_L(r, i, \infty)$
Between repetition	$r\{i, j\}$	$\rho_L(r, i, j)$
Concatenation	$r_0 \ldots r_{n-1}$	$L(r_0) \ldots L(r_{n-1})$
Union	$r_0 \mid \ldots \mid r_{n-1}$	$L(r_0) \cup \ldots \cup L(r_{n-1})$

Figure 2. Regular expressions supported by SILex and the corresponding languages.

as:

$$\rho_L(r, b, B) = \bigcup_{\substack{i \in \mathbb{N} \\ b \le i \le B}} (L(r))^i$$

Many details are omitted in the presentation of $\mathcal{R}$ by lack of relevance for this paper. For instance, the exact set of ordinary characters and the syntax of the character classes are not really interesting here. For complete information about the syntax, we refer the reader to the documentation of SILex [3]. The important thing to know about character classes is that an expression $[\textit{spec}]$ matches nothing else than a single character and it does match a character c if $c \in C$ where C is the set of characters denoted by *spec*.

Operators used to build up regular expressions have different priority. We assume that the repetition operators (*, $^?$, $\{i,\}$, ...) have higher priority than the (implicit) concatenation operator and that the latter has higher priority than the union operator. Moreover, we expect unions (and concatenations) to account for all sub-expressions that are united (or concatenated, respectively). In other words, when we write a union $r_0 \cup \ldots \cup r_{n-1}$, none of the r_i should be a union. Likewise, when we write a concatenation $r_0 \ldots r_{n-1}$, none of the r_i should be a concatenation (nor a union, naturally). Repetition operators, though, can be piled up (e.g. as in expression $\mathtt{d}^?\{2, 4\}^+$).

From now on, we forget about the first 5 kinds of regular expressions. These can all be represented by totally equivalent character classes (equivalent according to their language *and* according to their associated parse trees, too). For instance, expressions `f` and . can be replaced by [`f`] and [`^\n`], respectively. As for the literal strings, we choose *not* to forget about them. Although it could be tempting to replace them by concatenation of characters, which would denote the same language, we refrain to do so because, as we see later, it would change the associated parse trees. For efficiency reasons, the parse trees for literal strings are different from those for concatenations. The former are cheaper to generate than the latter.

3.2 Incompatibilities with the original technique

The original technique for the construction of parse trees for lexemes cannot be integrated directly into SILex for two reasons. First, SILex provides a larger set of operators in regular expressions than the one presented in the original paper. Second, the original technique builds lists by adding elements *to the right*. This does not correspond to the efficient and purely functional way of building lists in Scheme. Consequently, the rules for the construction of the NFA with commands have to be adapted to the larger set of operators and to the direction in which Scheme lists are built.

4. Definitions

There are some terms specific to the domain of lexical analysis that need to be defined. At this point, we have already defined *regular expressions* along with their language. In the context of compiler technology, unlike in language theory, we are not only interested in checking if a word w matches a regular expression r (i.e. whether $w \in L(r)$), but also in the decomposition of the input u ($\in \Sigma^*$) into a stream of lexemes that leads to a stream of tokens. A *lexeme* is a prefix w of the input u ($u = wu'$) that matches some regular expression r. Based on the matching regular expression r and the matched lexeme w, a *token* is produced. Examples of tokens include: the identifier named `trace`, the reserved keyword `begin`, the operator `+`, etc. Typically, the stream of tokens that is produced by lexical analysis constitutes the input to the syntactic analyzer. While the concept of token is variable and depends on the application, the concept of lexeme is standard and can be defined in terms of language theory. Usually, when a lexeme w has been identified, i.e. when $u = wu'$, and that the corresponding token has been produced, w is considered to have been consumed and the remaining input is u'.

In the context of automatic generation of lexical analyzers, there is typically more than one regular expression, r_i, that may match lexemes. Lexical analyzers are usually specified using a list of *rules*, each rule being an association between a regular expression r_i and an *action* α_i. An action α_i is some statement or expression in the target programming language that indicates how to produce tokens when lexemes are found to match r_i. The action normally has access to the matching lexeme and also has the opportunity to create some side effects such as: updating the table of symbols, increasing counters, etc. During lexical analysis, the analyzer may match a prefix of the input with the regular expression r_i of any (active) rule.

Lexical analyzers produced by SILex, like many other lexical analyzers, obey some principles when trying to find and select matches. SILex follows the *maximal-munch* (aka, longest-match) *tokenization* principle. It means that when there is a match between prefix w_1 and regular expression r_i that compete with another match between prefix w_2 and expression r_j, such that $|w_1| > |w_2|$,

$$
\begin{array}{rcl}
T([spec], w) & = & \begin{cases} \{w\}, & \text{if } w \in L([spec]) \\ \emptyset, & \text{otherwise} \end{cases} \\
T(\text{"}v\text{"}, w) & = & \begin{cases} \{v\}, & \text{if } w = v \\ \emptyset, & \text{otherwise} \end{cases} \\
T((r), w) & = & T(r, w) \\
T(r^*, w) & = & \rho_T(r, w, 0, \infty) \\
T(r^+, w) & = & \rho_T(r, w, 1, \infty) \\
T(r^?, w) & = & \rho_T(r, w, 0, 1) \\
T(r\{i\}, w) & = & \rho_T(r, w, i, i) \\
T(r\{i,\}, w) & = & \rho_T(r, w, i, \infty) \\
T(r\{i,j\}, w) & = & \rho_T(r, w, i, j) \\
T(r_0 \ldots r_{n-1}, w) & = & \left\{ [t_0, \ldots, t_{n-1}] \;\middle|\; \begin{array}{l} \exists w_0 \in \Sigma^*. \;\ldots\; \exists w_{n-1} \in \Sigma^*. \\ w = w_0 \ldots w_{n-1} \wedge \\ \forall 0 \le i < n.\; t_i \in T(r_i, w_i) \end{array} \right\} \\
T(r_0 \mid \ldots \mid r_{n-1}, w) & = & \{\#i : t \mid 0 \le i < n \;\wedge\; t \in T(r_i, w)\}
\end{array}
$$

where:

$$
\rho_T(r, w, b, B) = \left\{ [t_0, \ldots, t_{n-1}] \;\middle|\; \begin{array}{l} \exists n \in \mathbb{N}.\; b \le n \le B \wedge \\ \exists w_0 \in \Sigma^*. \;\ldots\; \exists w_{n-1} \in \Sigma^*. \\ w = w_0 \ldots w_{n-1} \wedge \\ \forall 0 \le i < n.\; t_i \in T(r, w_i) \end{array} \right\}
$$

Figure 3. Parse trees for a word that matches a regular expression.

then the former match is preferred. SILex also gives priority to first rules. It means that when there is a match between prefix w and expression r_i that compete with another match between w and r_j, such that $i < j$, then the former match is preferred. Note that, although these two principles uniquely determine, for each match, the length of the lexeme and the rule that matches, they say nothing about the parse tree that one obtains for the lexeme. As we see below, a single pair of a regular expression and a word may lead to more than one parse tree. In such a case, the lexical analyzer is free to return any of these.

5. Adapting the construction of parse trees

Before the adapted technique is presented, the notation for the parse trees is introduced and the parse trees for a word according to a regular expression. The following two subsections present the finite-state automata that are at the basis of the construction of parse trees. Finally, we consider the issue of converting the NFA into DFA.

5.1 Syntax of the parse trees

Let us present the syntax of the parse trees. Let $\mathcal{T}$ be the set of all possible parse trees. $\mathcal{T}$ contains basic trees, which are words, and composite trees, which are *selectors* and lists. $\mathcal{T}$ is the smallest set with the following properties.

$$
\begin{array}{ll}
\forall w \in \Sigma^*. & w \in \mathcal{T} \\
\forall i \in \mathbb{N}. \quad \forall t \in \mathcal{T}. & \#i : t \in \mathcal{T} \\
\forall n \ge 0. \quad \forall i \in \mathbb{N} \text{ s.t. } 0 \le i < n. \quad \forall t_i \in \mathcal{T}. & \\
& [t_0, \ldots, t_{n-1}] \in \mathcal{T}
\end{array}
$$

Note that we do not represent parse trees graphically as is customary in presentation of parsing technology. Instead, we use a notation similar to a data structure (to an algebraic data type, to be more specific) to represent them. However, the essence of both representations is the same as the purpose of a parse tree is to serve as an explicit demonstration that a particular word can effectively be generated by a regular expression (or, usually, by a context-free grammar).

In particular, let us recall that if we have a parse tree t for a word w according to a context-free grammar, then we can find all the characters of w, in order, at the leaves of t. We can do the same with our parse trees associated to regular expressions. Let us define an extraction function $X : \mathcal{T} \to \Sigma^*$ that allows us to do so.

$$
\begin{array}{rcl}
X(w) & = & w \\
X(\#i : t) & = & X(t) \\
X([t_0, \ldots, t_{n-1}]) & = & X(t_0) \ldots X(t_{n-1})
\end{array}
$$

5.2 Parse trees for lexemes

We can now describe the parse trees for a word that matches a regular expression. Figure 3 presents the T function. $T(r, w)$ is the set of parse trees that show how w is generated by r. We use the plural form “parse trees” as there may be more than one parse tree for a single expression/word pair. Borrowing from the context-free grammar terminology, we could say that a regular expression may be *ambiguous*.

Note that, once again, we need a repetition function $\rho_T : \mathcal{R} \times \Sigma^* \times \mathbb{N} \times (\mathbb{N} \cup \{\infty\}) \to 2^{\mathcal{T}}$ to help shorten the definitions for the numerous repetition operators. The definition of the repetition function can be found at the bottom of Figure 3.

The meaning of $T(r', w)$, for each form of r', is explained in the following. Some examples are given. Note that, for the sake of brevity, we may use single-character regular expressions such as `a` instead of the equivalent class variants such as [`a`].

- Case $r' = [spec]$. The only valid parse tree, if it exists, is a single character c. c has to be a member of the character class specification and has to be equal to the single character in w. Examples: $T([\texttt{ab}], \texttt{a}) = \{\texttt{a}\}$; $T([\texttt{ab}], \texttt{c}) = \emptyset = T([\texttt{ab}], \texttt{baa})$.

- Case $r' = "v"$. The only valid parse tree is v and it exists if $w = v$. Note that, from the point of view of the parse tree data type, parse tree v is considered to be atomic (or basic), even though, from the point of view of language theory, $v \in \Sigma^*$ may be a composite object. Example: $T(\texttt{"abc"}, \texttt{abc}) = \{\texttt{abc}\}$.
- Case $r' = (r)$. Parentheses are there just to allow the user to override the priority of the operators. They do not have any effect on the parse trees the are generated.
- Cases $r' = r^*$, $r' = r^+$, $r' = r^?$, $r' = r\{i\}$, $r' = r\{i,\}$, and $r' = r\{i,j\}$. The parse trees for w demonstrate how w can be partitioned into n substrings w_0, …, w_{n-1}, where n is legal for the particular repetition operator at hand, and how each w_i can be parsed using r to form a child parse tree t_i, with the set of all the t_i collected into a list. The lists may have varying lengths but the child parse trees they contain are all structured according to the single regular expression r. Example: $T(\texttt{a}\{2,3\}^*, \texttt{aaaaaa}) = \{[[\texttt{a},\texttt{a}],[\texttt{a},\texttt{a}],[\texttt{a},\texttt{a}]],[[\texttt{a},\texttt{a},\texttt{a}],[\texttt{a},\texttt{a},\texttt{a}]]\}$.
- Case $r' = r_0 \dots r_{n-1}$. The parse trees for w demonstrate how w can be partitioned into exactly n substrings w_0, …, w_{n-1}, such that each w_i is parsed according to its corresponding child regular expression r_i. In this case, the lists have constant length but the child parse trees are structured according to various regular expressions. Examples: $T(\texttt{abc}, \texttt{abc}) = \{[\texttt{a},\texttt{b},\texttt{c}]\}$; $T(\texttt{a}^*\texttt{ab}, \texttt{aaab}) = \{[[\texttt{a},\texttt{a}],\texttt{a},\texttt{b}]\}$.
- Case $r' = r_0 \mid \dots \mid r_{n-1}$. A parse tree for w demonstrates how w can be parsed according to one of the child regular expressions. It indicates which of the child expressions (say r_i) matched w and it contains an appropriate child parse tree (for w according to r_i). Example: $T(\texttt{a}^*|(\texttt{aa})^+|\texttt{a}^?\texttt{a}^?, \texttt{a}) = \{\#0 : [\texttt{a}], \#2 : [[\texttt{a}],[\,]], \#2 : [[\,],[\texttt{a}]]\}$.

Function T has some interesting properties. The first one is that parse trees exist only for words that match a regular expression; formally, $T(r,w) \neq \emptyset$ if and only if $w \in L(r)$. The second one is that, from any parse tree for a word according to a regular expression, we can extract the word back; formally, if $t \in T(r,w)$, then $X(t) = w$.

Depending on the regular expression, the "amount" of ambiguity varies. The union operator tends to additively increase the number of different parse trees produced by the child expressions. On the other hand, the concatenation operator tends to polynomially increase the number of different parse trees. Even more extreme, some of the repetition operators tend to increase the number exponentially and even infinitely. Let us give instances of such increases. Let the r_i's be expressions that lead to one or two parse trees for any non-empty word and none for ϵ. Then $r_0 \mid \dots \mid r_{n-1}$, $r_0 \dots r_{n-1}$, $((r_0)^+)^*$, and $((r_0)^*)^*$ produce additive, polynomial, exponential, and infinite increases, respectively.

5.3 Strategy for the construction of parse trees

In the original paper, it is shown how the construction of parse trees for lexemes can be automated. The technique is an extension of Thompson's technique to construct finite-state automata [14]. The extension consists in adding *construction commands* on some of the edges of the automata. Essentially, each time a path through an automaton causes some word to be consumed, then the sequence of commands found along that path forms a "recipe" for the construction of a parse tree for the word.

In general, a parse tree may be an assemblage of many subtrees. These sub-trees cannot all be built at once. They are created one after the other. Consequently, the sub-trees that are already built have to be kept somewhere until they are joined with the other subtrees. It was shown that a data structure as simple as a stack was providing the appropriate facilities to remember and give back parts of a parse tree under construction. All the parse tree construction commands are meant to operate on a stack.

The commands used by the original technique are: "push constant", "wrap in selector", and "extend list". The "push constant" command has a constant tree t as operand and performs the following operation: it modifies the stack it is given by pushing t. The "wrap in selector" command has a number i as operand and performs the following operation: it modifies the stack by first popping a tree t, by building the selector $\#i : t$, and then by pushing $\#i : t$ back. Finally, the "extend list" command has no operand and performs the following operation: it modifies the stack by first popping a tree t and then a list l, by adding *t at the end of l* to form l', and then by pushing l' back.

As explained above, the Scheme language does feature lists but these lists are normally (efficiently) accessed by the front and not by the end. Strictly speaking, Scheme lists *can* be extended efficiently by the end but only in a destructive manner. We prefer to avoid going against the usual programming style used in functional languages and choose to adapt the original technique to make it compatible with the natural right to left construction of lists in Scheme.

This choice to adapt the original technique to build lists from right to left has an effect on the way automata with commands are traversed. In the adapted technique, we have the property that, if a path traverses an automaton forwards and consumes some word, then the sequence of commands found *on the reversed path* forms a recipe to build a parse tree for the word. Thus, the next section presents a technique to build finite-state automata with commands similar to that of the original paper except for the facts that we have a larger set of regular expression operators and that the commands are placed differently in the automata.

5.4 Automata with construction commands

We present the construction rules for the finite-state automata with commands. The construction rules take the form of a procedure A that takes a regular expression r and builds the automaton $A(r)$. A is defined by structural induction on regular expressions. The construction rules are similar to those prescribed by Thompson [14] but with commands added on the edges. The rules are presented in Figures 4 and 5.

Each construction rule produces an automaton with distinguished entry and exit states named p and q, respectively. When an automaton $A(r)$ embeds another one $A(r')$, we depict $A(r')$ as a rectangle with two states which are the entry and exit states of $A(r')$. In each automaton $A(r)$, there is no path going from q to p using edges of $A(r)$ only. In other words, any path from q to p, if it exists, has to go through at least one edge added by a surrounding automaton. The parse tree construction commands are shown using a compact notation. A "push constant" command with operand t is denoted by `push` t. A "wrap in selector" command with operand i is denoted by `sel` i. An "extend list" command is (of course) denoted by `cons`.

We mention, without proof, the few key properties of the automata. Let r be a regular expression and P be a path that traverses $A(r)$ from entry to exit. First, the sequence of commands that are met by following P *backwards* causes exactly one parse tree to be pushed. More precisely, if we take a stack σ and apply on it all the commands that we meet by following P *backwards*, then the net effect of these commands is to push exactly one parse tree t on σ. Second, the automata are correct in the sense that if the word that is consumed along P is w, then $t \in T(r,w)$. Third, the automata are exhaustive with respect to T in the sense that, for any $r \in \mathcal{R}$, $w \in L(r)$, $t \in T(r,w)$, and stack σ, then there exists a path P that traverses $A(r)$, that consumes w, and whose reversed sequence of

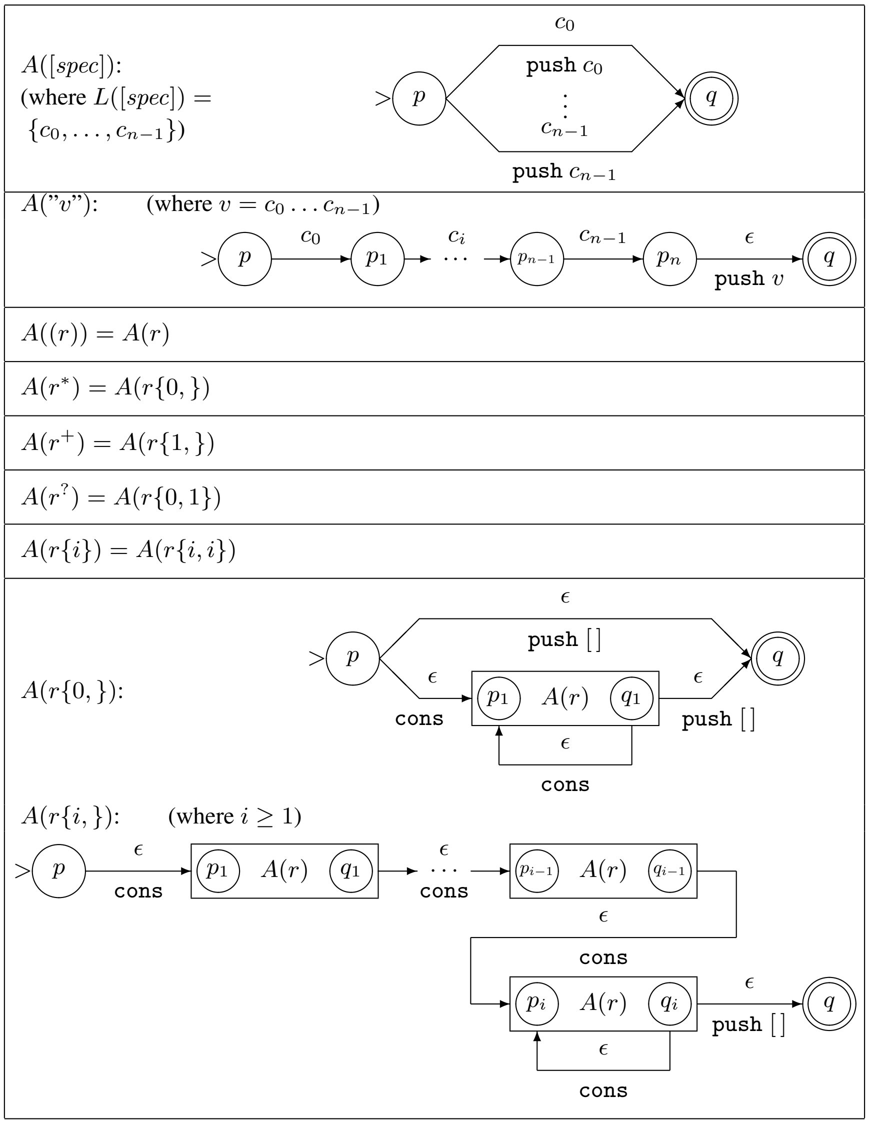


Figure 4. Construction rules for the automata with commands (Part I).

commands causes t to be pushed on σ. These properties can be proved straightforwardly by structural induction on $\mathcal{R}$.

5.5 Using deterministic automata

For efficiency reasons, it is preferable to use a DFA instead of a NFA. As explained above, the NFA obtained using function A may be converted into a DFA to allow fast recognition of the lexemes but three tables have to be built in order to be able to translate paths through the DFA back into paths through the original NFA.

We assume the conversion of the NFA into a DFA to be a straightforward one. We adopt the point of view that deterministic states are sets of non-deterministic states. Then, our assumption says that the deterministic state that is reached after consuming some word w is *exactly* the set of non-deterministic states that can be reached by consuming w.[1]

We may now introduce the three tables Acc, f, and g. Table g indicates how to reach a state q from the non-deterministic start state using only ϵ-transitions. It is defined only for the non-deterministic states that are contained in the deterministic start state. Table f indicates how to reach a non-deterministic state q from some state in a deterministic state s using a path that consumes a single character c. It is usually not defined everywhere. Table Acc indicates, for a deterministic state s, which non-deterministic state in s accepts on behalf of the same rule as s. It is defined only for accepting deterministic states.

Let us have a word $w = c_0 \ldots c_{n-1}$ that is accepted by the DFA and let $P_D = s_0 \ldots s_n$ be the path that is taken when w is consumed. Each s_i is a deterministic state, s_0 is the start state, and s_n is an accepting state. Note that an accepting state does not simply accept, but it accepts on behalf of a certain rule. In fact, an accepting deterministic state may contain more than one accepting

[1] Note that this assumption precludes full minimization of the DFA. SILex currently does not try to minimize the DFA it builds. The assumption is sufficiently strong to ensure that paths through the NFA can be recovered but it may happen to be unnecessarily strong. More investigation should be made to find a sufficient and necessary condition on the conversion.

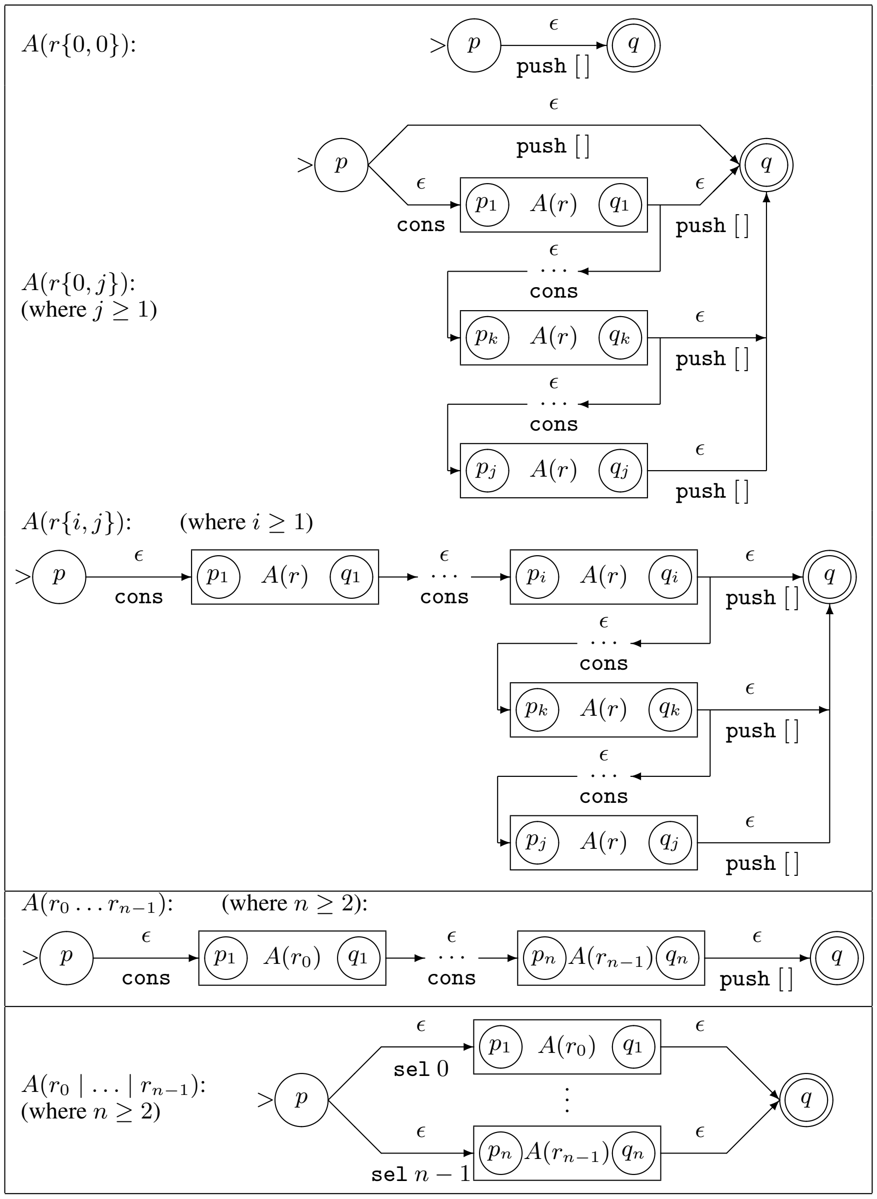


Figure 5. Construction rules for the automata with commands (Part II).

non-deterministic states, each on behalf of its corresponding rule. In such a case, the deterministic state accepts on behalf of the rule that has highest priority. The non-deterministic path P_N that corresponds to P_D is recovered backwards portion by portion. The idea consists in determining non-deterministic states $\{q_i\}_{0 \leq i \leq n}$ and portions of path $\{P_i\}_{0 \leq i \leq n}$ such that: each q_i is in s_i; each P_i starts at q_{i-1}, ends at q_i, and consumes c_{i-1}, except for P_0, which starts at the non-deterministic start state, ends at q_0, and consumes ϵ; q_n is a state that accepts on behalf of the same rule as s_n.

The recovery is initialized by determining q_n directly from s_n using the query $Acc(s_n)$. Next, the main part of the recovery consists in an iteration, with i going from n down to 1. At step i, given q_i, one can determine portion of path P_i and intermediate non-deterministic state q_{i-1}. P_i is obtained from the query $f(s_{i-1}, c_{i-1}, q_i)$. By doing so, q_{i-1} is also obtained as it is the source state of P_i. As the final part of the recovery, P_0 is obtained using the query $g(q_0)$. Then path P_N is simply the linkage of all the portions together; i.e. $P_N = P_0 \cdot \ldots \cdot P_n$.

Note that the preceding explanation contains some minor inaccuracies. First, tables f and g do not exactly contain portions of path but *reversed* ones. Indeed, recall that the NFA presented in this paper are such that commands must be executed in the order in which they are met when following paths backwards. Second, there is no need to recover path P_N (or its reverse) explicitly. It is sufficient to keep references to the portions that form P_N and to later execute the commands by following the portions one after the other. Better yet, one may eagerly execute the commands contained in each portion as the latter gets determined. This way, it is unnecessary to remember P_N nor its portions. Only the current state of the construction stack needs to be preserved. Last, one may observe that the sole purpose of the portions of path stored in tables f and g is to be followed in order to recover the parse tree construction commands. It is possible to skip the step of converting a portion of path into a sequence of commands by directly storing sequences of commands in f and g. It not only saves time by avoiding the conversion but also because sequences of commands can be no longer than the paths from which they are extracted since at most one command gets attached to each arc. One must be careful in the case of table f because a mere sequence of commands would not indicate which non-deterministic state is the origin of the portion of path. Consequently, the latter also has to be returned by f. To recapitulate: a query $g(q)$ provides the sequence of commands that would be met by following some ϵ-consuming path from the non-deterministic start state to q backwards; a query $f(s, c, q)$ provides a pair of a non-deterministic state $q' \in s$ and the sequence of commands that would be met by following some c-consuming path from q' to q backwards.

Remember that some regular expressions are ambiguous. Let r be an ambiguous expression and w a word that has at least two parse trees. We said that, in the context of automatically generated lexical analyzers, it is sufficient to build only one parse tree for w. In other words, from the path P_D that traverses the DFA, it is sufficient to recover only one (P_N) of the corresponding paths that traverse the NFA. Indeed, by the use of fixed tables f, g, and Acc, the recovery of P_N from P_D and w is deterministic. Essentially, the choices among all possible paths are indirectly made when unique values are placed into tables entries that could have received any of numerous valid values. Nevertheless, even if in practice, any single lexical analyzer produces parse trees in a deterministic manner, it remains more convenient to specify the parse tree construction as a non-deterministic process.

6. Modifications to SILex

The addition of parse trees to SILex has little impact on the way SILex is used. The only visible modification is the presence of an additional variable in the scope of the actions. The name of this variable is `yyast`, for Abstract Syntax Tree.[2] An action may refer to this variable as any other variable provided by SILex, such as `yytext`, which contains the lexeme that has just been matched, `yyline`, which contains the current line number, etc.

While the observable behavior of SILex has not changed much, there are many changes that have been made to the implementation of SILex. The most important changes were made in the generation-time modules. First, the original version of SILex used to convert many regular expression operators into simpler forms in order to handle as few native operators as possible. It was doing so during syntactic analysis of the regular expressions. For example, SILex eliminated some forms by converting strings like "v" into concatenations, by breaking complex repetition operators into a combination of simpler ones and concatenations, and by splitting large concatenations and unions into binary ones. While such conversions do not change the language generated by the expressions, they *do* change the set of valid parse trees for most or all words. The new version has to represent most syntactic forms as they appear in the specification files. Still, there are now new opportunities to translate simple forms, such as r^*, r^+, $r^?$, and $r\{i\}$, into the more general forms $r\{b, B\}$, which have to be supported anyway.

Second, the construction rules for the automata have been changed to correspond to the new list of syntactic forms and to conform to the specifications of Figures 4 and 5. Of course, the representation of the arcs (in the NFA) had to be extended so that commands could be attached.

[2] Actually, we consider the name `yyast` to be rather inappropriate as the parse trees that the new variable contains are indeed *concrete* syntax trees. Still, since the version of SILex that we are working on uses that name, we prefer to stick to the current conventions.

Third, a phase which used to clean up the NFA between the elimination of the ϵ-transitions and the conversion of the NFA into a DFA has been eliminated. It eliminated useless states and renumbered the remaining states. The modification of the numbers interfered with the construction of the three new tables and the quick and dirty solution has been to completely omit the phase. The generated analyzers would benefit from the re-introduction of the clean-up phase and, in order to do so, some adaptation should be made to the currently abandoned phase or to the implementation of the table construction.

Fourth, we added the implementation of the construction and the printing of the three tables. The construction of the tables mainly consists in extracting reachability information from the graph of the NFA.

The next modifications were made to the analysis-time module. Fifth, the lexical analyzers had to be equipped with instrumentation to record the paths that are followed in the DFA. Also, requests for the construction of parse trees when appropriate have been added.

Sixth, we included the functions that build parse trees when they are given a path through the DFA, the recognized lexeme, and the three tables.

Up to this point, the modifications aimed only at providing the parse tree facility when the tables of the DFA are represented using the ordinary format. So, at last, we modified both the generation-time and the analysis-time modules so that parse trees could also be built when the DFA is represented using portable tables or Scheme code. In the case of the portable tables, it required only the creation of simple conversion functions to print a portable version of tables f and g at generation time and to translate the portable tables back into the ordinary format at analysis time. In the case of the DFA as Scheme code, the modifications are more complex as extra code must be emitted that takes care of the recording of the path through the DFA and the requests for the construction of parse trees. Note that the functions that perform the very construction of the parse trees are the same no matter which format for the tables of the DFA is used. It means that the construction of parse trees is an interpretative process (based on queries to the three tables), even when the DFA is implemented efficiently as code.

Note that, although SILex gives the impression that parse trees are always available to actions, SILex is lazy with their construction. It builds them only for the actions that *seem* to access the variable `yyast`. The path followed into the DFA is always recorded, however. Still, SILex's laziness substantially reduces the extra cost caused by the addition of the parse trees as most of it comes from the construction of trees, not the recording of paths.

The current state of the prototype is the following. The integration is complete enough to work but the code needs a serious clean-up. The three additional tables for DFA to NFA correspondence are much too large. The implementation of the mechanisms for path recording and parse tree construction is not really optimized for speed.

7. Examples of parse tree construction for lexemes

We present a few concrete examples of the use of parse tree construction using SILex. We first start by describing the Scheme representation of the parse trees.

7.1 Representation of parse trees in Scheme

The representation of trees in $\mathcal{T}$ in Scheme is direct. A list tree $[t_0, \ldots, t_{n-1}]$ becomes a Scheme list (S_0 ... S_{n-1}) where each S_i is the Scheme representation of t_i. Next, a selector $\#i : t$ also becomes a Scheme list (i S) where i remains the same and S corresponds to t. Finally, a word w may take two forms in Scheme.

If w is a parse tree that originates from a string regular expression "w", then it becomes a Scheme string "w", otherwise w is necessarily one-character long and it becomes a Scheme character #\w.

7.2 Simple examples

Let us consider the following short SILex specification file:

```
%%
a{2,4}  (list 'rule1 yyast)
a{0,3}  (list 'rule2 yyast)
```

where only some sequences of a are deemed to be legal tokens and where the actions simply return tagged lists containing the parse trees that are produced. If we generate a lexical analyzer from this specification file and ask it to analyze the input aaaaa, then it will produce the following two results before returning the end-of-file token:

```
(rule1 (#\a #\a #\a #\a))
(rule2 (#\a))
```

Both parse trees indicate that the matched lexemes were made of repetitions of the character a, which is consistent with the shape of the regular expressions. Note how the first token had to be as long as possible, following the maximal-munch tokenization principle.

Now, let us consider a more complex example. The following specification file allows the analyzer-to-be to recognize Scheme strings:

```
%%
\"([^"\\]|"\\\""|"\\\\")*\"  yyast
```

One must not forget about the necessary quoting of special characters " and \. If we feed the analyzer generated from this specification with the following Scheme string:

```
"Quote \" and \\!"
```

then the analyzer returns a parse tree that denotes a sequence of three sub-trees, where the middle one is a sequence of 14 sub-sub-trees, where each is a selector among the three basic string elements:

```
(#\"
 ((0 #\Q) (0 #\u) (0 #\o) (0 #\t) (0 #\e)
  (0 #\space) (1 "\\\"") (0 #\space)
  (0 #\a) (0 #\n) (0 #\d) (0 #\space)
  (2 "\\\\") (0 #\!))
 #\")
```

These two examples may not be that convincing when it comes to justifying the implementation of automatic construction of parse trees for lexemes. However, the one below deals with a regular expression that is way more complex.

7.3 Lexical analysis of Scheme numbers

Scheme provides a particularly rich variety of numbers: from integers to complex numbers. It also provides a "syntax" for the external representation of all these kinds of numbers. An implementor has much work to do in order to handle all the kinds of numbers. In particular, when it comes to *reading* them. There are so many cases that reading them in an *ad hoc* way tends to be error-prone.

Even when one automates part of the process by using an automatically generated lexical analyzer to scan Scheme numbers, only half of the problem is solved. Indeed, merely knowing that a lexeme is the external representation of a Scheme number does not provide any easy way to recover the internal representation from the lexeme. That is, it is not easy unless the lexical analyzer is able to provide a parse tree for the lexeme. In Figure 6, we present a relatively complete specification for the Scheme numbers. Note that we restrict ourselves to numbers in base 10 only and that we do not handle the unspecified digits denoted by #. The specification file is mostly made of macros and there is a single rule which takes the parse tree for the number and passes it to a helper function.

The helper function is very simple as it only has to traverse the tree and *rebuild* the number. This reconstruction is made easy by the fact that the hierarchical structure of the lexeme according to the regular expression is clearly exposed and that any "choice" between various possibilities is indicated by the tree. Figure 7 presents the implementation of our helper function, which is less than one hundred lines of very systematic code. The reader needs not necessarily study it closely—the font is admittedly pretty small—as the main point here is to show the size and the shape of the code. If we were to complete our implementation to make it able to handle the full syntax, it would be necessary to add many macros in the specification file but the helper function would not be affected much.

8. Experimental results

In order to evaluate the cost of the construction of parse trees, we ran a few experiments. The experiments consist in analyzing the equivalent of 50 000 copies of the following 10 numbers (as if it were a giant 500 000-line file).

```
32664
-32664
32664/63
+32664/63
-98327E862
+i
-453.3234e23+34.2323e1211i
+.32664i
-3266.4@63e-5
+32664/63@-7234.12312
```

We used three different lexical analyzers on the input. The first one is a lexical analyzer generated by the original version of SILex. The second one is generated by the new version of SILex and build a parse tree for each of the recognized lexemes. The third one is also generated using the new version of SILex but it does not ask for the construction of the parse trees (i.e. the action does not access yyast). This last analyzer is used to evaluate the cost of the instrumentation added to record the path through the DFA.

The lexical analyzers have been generated by (either version of) SILex to be as fast as possible; that is, their DFA is implemented as Scheme code and they maintain no counters to indicate the current position in the input. The lexical analyzers have been compiled using Gambit-C version 3.0 with most optimizations turned on. The resulting C files have been compiled using GCC version 3.3.5 with the '-O3' switch. The analyzers were executed on a 1400 MHz Intel Pentium 4 processor with 512 MBytes of memory.

The execution times for the three analyzers are 15.3 seconds, 39.8 seconds, and 20.2 seconds, respectively. Clearly, building the parse trees incurs a serious cost as the execution time almost triples. This is not that surprising given the complexity of building a parse tree compared to the simplicity of a mere recognition using a DFA. However, the third measurement indicates that the added instrumentation causes the operations of the DFA to take significantly longer. The increase is about by a third. While the increase is much less than in the case of parse tree construction, it is still less acceptable. Construction of parse trees can be seen as a sophisticated operation that is relatively rarely performed. One might accept more easily to pay for a service that he does use. However, the extra cost due to the instrumentation is a cost without direct benefit and that

```
; Regular expression for Scheme numbers
; (base 10 only, without '#' digits)

digit           [0-9]
digit10         {digit}
radix10         ""|#[dD]
exactness       ""|#[iI]|#[eE]
sign            ""|"+"|"-"
exponent_marker [eEsSfFdDlL]
suffix          ""|{exponent_marker}{sign}{digit10}+
prefix10        {radix10}{exactness}|{exactness}{radix10}
uinteger10      {digit10}+
decimal10       ({uinteger10}|"."{digit10}+|{digit10}+"."{digit10}*){suffix}
ureal10         {uinteger10}|{uinteger10}/{uinteger10}|{decimal10}
real10          {sign}{ureal10}
complex10       {real10}|{real10}@{real10}|{real10}?[-+]{ureal10}?[iI]
num10           {prefix10}{complex10}
number          {num10}

%%

{number}        (lex-number yyast)
```

Figure 6. SILex specification for the essentials of the lexical structure of Scheme numbers.

```
; Companion code for Scheme numbers

(define lex-number
  (lambda (t)
    (let* ((digit
            (lambda (t)
              (- (char->integer t) (char->integer #\0))))
           (digit10
            (lambda (t)
              (digit t)))
           (exactness
            (lambda (t)
              (case (car t)
                ((0)  (lambda (x) x))
                ((1)  (lambda (x) (* 1.0 x)))
                (else (lambda (x) (if (exact? x) x (inexact->exact x)))))))
           (sign
            (lambda (t)
              (if (= (car t) 2)
                  -1
                  1)))
           (digit10+
            (lambda (t)
              (let loop ((n 0) (t t))
                (if (null? t)
                    n
                    (loop (+ (* 10 n) (digit10 (car t))) (cdr t))))))
           (suffix
            (lambda (t)
              (if (= (car t) 0)
                  0
                  (let ((tt (cadr t)))
                    (* 1.0
                       (sign (list-ref tt 1))
                       (digit10+ (list-ref tt 2)))))))
           (prefix10
            (lambda (t)
              (exactness (list-ref (cadr t) (- 1 (car t))))))
           (uinteger10
            (lambda (t)
              (digit10+ t)))
           (decimal10
            (lambda (t)
              (let* ((e2 (suffix (list-ref t 1)))
                     (tt (list-ref t 0))
                     (ttt (cadr tt)))
                (case (car tt)
                  ((0)
                   (* (digit10+ ttt) (expt 10 e2)))
                  ((1)
                   (let ((tttt (list-ref ttt 1)))
                     (* (digit10+ tttt) (expt 10.0 (- e2 (length tttt))))))
                  (else
                   (let* ((tttt1 (list-ref ttt 0))
                          (tttt2 (list-ref ttt 2)))
                     (* (digit10+ (append tttt1 tttt2))
                        (expt 10.0 (- e2 (length tttt2))))))))))
           (ureal10
            (lambda (t)
              (let ((tt (cadr t)))
                (case (car t)
                  ((0)
                   (uinteger10 tt))
                  ((1)
                   (/ (uinteger10 (list-ref tt 0))
                      (uinteger10 (list-ref tt 2))))
                  (else
                   (decimal10 tt))))))
           (real10
            (lambda (t)
              (* (sign (list-ref t 0)) (ureal10 (list-ref t 1)))))
           (opt
            (lambda (op t default)
              (if (null? t)
                  default
                  (op (list-ref t 0)))))
           (complex10
            (lambda (t)
              (let ((tt (cadr t)))
                (case (car t)
                  ((0)
                   (real10 tt))
                  ((1)
                   (make-polar (real10 (list-ref tt 0))
                               (real10 (list-ref tt 2))))
                  (else
                   (make-rectangular
                    (opt real10 (list-ref tt 0) 0)
                    (* (if (char=? (list-ref tt 1) #\+) 1 -1)
                       (opt ureal10 (list-ref tt 2) 1))))))))
           (num10
            (lambda (t)
              ((prefix10 (list-ref t 0))
               (complex10 (list-ref t 1)))))
           (number
            (lambda (t)
              (num10 t))))
      (number t))))
```

Figure 7. Implementation of a helper function for the lexical analysis of numbers.

one cannot get rid of, even when parse tree construction is almost never used.

9. Discussion

As far as we know, the original technique is the only one that makes automatically generated lexical analyzers able to build parse trees for lexemes using only finite-state tools and this work is the only implementation of it.

Generated lexical analyzers always give access to the matched lexemes. It is essential for the production of tokens in lexical analysis. To also have access to information that is automatically extracted from the lexemes is a useful feature. However, when such a feature is provided, it is typically limited to the ability to extract sub-lexemes that correspond to tagged (e.g. using `\(` and `\)`) sub-expressions of the regular expression that matches the lexeme. Techniquely, for efficiency reasons, it is the *position* and the *length* of the sub-lexemes that get extracted. The IEEE standard 1003.1 describes, among other things, which sub-lexemes must be extracted [7]. Ville Laurikari presents an efficient technique to extract sub-lexemes in a way that complies with the standard [11]. In our opinion, extraction by tagging is too restrictive. The main problem is that, when a tagged sub-expression lies inside of a repetition operators (or inside of what is sometimes called a *non-linear* context) and this sub-expression matches many different parts of a given lexeme, only one of the sub-lexemes is reported. So extraction by tagging starts to become ineffective exactly in the situations where the difficulty or the sophistication of the extraction would make automated extraction most interesting.

Since the conventional way of producing parse trees consists in using a syntactic analyzer based on context-free grammar technology, one might consider using just that to build parse trees for his lexemes. For instance, one could identify lexemes using a DFA and then submit the lexemes to a subordinate syntactic analyzer to build parse trees. Alternatively, one could abandon finite-state technology completely and directly use a *scanner-less* syntactic analyzer. However, both options suffer from the fact that analyzers based on context-free grammars are much slower than those based on finite-state automata. Moreover, an ambiguous regular expression would be translated into an ambiguous context-free grammar. Our technique handles ambiguous expressions without problem but most parsing technology cannot handle ambiguous grammars. Of course, there exist parsing techniques that can handle ambiguous grammars, such as Generalized LR Parsing [10, 15], the Earley algorithm [5], or the CYK algorithm [8, 17, 2], but these exhibit worse than linear time complexity for most or all ambiguous grammars. Finally, it is possible to translate any regular expression into an unambiguous left- or right-linear grammar [6]. However, the resulting grammar would be completely distorted and would lead to parse trees that have no connection to the parse trees for lexemes that we introduced here.

10. Future work

- We intend to complete the integration of automatic parse tree construction into SILex and to clean up the whole implementation.
- Parse tree construction could be made faster. In particular, when the DFA is represented as Scheme code, the functions that build the trees ought to be specialized code generated from the information contained in the three tables.
- The penalty that is strictly due to the additional instrumentation (i.e. when no parse trees are requested) ought to be reduced. A way to improve the situation consists in marking the deterministic states that may reach an accepting state that corresponds to a rule that requests the construction of a parse tree. Then, for states that are *not* marked, the instrumentation that records the path in the DFA could be omitted.
- All tables generated by SILex ought to be compacted but the one for f, in particular, really needs it. Recall that f takes a three-dimensional input and returns a variable-length output (a pair that contains a sequence of commands).
- Some or all of the following regular operators could be added to SILex: the difference (denoted by, say, $r_1 - r_2$), the complement ($\overline{r}$), and the intersection ($r_1 \& r_2$). Note that, in the case of the complement operator, there would be no meaningful notion of a parse tree for a lexeme that matches $\overline{r}$. In the case of the difference $r_1 - r_2$, the parse trees for a matching lexeme w would be the demonstration that r_1 generates w. Finally, in the case of the intersection, for efficiency reasons, only one of the sub-expressions should be chosen to be the one that dictates the shape of the parse trees.
- The parse trees act as (too) detailed demonstrations. Almost always, they will be either transformed into a more convenient structure, possibly with unnecessary details dropped, or completely consumed to become non-structural information. In other words, they typically are transient data. Consequently, it means that only their informational contents were important and that they have been built as concrete data structures to no purpose. In such a situation, deforestation techniques [16] could be used so that the consumer of a parse tree could virtually traverse it even as it is virtually built, making the actual construction unnecessary.

11. Conclusion

This paper presented the adaptation and the implementation of the automated construction of parse tree for lexemes. The technique that has been adapted was originally presented in 2000 by Dubé and Feeley. It has been implemented and integrated in SILex, a lexical analyzer generator for Scheme.

The adaptation was a simple step as it consisted only in modifying the automaton construction rules of the original technique so that the larger set of regular operators of SILex was handled and so that the way the parse trees are built match the right-to-left direction in which lists are built in Scheme.

The implementation was a much more complicated task. Fortunately, SILex, like the original technique, is based on the construction of non-deterministic automata that get converted into deterministic ones. Still, most parts of the generator had to be modified more or less deeply and some extensions also had to be made to the analysis-time module of the tool. Modifications have been done to the representation of the regular expressions, to the way the non-deterministic automata are built and represented, to the conversion of the automata into deterministic ones, to the printing of SILex's tables, to the generation of Scheme code that forms parts of the lexical analyzers, to the algorithm that recognize lexemes, and to the (previously inexistent) construction of the parse trees.

The new version of SILex does work, experiments could be run, but the implementation is still somehow disorganized. The construction of parse trees is a pretty costly operation compared to the normal functioning of a deterministic automaton-based lexical analyzer and, indeed, empirical measurements show that its intensive use roughly triples the execution time of an analyzer.

Acknowledgments

We wish to thank the anonymous referees whose comments really helped to improve this paper.

References

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

[2] J. Cocke and J. T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[3] D. Dubé. Scheme Implementation of Lex, 2001. `http://www.iro.umontreal.ca/~dube/silex.tar.gz`.

[4] D. Dubé and M. Feeley. Efficiently building a parse tree from a regular expression. *Acta Informatica*, 37(2):121–144, 2000.

[5] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, feb 1970.

[6] J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages and computation*. Addison-Wesly Publishing Company, 1979.

[7] IEEE std 1003.1, 2004 Edition.

[8] T. Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Lab, Bedford, MA, USA, 1965.

[9] R. Kelsey, W. Clinger, and J. Rees (eds.). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, aug 1998.

[10] B. Lang. Deterministic techniques for efficient non-deterministic parsers. In *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*, pages 255–269, London, UK, 1974. Springer-Verlag.

[11] Ville Laurikari. NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *Proceedings of the 7th International Symposium on String Processing and Information Retrieval*, pages 181–187, sep 2000.

[12] M. E. Lesk. Lex—a lexical analyzer generator. Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ, USA, 1975.

[13] J. Levine, T. Mason, and D. Brown. *Lex & Yacc*. O'Reilly, 2nd edition, 1992.

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

[15] M. Tomita. Efficient parsing for natural languages. *A Fast Algorithm for Practical Systems*, 1986.

[16] P. L. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

[17] D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.