SILex

A Scheme Implementation of Lex
Documentation for SILex version 1.0

Danny Dubé

1 Overview

SILex is a lexical analyser generator similar to the Lex and Flex programs, but for Scheme. “SILex” stands for “Scheme Implementation of Lex”.

SILex has many similarities with the C programs, but has many differences, too. The syntax of the specification files for SILex is close to that of Lex and Flex. Of course, the actions must be written in Scheme and not in C. The set of regular expressions is mostly the same. An important difference is relative to the multiple start states in the C analysers. SILex replaces them by allowing multiple analysers to take their input from the same source. Different inputs can be analysed at the same time, possibly with different instances of one or more lexical analysers. The analysers are created dynamically.

SILex provides many other features. The designer of a lexical analyser can specify the actions to be taken when the end of file is reached or when an error occurs. The analyser can keep track of the position in the input in terms of the number of the line, column and offset. An analyser can take its input from an input port, a string or a function. SILex is portable; it does not depend on a particular character set. It can generate analysers that are portable, too. Finally, the table encoding the behavior of the analyser can be compiled to Scheme code. The fastest lexical analysers can be produced this way.

2 Syntax of the specification file

A specification file for a lexical analyser contains two parts: the *macro definitions part* and the *rules part*. The two parts are separated by the mark `%%`. The first part is used to define *macros*; that is, to give names to some regular expressions. The second part is used to indicate the regular expressions with which the input will have to match and the *actions* associated with each expression.

Comments can be inserted any place where white space is allowed and is considered as white space itself. The syntax of the comments is the same as in Scheme. That is, it begins with a semicolon '`;`' and extends up to the end of a line. The semicolon is a valid token in many languages, so you should take care not to comment out an entire line when you write a regular expression matching a semicolon.

The syntax of each part is presented, except for the regular expressions, which are described apart. A small example follows.

2.1 Macro definitions part

The first part of a specification file contains zero or more macro definitions. A definition consists of a name and a regular expression, separated by white space. It looks better when each definition is written on a separate line.

The syntax for a macro name is that of a Scheme symbol. The case of the letters is not significant. For example, `abcd`, `+`, `...`, `Digit` and `digit` are all valid macro names; the last two being the same. You cannot write two macro definitions with the same name.

The defined macro can be referenced in regular expressions using the syntax `{`*`name`*`}` (see Section 2.3 [Regular expressions], page 3). The scope of a macro definition includes the remaining definitions and the rules part of the file. It is analogous to the `let*` is Scheme, where the macro definitions correspond to the bindings and the rules part correspond to the body.

End the macro definitions part with `%%`.

2.2 Rules part

The rules part contains the rules up to the end of the specification file. Each rule is a *pattern* optionally followed by an *action*. The pattern is a regular expression. The action, if there is one, is formed of one or more Scheme expressions.

The actions can span over several lines. To distinguish between the remaining of the current action and the start of a new rule, SILex checks the indentation. A new rule must start at the beginning of the line. That is, the action starts right after the pattern and contains all the following lines that start with white space.

SILex does not parse the actions. It simply captures the text up to the start of the next rule. So a syntax error in an action is not detected by SILex.

Nevertheless, SILex is able to detect that an action has been omitted. In that case, a default action is supplied.

2.3 Regular expressions

We first describe the atomic regular expressions. Then, we show how to build more complex regular expressions from simpler ones. Finally, the markers are introduced.

The following constructs are regular expressions:

`c` *Ordinary character*. It is a regular expression that matches the character *c* itself. *c* cannot be one of '`.`', '`\`', '`{`', '`"`', '`[`', '`|`', '`?`', '`+`', '`*`', '`(`', '`)`', '`^`', '`$`', '`;`' or any white space.

`.` *Wild card*. It matches any character except the newline character.

`\n`
`\integer`
`\c` *Backslash*. The backslash is used for two things: protect a character from special meaning; generating non-printable characters. The expression `\n` matches the newline character. The expression `\integer` matches the character that has number *integer* (in the sense of `char->integer`). *integer* must be a valid character number on the implementation that you use. It may be more than 3 digits long and even negative[1]. The expression `\c` matches the character *c* if *c* is not '`n`', '`-`' nor a digit.

`{name}` *Macro reference*. This expression matches the same lexemes as those matched by the regular expression named *name*. You can imagine that the reference is replaced by the text of the named expression. However, it works as if parentheses had been added to protect the substituting expression.

`"some text"`
String. A string matches a lexeme identical to its contents. In a string, the only special characters are '`"`', which closes the string, and '`\`' which keeps the effect mentioned above.

`[list of characters]`
`[]list of characters]`
`[-list of characters]`
`[^list of characters]`
Character class. The expression matches one of the enumerated characters. For example, the expression '`[abc]`' matches one of '`a`', '`b`' and '`c`'. You can list a range of characters by writing the first character, the '`-`' and the last character. For example, '`[A-Za-z]`' matches one letter (if the letters are ordered and contiguous in the character set used by your implementation). The special characters in a class are '`]`', which closes the class, '`-`', which denotes a range of character, and '`\`', which keeps its usual meaning. There is an exception with the first character in a class. If the first character is '`]`' or '`-`', it loses its special meaning. If the first character is '`^`', the expression matches one character if it is *not* enumerated in *list of characters*.

Suppose *r* and *s* are regular expressions. Then the following expressions can be built:

[1] The Scheme standards do not impose a particular character set, such as ASCII. The only requirement is that the function `char->integer` returns an integer.

`r|s` *Union*. This regular expression matches a lexeme if the lexeme is matched by *r* or by *s*.

`rs` *Concatenation*. This expression matches a lexeme if the lexeme can be written as the concatenation of a lexeme matched by *r* and a lexeme matched by *s*.

`r?` *Optional expression*. A lexeme matches this expression if it is the empty lexeme or if it matches *r*.

`r+` *Positive closure*. This expression matches a lexeme that can be written as the concatenation of one or more lexemes, where each of those matches *r*.

`r*` *Kleene closure*. A lexeme is matched by this expression if it can be written as the concatenation of zero or more lexemes, where each of those matches *r*.

`r{i}`
`r{i,}`
`r{i,j}` *Power or repetition of an expression*. These expressions allow the "repetition" of a regular expression a certain number of times. *i* and *j* must be positive integers and *j* must be greater or equal to *i*. The first form repeats the expression *r* exactly *i* times. The second form repeats *r* at least *i* times. The last form repeats *r* at least *i* times and at most *j* times. You should avoid using large numbers (more than 10), because the finite automaton for *r* is copied once for each repetition. The tables of the analyser may quickly become very large. You should note that the syntax of these expressions does not conflict with the syntax of the macro reference.

`(r)` *Parentheses*. This expression matches the same lexemes as *r*. It is used to override the precedence of the operators.

The building operators are listed in order of increasing precedence. The `?`, `+`, `*` and repetition operators have the same precedence.

The remaining "expressions" would better be called *markers*. They all match the empty lexeme but require certain conditions to be respected in the input. They cannot be used in all regular expressions. Suppose that *r* is a regular expression without markers.

`^r`
`r$` *Beginning and end of line*. These markers require that the lexeme is found at the beginning and at the end of the line, respectively. The markers lose their special meaning if they are not placed at their end of the regular expression or if they are used in the first part of the specification file. In those cases, they are treated as regular characters.

`<<EOF>>` *End of file*. This marker is matched only when the input is at the end of file. The marker must be used alone in its pattern, and only in the second part of the file. There can be at most one rule with this particular pattern.

`<<ERROR>>`
Error. This marker is matched only when there is a parsing error. It can be used under the same conditions as `<<EOF>>`.

White space ends the regular expressions. In order to include white space in a regular expression, it must be protected by a backslash or placed in a string.

2.4 An example of a specification file

Here is an example of a SILex specification file. The file is syntactically correct from the SILex point of view. However, many common mistakes are shown. The file is not a useful one.

```
; This is a syntactically correct but silly file.

partial     hel
complete    {partial}lo            ; Backward macro ref. only
digit       [0-9]
letter      [a-zA-Z]

%%

-?{digit}+    (cons 'integer yytext)   ; yytext contains
                                       ; the lexeme
-?{digit}+\.{digit}+[eE][-+]?{digit}+
              (cons                ; A long action
               'float
               yytext)

;             (list 'semicolon)    ; Probably a mistake

begin         )list 'begin(        ; No error detected here
end                                ; The action is optional

\73           (list 'bell-3)       ; It does not match the
                                   ; char. # 7 followed by '3'
\0073         (list 'bell-3)       ; Neither does it
(\7)3         (list 'bell-3)       ; This does it

"*()+|{}[].? are ordinary but \" and \\ are special"

[^\n]         (list 'char)         ; Same thing as '.'
({letter}|_)({letter}|_|{digit})*  ; A C identifier
[][]                               ; One of the square brackets

Repe(ti){2}on   (list 'repetition)

^{letter}+:   (cons 'label yytext) ; A label placed at the
                                   ; beginning of the line
$^                                 ; No special meaning
<<EOF>>       (list 'eof)          ; Detection of the end of file
<<ERROR>>     (my-error)           ; Error handling
```

3 Semantics of the specification file

An important part of the semantics of a specification file is described with the syntax of the regular expressions. The remainder is presented here. We begin with the role of the actions. Information on the matching method follows.

3.1 Evaluation of the actions

The action of a rule is evaluated when the corresponding pattern is matched. The result of its evaluation is the result that the lexical analyser returns to its caller.

There are a few local variables that are accessible by the action when it is evaluated. Those are `yycontinue`, `yygetc`, `yyungetc`, `yytext`, `yyline`, `yycolumn` and `yyoffset`. Each one is described here:

`yycontinue`
: This variable contains the lexical analysis function itself. Use `(yycontinue)` to ask for the next token. Typically, the action associated with a pattern that matches white space is a call to `yycontinue`; it has the effect of skipping the white space.

`yygetc`
`yyungetc`
: These variables contain functions to get and unget characters from the input of the analyser. They take no argument. `yygetc` returns a character or the symbol '`eof`' if the end of file is reached. They should be used to read characters instead of accessing directly the input port because the analyser may have read more characters in order to have a look-ahead. It is incorrect to try to unget more characters than has been gotten since *the parsing of the last token*. If such an attempt is made, `yyungetc` silently refuses.

`yytext`
: This variable is bound to a string containing the lexeme. This string is guaranteed not to be mutated. The string is created only if the action 'seems' to need it. The action is considered to need the lexeme when '`yytext`' appears somewhere in the text of the action.

`yyline`
`yycolumn`
`yyoffset`
: These variables indicate the position in the input at the beginning of the lexeme. `yyline` is the number of the line; the first line is the line 1. `yycolumn` is the number of the column; the first column is the column 1. It is important to mention that characters such as the tabulation generate a variable length output when they are printed. So it would be more accurate to say that `yycolumn` is the number of the first character of the lexeme, starting at the beginning of the line. `yyoffset` indicates the distance from the beginning of the input; the first lexeme has offset 0. The three variables may not all be existant depending on the kind of counting you want the analyser to do for you (see Section 4.3.1 [Counters], page 11).

There is a default action that is provided for a rule when its action is omitted. If the pattern is '`<<EOF>>`', the default action returns the object '`(0)`'. If the pattern is

'<<ERROR>>', the default action displays an error message and returns the symbol 'error'[1]. The default action for the other patterns is to call the analyser again. It is clearer (and normally more useful) to specify explicitly the action associated with each rule.

3.2 Matching the rules

Each time the analyser is asked to return a token, it tries to match a prefix of the input with a pattern. There may be more than one possible match; when it is the case, we say there is a conflict. For example, suppose we have those regular expressions:

```
begin
[a-z]*
```

and the input is 'beginning1 . . .'. We have a match with the first expression and we have many different matches with the second. To resolve such a conflict, the longest match is chosen. So the chosen match is the one between the lexeme 'beginning' and the second expression.

Suppose we have the same regular expressions but the input is 'begin+ . . .'. We have *two* longest match. This conflict is resolved by choosing the first pattern that allows a longest match. So the chosen match is between the lexeme 'begin' and the first pattern.

The analyser generated by SILex allows the empty lexeme to be matched if there is no longer match. However, you should take care not to call the analyser again without consuming at least one character of the input. It would cause an infinite loop.

The pattern '<<EOF>>' is matched when the analyser is called and the input is at end of file. In this situation, the marker is matched even if there is a pattern that matches the empty lexeme. The analyser can be called again and again and the '<<EOF>>' pattern will be matched each time, causing its corresponding action to be evaluated each time, too.

The pattern '<<ERROR>>' is matched when the input is not at end of file and no other match is possible. Depending on the action associated with this pattern, your program may choose to stop or choose to try to recover from the error. To recover from the error, your program has to read some characters from the input before it can call the analyser again.

All lexical analysers generated by SILex are interactive. That is, they read as few characters as possible to get the longest match. This is a useful property when the input is coming from a terminal. A lexical analyser is normally based on a finite automaton; it is the case for the analysers generated by SILex. A non-interactive analyser always needs an extra character to provoke an invalid transition in the automaton. The longest match is detected this way. With an interactive analyser, an extra character is not required when it is impossible to obtain a longer match.

A lexical analyser generated by SILex does not impose any *a priori* limit on the size of the lexemes. The internal buffer is extended each time it is necessary.

[1] Note that there is no portable way for the analyser to end the execution of the program when an error occurs.

4 Generating and using a lexical analyser

The most common use of SILex is to generate a single complete lexical analyser. In some situations however, it is preferable to only generate the tables describing the analysers and leaving to the program to build complete analysers at run time. It is the case when the program has to parse many files simultaneously with the same analyser; and when a file is to be parsed using many different analysers. After the description of the two modes, we describe the SILex options and the different input methods.

4.1 One complete analyser

The function `lex` generates a complete lexical analyser. We first describe its parameters. Then the interface with the generated analyser is presented.

4.1.1 The `lex` command

Here is the template of a call to `lex`:

```
(lex input-file output-file [options ...])
```

input-file is a string containing the name of the specification file. *output-file* is a string containing the name of the file in which the lexical analyser is written. For a description of the options, see Section 4.3 [Options], page 11.

This is an example of a call to `lex`:

```
(lex "pascal.l" "pascal.l.scm")
```

4.1.2 The functions in the lexical analyser

The file generated by `lex` contains a few global definitions. A program using the analyser needs only the following functions: `lexer`, `lexer-get-line`, `lexer-get-column`, `lexer-get-offset`, `lexer-getc`, `lexer-ungetc` and `lexer-init`.

`lexer` The lexical analysis function.

`lexer-get-line`
`lexer-get-column`
`lexer-get-offset`
Functions to obtain the current position in the input.

`lexer-getc`
`lexer-ungetc`
Reading and returning characters. These functions have the advantage of being accessible from outside the actions.

`lexer-init`
Initializing the analyser with the input source.

To avoid name conflicts, these variables and others that we did not mention all begin with ‘`lexer...`’.

4.1.3 Using the lexical analyser

The first function that must be called is the initialization function. It is necessary to give to the analyser its source of characters. Here is the template of a call to this function:

`(lexer-init` *`input-type input`*`)`

The values *input-type* and *input* are described in Section 4.4 [Input], page 13.

Once the initialization is done, the program can get *tokens* from the analyser by calling the lexical analysing function:

```
(lexer)
```

The token is the result of the evaluation of the action corresponding to the matched pattern. The current position can be obtained with:

```
(lexer-get-line)
(lexer-get-column)
(lexer-get-offset)
```

As is described in Section 4.3 [Options], page 11, some or all of these functions may not be available. Characters can be gotten and ungotten from the input this way:

```
(lexer-getc)
(lexer-ungetc)
```

It is important to note that the analyser remembers the characters previously gotten. Your program does not have to keep those itself.

Even after the end of file has been reached or an error has occured, the `lexer` function can be called again. Its behavior depends on the remaining characters in the input.

The analyser can be reinitialized in any time with a new input.

4.2 Many analysers

There are applications where it is necessary to have more than one lexical analyser parsing more than one file at a time. For example:

- The parsing of a C file (with cpp) may cause the parsing of other files recursively because of the `#include` commands.
- An interactive compiler has to be able to compile a file without closing the communication with the standard input.
- SILex itself parses the macro names, the regular expressions, the interior of a string, . . . , with different sets of patterns.

We first begin with an overview on how SILex allows the programmer to create multiple lexical analysers. We continue with a description of the function `lex-tables`. We end the explanations with the functions used to creat analysers dynamically.

4.2.1 Creating analysers dynamically

It is quite easy to create new analysers at run-time. Suppose there is an input that you want to analyse. There are just two steps to make.

- Create an *input system* from the input. An input system provides the buffering, the line counting and similar low level services.
- Create one or more analysers from the input system and the analyser tables. The tables are generated by the function `lex-tables` from a specification file. A table contains all the necessary information to build up an analyser. Normally, you have to use more than one analyser per input when you expect the syntax to vary greatly in the input.

The following example shows a typical organization for a multi-analyser lexical analysis. Note that one table may have been used to produce many instances of analysers. Those analysers would simply be connected to different input systems[1].

```
        Input1            Input2        Input3
          |                 |             |
          |                 |             |
         IS1               IS2           IS3
          |                 |             |
  +-------+-------+         |          +--+---+
  |       |       |         |          |      |
An1.1   An1.2   An1.3     An2.1      An3.1  An3.2
```

There is no *a priori* limit on the number of input systems and analysers that you can create dynamically.

4.2.2 The `lex-tables` command

The function `lex-tables` produces a table describing an analyser from a specification file. A call to `lex-tables` looks like:

(`lex-tables` *`input-file table-name output-file`* `[`*`options`*` ...])`

input-file must be a string containing the name of the specification file. *output-file* is a string containing the name in which the result is printed. A definition is written in the output file. *table-name* must be a string and it is the name appearing in the definition. The options are defined in Section 4.3 [Options], page 11.

This is an example of a call to `lex-tables`:

```
(lex-tables "c.l" "c-table" "c.l.scm")
```

4.2.3 Building and using lexical analysers dynamically

In order to be able to create dynamically the analysers the program needs, the files containing the tables and the file '`multilex.scm`' must be loaded as part of the program. The name convention is the following: all definitions in '`multilex.scm`' introduce names beginning with '`lexer...`' and the definitions in the other files introduce names that are specified by the programmer. This way, it is easy to avoid name conflicts.

Input systems are created with the function `lexer-make-IS`. A call to this function looks like:

[1] It would make no sense to create two instances coming from the same table and being connected to the same input system. They would both have exactly the same behavior.

`(lexer-make-IS` *`input-type input`* `[`*`counters`*`])`

The values *input-type* and *input* are described in Section 4.4 [Input], page 13. The value of *counters* determines which counters the input system should maintain. This is discussed in Section 4.4 [Input], page 13. Input systems are associative lists that cannot be used directly.

Useful functions can be extracted from an input system. The following calls return functions that allows the program to interact with the input system:

```
(lexer-get-func-line input-system)
(lexer-get-func-column input-system)
(lexer-get-func-offset input-system)
(lexer-get-func-getc input-system)
(lexer-get-func-ungetc input-system)
```

Lexical analysers are created with the function `lexer-make-lexer`. The template of a call to this function is:

`(lexer-make-lexer` *`table input-system`*`)`

table is a table generated by SILex. *input-system* is the input system from which the analyser will take its input. The result of the call is the analysis function. The analysis function takes no argument and returns tokens.

This example summarizes all the step in the creation of an analyser:

```
(let* ((my-port       (open-input-file "my-file"))
       (my-IS         (lexer-make-IS 'port my-port))
       (my-get-line   (lexer-get-func-line IS))
       (my-get-column (lexer-get-func-column IS))
       (my-get-offset (lexer-get-func-offset IS))
       (my-getc       (lexer-get-func-getc IS))
       (my-ungetc     (lexer-get-func-ungetc IS))
       (my-analyser   (lexer-make-lexer my-table IS)))
  (let loop ((tok (my-analyser)))
    (cond ((eq? tok 'eof)
           ...
```

4.3 Options at generation time

We describe the options that can be passed to `lex` and `lex-tables`. They indicate which counters (line, column and offset) the actions need; which table encoding should be used; and whether the tables should be pretty-printed.

4.3.1 Line, column and offset counters

There are three different counting modes: no counter, line counter and all counters. The more counters the input system maintains, the more it is slowed down. The default is the line counting.

This option is specified when the program calls the functions `lex`, `lex-tables` and `lexer-make-IS`. The three modes are represented by the symbols '`none`', '`line`' and '`all`'. When one of the first two functions is called the mode must be preceded by the symbol '`counters`'. These examples illustrate the use of the option:

```
(lex "html.l" "html.l.scm" 'counters 'none)

(lex-tables "cobol.l" "cobol-table" "cobol.l.scm" 'counters 'line)

(lexer-make-IS 'port my-port 'all)
```

You should be careful when you build analysers dynamically. The mode specified at the input system creation must be consistent with the mode specified at the tables creation.

4.3.2 Encoding of the table of an analyser

SILex provides three different encodings of the tables: the default encoding, the portable encoding and the "compilation" to Scheme code.

With the default encoding, the finite automaton of the analyser is represented with data structures that contain the *numbers* of the characters (in the sense of `char->integer`). Since the numbers associated with the characters may depend on the Scheme implementation, an analyser generated with an implementation can be safely used only with the same implementation. An analyser encoded in the default style is not portable. But this representation is the most compact.

With the portable encoding, the data structures describing the automaton contain characters directly. If the automaton, as generated, contains a transition from state *s* to state *t* on character *c*, then somewhere in the table there is the Scheme character '`#\c`'. When the file containing the analyser is loaded in any implementation, the character is read as is, and not as the number '`(char->integer #\c)`' as evaluated by the original implementation. As long as the implementation using the analyser recognizes the characters mentionned in it, there is no problem.

So this encoding is portable. However, it is less compact. This is because something like '`(65 90)`' is more compact than something like '`(#\A #\B ... #\Y #\Z)`' to represent '`[A-Z]`'. The construction of an analyser from a portable table takes more time than the construction from a default table. But, once built, the performance of the analyser is the same in both cases.

It is important to note that in some character sets, the letters or the digits are not contiguous. So, in those cases, the regular expression '`[A-Z]`' does not necessarily accept only the uppercase letters.

The last encoding is the compilation to Scheme code. This produces a fast lexical analyser. Instead of containing data structures representing the behavior of the automaton, the table contains Scheme code that "hard-codes" the automaton. This encoding often generates big tables. Such an analyser is not portable.

The encoding of the tables can be specified as an option when `lex` and `lex-tables` are called. The symbols '`portable`' and '`code`' are used to specify that the table must be portable and that the table must be compiled, respectively. For example, these calls illustrate the use of the options:

```
(lex "c.l" "c.l.scm")                ; Default encoding

(lex "c.l" "c.l.scm" 'portable)      ; Portable encoding
```

```
(lex "c.l" "c.l.scm" 'code)          ; Compilation of the automaton
```

4.3.3 Pretty printing the tables

The pretty-print option (specified with the symbol '`pp`') tells SILex to pretty-print the contents of the table. Normally, the table is displayed as a compact mass of characters fitting in about 75 columns. The option is useful only for a developer of SILex. The Scheme code generated with the '`code`' option is always pretty-printed.

4.4 Input methods

An analyser can take its input from three different objects: an input port, a string or a function. The type of input and the input itself must be passed when an analyser is initialized and when an input system is created. The input type is specified using one of the three symbols: '`port`', '`string`' or '`procedure`'. For example:

```
(lexer-init 'port (current-input-port))

(lexer-make-IS 'string "Input string.")
```

When an input port is used by an analyser, the program should avoid reading characters directly from the port. This is because the analyser may have needed a look-ahead to do the analysis of the preceding token. The program would not find what it expects on the port. The analyser provides safe functions to get characters from the input. The analyser never closes itself the port it has received, this task is left to the program.

When the analyser is initialized with a string, it takes a copy of it. This way, eventual mutations of the string do not affect the analysis.

The use of a function as character source allows the analyser to parse any character stream, no matter how it is obtained. For example, the characters may come from the decompression or decryption of a huge file, the task being done lazily in order to save space. The function must take no argument and return a character each time it is called. When the end of file (or its logical equivalent) is reached, the function must return an object that is not a character (for example, the symbol '`eof`'). After the function has returned an end of file indicator, it is not called again.

Appendix A Interfacing with an LALR(1) parser

A nice LALR(1) parser generator for Scheme has been written by Dominique Boucher. The generator is accessible at the Scheme Repository at `ftp://ftp.cs.indiana.edu` in the file '`/pub/scheme-repository/code/lang/lalr-scm.tar.gz`'.

The parsers that are generated need two functions to operate: a lexical analysis function and an error function. The analysis function must take no argument and return a token each time it is called. This is exactly the behavior of the lexical analysis functions created by SILex.

The LALR(1) parsers expect that the tokens are pairs with a number in the CAR, the token number, and any value in the CDR, the token attribute. It is easy to respect this convention with a SILex lexical analyser since the actions can be any Scheme expressions. Furthermore, the file created by the LALR(1) parser generator contains definitions that give names to the number of the tokens. A lexical analyser can use those names in its actions in order to simplify the coordination between the two analysers.

Acknowledgements

I would like to thank my comrades of the laboratory for their support in this project. Especially Martin Larose and Marc Feeley for their numerous suggestions.

I hope SILex will be useful for many Scheme programmers.

If you find a bug, please let me know at `mailto:dube@iro.umontreal.ca`.

Index

Table of Contents