User's Guide to ML-Lex and ML-Yacc

containing

A lexical analyzer generator for Standard ML, Version 1.6.0

Andrew W. Appel[1] James S. Mattson David R. Tarditi[2]

ML-Yacc User's Manual, Version 2.4

David R. Tarditi Andrew W. Appel

Revised 2002-10-31, 2004-02-20, 2005-12-01
Minor revision 2006-02-13
Corrections and additional notes by Hans Leiss, 2009-03-22

[1]Department of Computer Science, Princeton University
[2]Microsoft Research

This Guide is based substantially on the documentation for ML-Lex and ML-Yacc which carries the following copyright notice:

Additional material:

Changes:

- 2005. Explanation of how to use ML-Lex with multi-byte character sets.
- 2009-03-20. Corrections and Hans Leiss's answers.

Contents

List of Figures

List of Tables

1 Introduction

> *There's a program they call ML-Yacc.*
> *It reads files that grunt coders must hack.*
> *And if Joe Grunt expects*
> *To integrate Lex,*
> *He must read this from front through to back.*

This User's Guide describes two programs and their interfaces: ML-Lex and ML-Yacc. Together they provide lexical analysis and parsing functions for general use: configuration scripts, database views, marked-up documents, messages, keyboard commands and computer programs.

The Guide is aimed at professionals who need to deliver working language processors as quickly as possible, but does not neglect academic needs.

The two programs, ML-Lex and ML-Yacc, may be used individually, together, or both integrated into a larger project such as a language processor. This Guide gives some examples of standalone use, but concentrates on the integration of ML-Lex and ML-Yacc into a larger project whose build is managed by the SML/NJ Compilation Manager.

The lexer and parser are often the front end of a compiler. SML is an excellent language for writing compilers, even if the other project languages are, for technical, political, commercial or mystical reasons, not SML. The lexing and parsing facilities offered with SML/NJ are therefore of practical interest.

The reader is assumed to have a good working knowledge of SML; see [Ull98, Pau96], and have some understanding of compilers [ASU86, App98].

If you are in a real panic, begin with the working example in chapters 11.2 and 11.3: use these as a starting point, and a demonstration to management that work is progressing. Adapt the example to your own language needs and add whatever processing the customers are calling for today. The rest of the Guide will, hopefully, answer some of your questions. The remaining questions might find answers in the SML/NJ mailing list `https://lists.sourceforge.net/lists/listinfo/smlnj-list`. The previous mailing list `sml-list@cs.cmu.edu` is now obsolete and abandoned.

1.1 Interfaces

Figure 1 shows the three key interfaces in a programming system which seeks to understand a stream of characters:

1. The characters are taken from a *character set* which should be clearly stated. As a possible starter for your project, chapter 2 describes a popular character set which ML-Lex and ML-Yacc can handle.

2. The tokens and their payload which represent the lexical items found in the input character stream. The set of possible tokens is defined in the second section of the `.yacc` or `.grm` file, see chapter 9.4.2.

3. The parse tree which represents a first understanding of the structure of the data in the source file. The set of possible constructions in the parse tree is often defined by ML datatypes in a separate file.

The language designer defines:

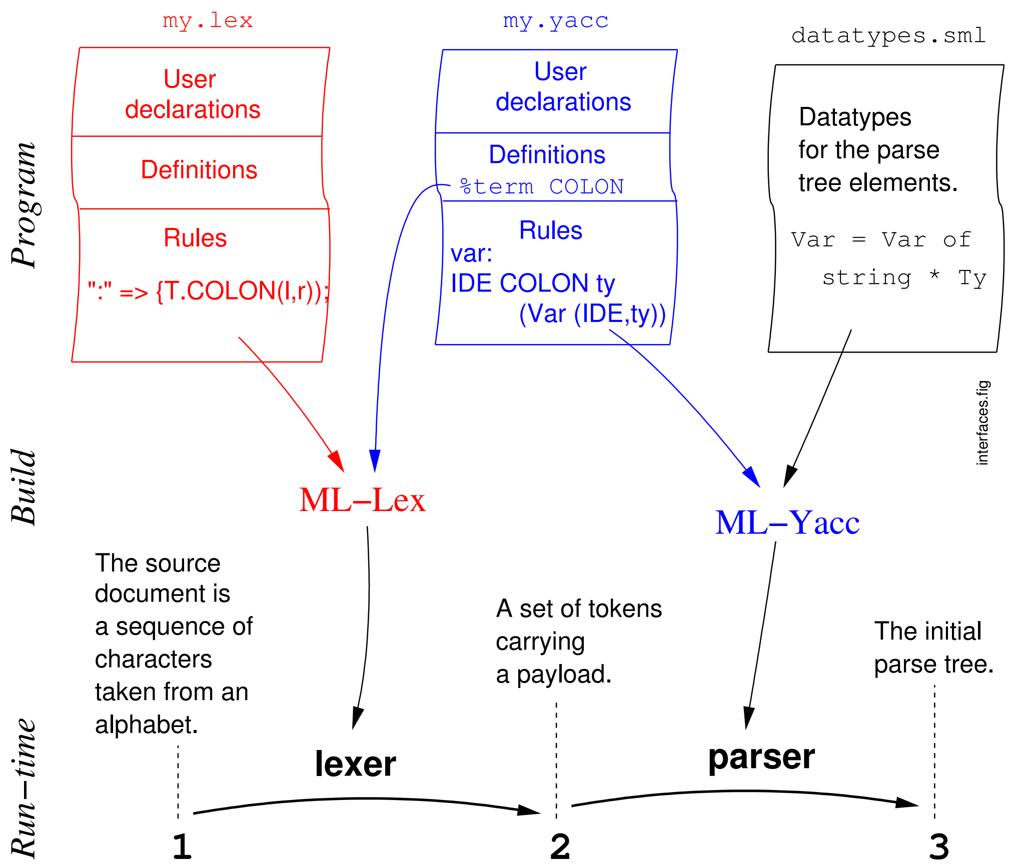

Figure 1: The key interfaces.

- The lexing function from interface 1 to interface 2 by specifying a set of ML-Lex rules in a file with extension `.lex`, chapter 7.3.

- The parsing function from interface 2 to interface 3 by specifying a set of ML-Yacc rules in a file with extension `.yacc` or `.grm`, chapter 9.5.

2 Alphabet

Discussion of sets of characters is often a confusing mixture of references to glyphs, character names, positions and values. This is rarely a problem for a basic character set such as “ASCII”, but to assist discussion of more complex sets, figure 2 introduces some of the terms used.

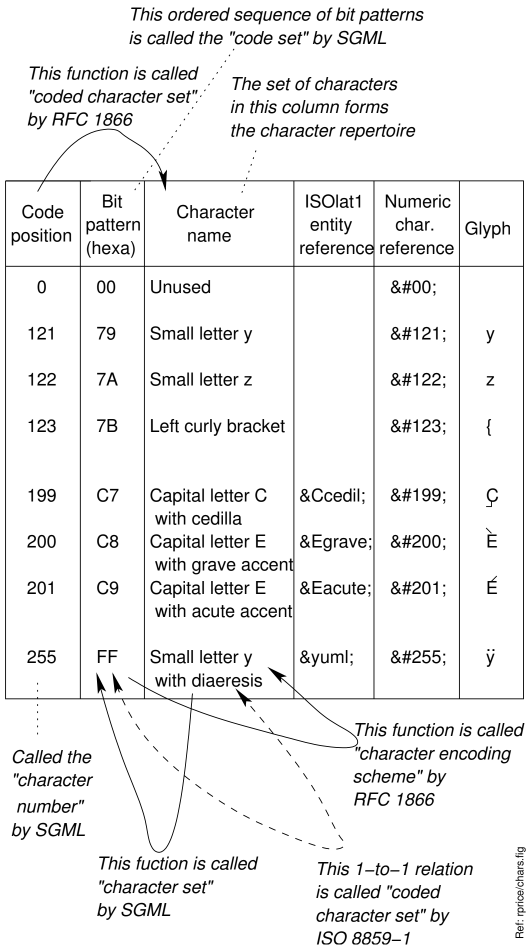

Figure 2: Terms used when describing character sets.

2.1 ISO Latin 1 and it's relatives

ML-Lex supports any 8-bit character set, and is conveniently used with ISO Latin 9[3] [ISO99], a variant of ISO Latin 1 [ISO87] which is the first 256 characters of Unicode[4] [TUC03] as defined by the Unicode Consortium. Since the characters are not "hard-wired" into the lexer, other 8-bit character sets can be used. For example character sets based on ISO 2022 or any of the other parts of ISO 8859 also known as "ISO Latin".

The character set may be reduced to the first 128 characters of Unicode, often known as "ASCII", if the option `%full`, chapter 7.2.1, is removed from the ML-Lex definitions section.

[3]ISO Latin 9, which is ISO Latin 1 with the currency symbol replaced by the Euro symbol and seven other changes, looks as if it will quickly replace ISO Latin 1 in popularity. It is intended for general purpose applications in typical office environments in at least the following languages of European origin: Albanian, Basque, Breton, Catalan, Danish, Dutch, English, Estonian, Faeroese, Finnish, French, Frisian, Galician, German, Greenlandic, Icelandic, Irish Gaelic (new orthography), Italian, Latin, Luxemburgish, Norwegian, Portuguese, Rhaeto-Romanic, Scottish Gaelic, Spanish, and Swedish. There are several official written languages outside Europe that are covered by Latin alphabet No. 9. Examples are Indonesian/Malay, Tagalog (Philippines), Swahili, Afrikaans.

[4]The online edition of the Unicode Standard, Version 4.1 is available at <http://www.unicode.org>.

ISO Latin 9 is listed in appendix A.

2.2 Unicode

ML-Lex may also be used with any set of characters taken from those defined by the Unicode Consortium [TUC03], and may be used with a wide variety of character encodings. The recommended technique is to

1. Convert the encoding of the source file from it's original encoding to big-endian UTF-32. This is always possible.

2. Modify the lexer specification to handle octets 4 at a time, that is one UTF-32 encoded character at a time.

See chapter 11.3 on page 45 for a worked example.

3 General description of ML-Lex

Computer programs often need to divide their input into words and distinguish between different kinds of words. Compilers, for example, need to distinguish between integers, reserved words, and identifiers. Applications programs often need to be able to recognise components of typed commands from users.

The problem of segmenting input into words and recognising classes of words is known as *lexical analysis*. Small cases of this problem, such as reading text strings separated by spaces, can be solved by using hand-written programs. Larger cases of this problem, such as tokenizing an input stream for a compiler, can also be solved using hand-written programs.

A hand-written program for a large lexical analysis problem, however, suffers from two major problems. First, the program requires a fair amount of programmer time to create. Second, the description of classes of words is not explicit in the program. It must be inferred from the program code. This makes it difficult to verify if the program recognises the correct words for each class. It also makes future maintenance of the program difficult.

Lex, a programming tool for the Unix system, is a successful solution to the general problem of lexical analysis. It uses regular expressions to describe classes of words. A program fragment is associated with each class of words. This information is given to Lex as a specification (a Lex program). Lex produces a program for a function that can be used to perform lexical analysis.

The function operates as follows. It finds the longest word starting from the current position in the input stream that is in one of the word classes. It executes the program fragment associated with the class, and sets the current position in the input stream to be the character after the word. The program fragment has the actual text of the word available to it, and may be any piece of code. For many applications it returns some kind of value.

Lex allows the programmer to make the language description explicit, and to concentrate on what to do with the recognised words, not how to recognise the words. It saves programmer time and increases program maintainability.

Unfortunately, Lex is targeted only at C. It also places artificial limits on the size of strings that can be recognised.

ML-Lex is a variant of Lex [LMB95] for the ML programming language. ML-Lex has a syntax similar to Lex, and produces an ML program instead of a C program. ML-Lex produces a program that runs very efficiently. Typically the program will be as fast or even faster than a hand-coded lexer implemented in Standard ML.

The program typically uses only a small amount of space. ML-Lex thus allows ML programmers the same benefits that Lex allows C programmers. It also does not place artificial limits on the size of recognised strings.

ML-Lex was designed for 7 and 8 bit character sets, but may be used with any Unicode based character set.

4 ML-Lex specifications

An ML-Lex specification has the general format:

ML-Lex user declarations
%%
ML-Lex definitions
%%
ML-Lex rules

Each section is separated from the others by a *%%* delimiter.

4.1 User declarations

You make ML declarations which will

1. Provide comments for the entire lexer.
2. Assist the glueing of the lexer to the parser.
3. Define values and functions available to all rule actions.

You must define at least two values in this section — the type `lexresult` and the function `eof`. The type `lexresult` defines the type of the basic payload values returned by the rule actions. The function `eof` is called by the lexer when the end of the input stream is reached. It will typically return a value signalling “eof” or raise an exception. It is called with the same argument as `lex`, see chapter 7.2.7, and must return a value of type `lexresult`. See 7.1.

4.2 Definitions

In the ML-Lex definitions section, you can define named regular expressions, a set of start states, and specify which of the various bells and whistles of ML-Lex are desired. See 7.2.

The start states allow you to control when certain rules are matched. Rules may be defined to match only when the lexer is in specific start states. You may change the lexer’s start state in a rule action. This allows you to specify special handling of lexical objects.

This feature is typically used to handle quoted strings with escapes to denote special characters. The rules to recognise the inside contents of a string are defined for only one start state. This start state is entered when the beginning of a string is recognised, and exited when the end of the string is recognised.

4.3 Rules

The rules are used to define the lexical analysis function. Each rule has two parts—a *regular expression* and an *action*. The regular expression defines the word class that a rule matches. The action is a program fragment to be executed when a rule matches the input. The actions are used to compute values, and must all return values of the same type. See chapter 7.3.

5 ML-Lex Output

The output from the lexer is a stream of *tokens* which are to be fed to a parser such as might be defined by ML-Yacc.

A token in ML-Lex is a function which takes as argument two or more values called the *payload*. The tokens are defined by the combined effect of

1. The `%term` commands used in the ML-Yacc declaration section of your ML-Yacc specification. These may add extra values to the token function's argument and thus extend the payload.

2. The `lexresult` type declaration in the user declarations of your ML-Lex specification. See line 9. This defines the type of the result.[5]

5.1 Tokens with basic payload

If a token has been defined by the `%term` command in the `.yacc` file with no type, then its payload is usually two integers — its the `%pos` declaration which says so, see chapter 9.4.3 on page 22. For example, looking at the SML/NJ compiler, we see that the semicolon is defined by the ML-Yacc `%term` command in file `ml.grm` as `SEMICOLON`. There is no type specification. The payload is two integers specifying the character positions in the source file of the start and end of the semicolon:

```
1  <INITIAL>";" => (Tokens.SEMICOLON(yypos,yypos+1));
```

Line 1 taken from the ML-Lex definition sections of file `ml.lex` shows that when a semicolon is detected, token `SEMICOLON` is sent to the parser with a basic payload giving the start and end of the semicolon. See chapter 7.3.5 for details of `yypos`.

5.2 Tokens with supplemental payload

If a token has been defined in ML-Yacc with a type, then its payload will be a value of that type, followed by two integers — again, its the `%pos` declaration which calls for those two integers, see chapter 9.4.3 on page 22.. For example, looking at the SML/NJ compiler, we see that a real number is defined by the ML-Yacc `%term` command in file `ml.grm` as `REAL of string`. The payload is therefore a string followed by two integers specifying the character position in the source file of the start and end of the real number:

```
<INITIAL>{real} => (Tokens.REAL(yytext,
                                yypos,
                                yypos+size yytext));
```

[5]It is unlikely that you will want to modify this.

Line 2 taken from the ML-Lex definitions section of file `ml.lex` shows that when a real number is detected, token `REAL` is sent to the parser with an argument giving the string representation of the real number, and the start and end positions of the number, lines 3 and 4. See chapter 7.3.1 for details of `yytext`.

6 ML-Lex regular expressions

Regular expressions are a simple language for denoting classes of strings. A regular expression is defined inductively over an alphabet with a set of basic operations.

The syntax and semantics of regular expressions will be described in order of decreasing precedence (from the most tightly binding operators to the most weakly binding):

- An individual character stands for itself, except for the reserved characters `? * + | ( ) ^ $ / ; . = < > [ { " \`

  A backslash followed by one of the reserved characters stands for that character.

- A set of characters enclosed in square brackets "`[ ]`" stands for any one of those characters. Inside the brackets, only the three symbols `\ - ^` are reserved. An initial up-arrow `^` stands for the complement of the characters listed, e.g. `[^abc]` stands any character except a, b, or c.

  The hyphen `-` denotes a range of characters[6], e.g. `[a-z]` stands for any lower-case non-accented alphabetic character, and `[0-9a-fA-F]` stands for any hexadecimal digit.

  If the source document is encoded in ISO Latin 9, then the specification `[A-Za-zŠšŽžŒ-ŸÀ-ÖØ-öø-ÿ]` stands for any alphabetic character including upper case and accented characters. Yes, that "Œ" is a single character. Yes, people do use this stuff.

  To include `^` literally in a bracketed set, put it anywhere but first; to include `-` literally in a set, put it first or last.

. The dot `.` character stands[7] for any character except newline, i.e. the same as `[^\n]`

- The following special escape sequences[8] are available, inside or outside of square-brackets:

[6] Is this correct? Its the explanation that is often given for the "-" notation, so perhaps it will do for a mid-term answer, but strictly speaking the hyphen denotes a range of *character positions* from the position of the left character through to the position of the right character. If we are lucky, as we are with the lower case non-accented letters `a` through `z`, this also corresponds to the desired range of characters, but its certainly not true for the accented characters.

[7] Only for one octet per character encodings; not for UTF-32.

[8] With the exception of the `\ddd` notation, these escape sequences are less exciting than they may appear. It is implicit that they can only be used if the source file is encoded with single octet characters in such a way to agree with "ASCII" in the first 127 characters. `\n` depends on the underlying operating system; see lines 324 on page 40 and 545 on page 50 for alternative definitions. Neither `\b`, `\n` nor `\t` are correct for UTF-32 encodings.

`\b`	backspace
`\n`	newline
`\t`	horizontal tab
`\h`	stands for all characters with codes > 127, when 7-bit characters are used.
`\ddd`	where `ddd` is a 3 digit decimal escape.

`"` A sequence of characters will stand for itself (reserved characters will be taken literally) if it is enclosed in double quotes `" "`. For example `"Dog"` will match `Dog`, but not `Dg`, `oog` or `gDo`.

{} A named regular expression, defined in chapter 7.2.9 on page 13, may be referred to by enclosing its name in braces { }.

() Any regular expression may be enclosed in parentheses `( )` for syntactic (but, as usual, not semantic) effect.

`*` The postfix operator `*` stands for Kleene closure: zero or more repetitions of the preceding expression.

`+` The postfix operator `+` stands for one or more repetitions of the preceding expression.

`?` The postfix operator `?` stands for zero or one occurrence of the preceding expression.

- A postfix repetition range $\{n_1, n_2\}$ where n_1 and n_2 are small integers stands for any number of repetitions between n_1 and n_2 of the preceding expression. The notation $\{n_1\}$ stands for exactly n_1 repetitions.

- Concatenation of expressions denotes concatenation of strings. The expression e_1e_2 stands for any string that results from the concatenation of one string that matches e_1 with another string that matches e_2.

`|` The infix operator `|` stands for alternation. The expression e_1 `|` e_2 stands for anything that either e_1 or e_2 stands for.

`/` The infix operator `/` denotes lookahead. Lookahead is not implemented and cannot be used, because there is a bug in the algorithm for generating lexers with lookahead. If it could be used, the expression e_1`/`e_2 would match any string that e_1 stands for, but only when that string is followed by a string that matches e_2.

Warning The use of the lookahead operator `/` will also slow down the entire lexer.

- When the up-arrow ˆ occurs at the beginning of an expression, that expression will only match strings that occur at the beginning of a line (right after a newline character).

$ The dollar sign of C Lex $ is not implemented, since it is an abbreviation for lookahead involving the newline character that is, it is an abbreviation for `/\n`.

Here are some examples of regular expressions, and descriptions of the set of strings they denote:

`0 | 1 | 2 | 3`	A single digit between 0 and 3
`[0123]`	A single digit between 0 and 3
`0123`	The string “0123”
`0*`	All strings of 0 or more 0’s
`00*`	All strings of 1 or more 0’s
`0+`	All strings of 1 or more 0’s
`[0-9]{3}`	Any three-digit decimal number.
`\\[ntb]`	A newline, tab, or backspace.
`(00)*`	Any string with an even number of 0’s.

7 ML-Lex section summary

7.1 ML-Lex user declarations

Anything up to the first %% is in the user declarations section.

This section contains ML declarations which are to be placed in a structure called `UserDeclarations`. The section is written using ML syntax and may include ML comments. No ML symbolic identifier containing %% can be used in this section.

If the lexer is to be used with the ML-Yacc parser, then additional glue declarations are needed:

```
5  structure T = Tokens
6  type pos = int               (* Position in file *)
7  type svalue = T.svalue
8  type ('a,'b) token = ('a,'b) T.token
9  type lexresult = (svalue,pos) token
10 type lexarg = string
11 type arg = lexarg
12 val linep = ref 1;           (* Line pointer *)
```

Lines 5 through 9 provide the basic glue. On line 9, `lexresult` returns the type of the result returned by the rule actions.

If you are passing a parameter to the lexer, then you also need the additional glue in lines 10 through 11.

The lexer offers the possibility of counting lines using value `yylineno` described in chapter 7.3.6. If you prefer to do this yourself with variable `linep`, you will need the declaration on line 12.

7.1.1 Payload

As described in chapter 5, the lexer provides a payload for each token which always includes two integers used to fix the position of the token in the source document. There are several styles for the use of these arguments. For example, the SML/NJ compiler uses them to represent the character positions in the source of the start and end of the token. A simpler arrangement is suitable if there is less syntactic activity on each line. The two arguments each hold the line number in the file. A compromise could be to use the first argument for the line number and the second for the position in the line. Its up to you, but once you have decided, you will need functions to print error messages for lexer errors and unwelcome characters:

```
13 val error : string * int * int -> unit = fn
14     (e,l1,l2) => TextIO.output(TextIO.stdOut,"lex:line "
15                 ^Int.toString l1^" l2="^Int.toString l2
16                 ^": "^e^"\n")
17 val badCh : int * char -> unit = fn
18     (l1,ch) => TextIO.output(TextIO.stdOut,"lex:line "
19               ^Int.toString l1^": Invalid character "
20               ^Int.toString(ord ch)^"="^str(ch)^"\n")
```

On line 13 the parameters are a human-readable text, a line number and a character position. On line 17 the parameters are a line number and an ML character. It looks as if some more work is needed on these functions to produce a polished output `:-)`.

The working example in chapter 11.2 provides an alternative definition for function `badCh`, see line 277 on page 38.

7.1.2 End of file (EOF)

What happens at the end of the source file? If all goes well the source document or program should be complete, but sometimes this is not the case. A typical error is to forget to close an ongoing comment. If you allow ML style nested comments `(* ... (* ... *) ... *)` then you will need some management of nested comments and possible end-of-file errors in the lexer.

```
21 val mlCommentStack : (string*int) list ref = ref [];
22 val eof = fn fileName =>
23     (if (!mlCommentStack)=[] then ()
24      else let val (file,line) = hd (!mlCommentStack)
25           in TextIO.output(TextIO.stdOut,
26              "       I am surprized to find the
27             ^" end of file \""fileName^"\"\n"
28             ^"       in a block comment which began"
29             ^" at "^file^"["^Int.toString line^"].\n")
30           end;
31      T.EOF(!linep,!linep));
```

Line 21 declares a stack for ML style comments. Each entry holds the file name[9] and line number at which the comment began. The function `eof` at line 22 provides some end of file management. It assumes that the ML-Lex command `%arg`, chapter 7.2.7, has been specified and the name of the source file `fileName` has been passed to the lexer, see line 417 on page 43. If this is not the case, then `fileName` is replaced by `()`. For this treatment of nested commands to work well, additional measures are needed for the ends of lines in the rules section 7.3.

7.1.3 Keywords

Your source language will probably include *keywords*, and now is the time to specify them with the functions to manage them. Here is an example which you could adapt to your needs:

[9] It might seem surprising in lines 21 and 24 to keep the file name in `mlCommentStack`. After all, the SML/NJ compiler doesn't do it. However if you work in an SGML/XML context, using an OASIS catalog, then you may find yourself taking characters from unexpected documents, some of which might have been down loaded by the "entity manager". In this case, the programmer needs every assistance in locating the sources of bugs.

```
32 structure KeyWord :
33 sig
34     val find:string->(int*int->(svalue,int) token) option
35 end =
36 struct
37  val TableSize =  422 (* 211 *)
38  val HashFactor = 5
39  val hash = fn
40      s => List.foldr (fn (c,v) => (v*HashFactor+(ord c))
41                       mod TableSize) 0 (explode s)
42  val HashTable = Array.array(TableSize,nil) :
43        (string * (int * int -> (svalue,int) token))
44        list Array.array
45  val add = fn
46      (s,v) => let val i = hash s
47               in Array.update(HashTable,i,(s,v)
48                  :: (Array.sub(HashTable, i)))
49               end
50  val find = fn
51      s => let val i = hash s
52               fun f ((key,v)::r) = if s=key then SOME v
53                                                  else f r
54                 | f nil = NONE
55           in  f (Array.sub(HashTable, i))
56           end
57  val _ = (List.app add [
58       ("ripoff",    T.RIPOFF),
59       ("shakedown", T.SHAKEDOWN),
60       ("kickback",  T.KICKBACK),
61       ("respect",   T.RESPECT),
62       ("cityhall",  T.CITYHALL)
63      ])
64 end
```

Place your case sensitive keywords in the list beginning on line 58. The list must agree with the keyword declaration in your ML-Yacc file.

7.2 ML-Lex definitions

The ML-Lex definitions section provides the following commands. They are all terminated with a semicolon `;`.

7.2.1 `%full`

Create lexer for the full 8-bit character set, with character codes in the range 0–255 permitted as input. If this command is omitted, the lexer accepts a 7-bit character set with the escape sequence `\h` representing the character codes 128 through 255.

7.2.2 `%header`

Use the specified code to create a functor header for the lexer structure. For example, if you are using ML-Yacc and you have specified `%name` *My* in the ML-Yacc declarations:

```
%header (functor MyLexFun(structure Tokens: My_TOKENS));
```

This has the effect of turning what would have been a structure into a functor. The functor is needed for the glue code which integrates the lexer into a project.

The SML/NJ compiler uses this technique with `ML` in place of *My*. Our working example also uses the technique with `Pi` in place of *My*. See lines 317 on page 40 and 391 on page 42.

If you prefer to create the lexer as an SML/NJ structure, then omit this command and use the command `%structure`.

7.2.3 `%structure`

If you prefer to create your lexer as an SML/NJ structure rather than a functor, when for example you are not using ML-Yacc, then use the command `%structure` *identifier* to name the structure in the output program `my.lex.sml` as *identifier* instead of the default `Mlex`.

7.2.4 `%reject`

Create a `REJECT` function. See 7.3.4.

7.2.5 `%count`

Count newlines using `yylineno`. See 7.3.6.

7.2.6 `%posarg`

Pass an initial-position argument to function `makeLexer`. See 10.4.

7.2.7 `%arg`

An extra (curried) formal parameter argument is to be passed to the `lex` functions, and to the `eof` function in place of `()`. See 7.3.2. For example:

```
66  %arg (fileName:string);
```

specifies that there is an argument for the lexer, its name is `fileName` and it has type `string`. The argument value is passed in the call to the parser. See line 415 on page 43.

7.2.8 `%s` *identifier list*

It is often convenient to place the rules in groups with a separate set of rules for each group. For example, rules for comments are often put into such a group. Each group corresponds to a *state* and the additional states that you create have to be declared. The base state of the lexer is `INITIAL`. You do not need to declare the base state.

```
67  %s A S F Q AQ L LL LLC LLCQ;
```

Line 67 shows the *identifier list* declaration of the SML/NJ compiler.

- An *identifier list* consists of one or more *identifiers*.
- Each *identifier* consists of one or more letters, digits, underscores, or primes, and must begin with a letter.

7.2.9 Named expressions

ML-Lex provides a macro facility for creating *named expressions*. The replacement text is a *regular expression* as defined in chapter 6.

The syntax is *identifier* = *regular expression*

```
68 idchars = [A-Za-z'_0-9];
69 id      = [A-Za-z]{idchars}*;
70 ws      = ("\012"|[\t\ ])*;
71 nrws    = ("\012"|[\t\ ])+;
72 eol     = ("\013\010"|"\010"|"\013");
73 some_sym= [!%&$+/:<=>?@~|#*]|\-|\^;
74 sym     = {some_sym}|"\\";
75 quote   = "`";
76 full_sym= {sym}|{quote};
77 num     = [0-9]+;
78 frac    = "."{num};
79 exp     = [eE](~?){num};
80 real    = (~?)(({num}{frac}?{exp})|({num}{frac}{exp}?));
81 hexnum  = [0-9a-fA-F]+;
```

Lines 68 through 81 show the named expressions used for the 7-bit lexer in the SML/NJ compiler. Note on line 72 the definition of three possible end-of-line character sequences to match the end-of-line markers used in a range of operating systems.

7.3 ML-Lex Rules

Each rule has the format:

<*start state list*> *regular expression* `=>` `(` *code* `);`

- All parentheses in *code* must be balanced, including those used in strings and comments.
- The *start state list* is optional. It consists of a list of identifiers separated by commas, and is delimited by angle brackets `< >`. Each identifier must be a start state defined by the `%s` command, 7.2.8.
- The regular expression is only recognised when the lexer is in one of the start states in the *start state list*. If no start state list is given, the expression is recognised in all start states.
- The lexer begins in a pre-defined start state called `INITIAL`.
- The lexer resolves conflicts among rules by choosing the rule with the longest match, and in the case two rules match the same string, choosing the rule listed first in the specification.
- The rules should match all possible input. If some input occurs that does not match any rule, the lexer created by ML-Lex will raise an exception `LexError`. Note that this differs from C Lex, which prints any unmatched input on the standard output.

The following values are available inside rules.

7.3.1 `yytext`

ML-Lex places the value of the string matched by a regular expression in `yytext`, a string variable.

7.3.2 `lex()` and `continue()`

If `%arg`, chapter 7.2.7, is not used, you may recursively call the lexing function with `lex()`.

```
82 [\ \t]+         => ( lex() );
```

For example, line 82 ignores spaces and tabs silently;

However, if `%arg` is used, the lexing function may be re-invoked with the same argument by using `continue()`.

```
83 <COMMENT>.      => (continue());
```

For example, line 83 silently ignores all characters except a newline when the parser is in the user-defined state `COMMENT`.

7.3.3 `YYBEGIN` *state*

To switch start states, you may call `YYBEGIN` with the name of a start state.

```
84 <WORK>"%"       => (YYBEGIN COMMENT; continue());
```

For example, line 84 switches the lexer from state `WORK` to `COMMENT`. This might happen if the percent character "%" were used as a line comment symbol as happens in LaTeX, Prolog and other fine languages.

```
85 <COMMENT>{eol} => (linep:=(!linep)+1;
86                    YYBEGIN WORK; continue ());
```

In line 85, whenever the lexer detects an end of line when in the state `COMMENT`, it bumps the line counter and switches back to the state `WORK`. Note that `eol` was defined on line 72 in the ML-Lex definitions section.

7.3.4 `REJECT`

The function `REJECT` is defined only if the command `%reject` has been specified, chapter 7.2.4. `REJECT()` causes the current rule to be "rejected". The lexer behaves as if the current rule had not matched; another rule that matches this string, or that matches the longest possible prefix of this string, is used instead.

7.3.4.1 Warning This function should be used only if necessary. Adding `REJECT` to a lexer will slow it down by 20%.

7.3.5 `yypos`

The value `yypos` contains the position of the first character of `yytext`, relative to the beginning of the file.

If you have decided to use the basic payload arguments to your tokens to position the start and end of the token in the source file, then the position of the end of the `yytext` is given by `yypos+size yytext`. See line 2 for an example taken from the SML/NJ compiler.

7.3.5.1 The good news The character-position, `yypos`, is not costly to maintain.

7.3.5.2 The bad news The position of the first character in the file is wrongly reported as 2, unless the `%posarg` feature is used, chapter 7.2.6. To preserve compatibility, this bug has not been fixed. See chapter 14 on page 54 for a fix.

7.3.6 `yylineno`

The value `yylineno` is defined only if command `%count` has been specified, chapter 7.2.5. `yylineno` provides the current line number.

7.3.6.1 Warning This function should be used only if it is really needed. Adding the `yylineno` facility to a lexer will slow it down by 20%. It is much more efficient to recognise `\n` and have an action that increments a line-number variable. For example, see chapter 11.2.3 on page 38 in our working example.

8 Introduction to ML-Yacc

8.1 General

ML-Yacc is a parser generator for Standard ML modelled after the Yacc parser generator [LMB95]. It generates parsers for LALR languages, like Yacc, and has a similar syntax. The generated parsers use a different algorithm for recovering from syntax errors than parsers generated by Yacc. The algorithm is a partial implementation of an algorithm described in [BF87]. A parser tries to recover from a syntax error by making a single token insertion, deletion, or substitution near the point in the input stream at which the error was detected. The parsers delay the evaluation of semantic actions until parses are completed successfully. This makes it possible for parsers to recover from syntax errors that occur before the point of error detection, but it does prevent the parsers from affecting lexers in any significant way. The parsers can insert tokens with values, known as the *payload*, and substitute tokens with values for other tokens. All symbols carry a basic payload which is the left and right position values[10] which are available to semantic actions and are used in syntactic error messages.

St. Anford professor Jeff
Writes books and books about everything known to CS.
But he's at his best,
When writing with Aho and Seth.

ML-Yacc uses context-free grammars to specify the syntax of languages to be parsed. See [ASU86] for definitions and information on context-free grammars and LR parsing. We briefly review some terminology here. A context-free grammar is defined by a set of terminals T, a set of non-terminals NT, a set of productions P, and a start non-terminal S. Terminals are interchangeably referred to as tokens. The terminal and non-terminal sets are assumed to be disjoint. The set of symbols is the union of the non-terminal and terminal sets. We use lower case Greek letters to denote a string of symbols. We use upper case Roman letters near the beginning of the alphabet to denote non-terminals. Each production gives a derivation of a string of symbols from a non-terminal, which we will write as $A \rightarrow \alpha$. We define a relation between strings of symbols α and β, written $\alpha \vdash \beta$ and read as α derives β, if and only

[10]This is the way in which the two values are used in the SML/NJ compiler, but in our working example, chapter 11.2, the two values are both used to give the left position.

if $\alpha = \delta A \gamma$, $\beta = \delta \phi \gamma$ and there exists some production $A \to \phi$. We write the transitive closure of this relation as $\vdash_*$. We say that a string of terminals α is a valid sentence of the language, *i.e.* it is derivable, if the start symbol $S \vdash_* \alpha$. The sequence of derivations is often visualised as a parse tree.

ML-Yacc uses an attribute grammar scheme with synthesised attributes. Each symbol in the grammar may have a value (i.e. attribute) associated with it. Each production has a semantic action associated with it. A production with a semantic action is called a rule. Parsers perform bottom-up, left-to-right evaluations of parse trees using semantic actions to compute values as they do so. Given a production $P = A \to \alpha$, the corresponding semantic action is used to compute a value for A from the values of the symbols in α. If A has no value, the semantic action is still evaluated but the value is ignored. Each parse returns the value associated with the start symbol S of the grammar. A parse returns a nullary value if the start symbol does not carry a value.

The synthesised attribute scheme can be adapted easily to inherited attributes. An inherited attribute is a value which propagates from a non-terminal to the symbols produced by the non-terminal according to some rule. Since functions are values in ML, the semantic actions for the derived symbols can return functions which takes the inherited value as an argument.

8.2 Modules

ML-Yacc uses the ML modules facility to specify the interface between a parser that it generates and a lexical analyser that must be supplied by you[11]. It also uses the ML modules facility to factor out a set of modules that are common to every generated parser. These common modules include a parsing structure, which contains an error-correcting LR parser[12], an LR table structure, and a structure which defines the representation of terminals. ML-Yacc produces a functor for a particular parser parameterised by the LR table structure and the representation of terminals. This functor contains values specific to the parser, such as the LR table for the parser[13], the semantic actions for the parser, and a structure containing the terminals for the parser. ML-Yacc produces a signature for the structure produced by applying this functor and another signature for the structure containing the terminals for the parser. You must supply a functor for the lexing module parameterised this structure.

Figure 3 is a dependency diagram of the modules that summarises this information. A module at the head of an arrow is dependent on the module at the tail.

The glue code in our working example, chapter 11.2.5 on page 42, assembles the modules described in this chapter, and satisfies the dependencies of figure 3.

8.3 Error Recovery

The error recovery algorithm is able to accurately recover from many single token syntax errors. It tries to make a single token correction at the token in the input stream at which the syntax error was detected and any of the 15 tokens[14] before that token. The

[11] Using ML-Lex :-) .

[12] A plain LR parser is also available.

[13] The LR table is a value. The LR table structure defines an abstract LR table type.

[14] An arbitrary number chosen because numbers above this do not seem to improve error correction much.

parsing structure	$\longrightarrow$	values for a particular parser
values for a particular parser	$\longrightarrow$	lexical analyser
(parsing structure, values for a particular parser, lexical analyser)	$\longrightarrow$	particular parser

Figure 3: Module Dependencies

algorithm checks corrections before the point of error detection because a syntax error is often not detected until several tokens beyond the token which caused the error.[15]

The algorithm works by trying corrections at each of the 16 tokens up to and including the token at which the error was detected. At each token in the input stream, it will try deleting the token, substituting other tokens for the token, or inserting some other token before the token.

The algorithm uses a parse check to evaluate corrections. A parse check is a check of how far a correction allows a parser to parse without encountering a syntax error. You pass an upper bound on how many tokens beyond the error point a parser may read while doing a parse check as an argument to the parser. This allows you to control the amount of lookahead that a parser reads for different kinds of systems. For an interactive system, you should set the lookahead to zero. Otherwise, a parser may hang waiting for input in the case of a syntax error. If the lookahead is zero, no syntax errors will be corrected. For a batch system, you should set the lookahead to 15.

The algorithm selects the set of corrections which allows the parse to proceed the farthest and parse through at least the error token. It then removes those corrections involving keywords which do not meet a longer minimum parse check. If there is more than one correction possible after this, it uses a simple heuristic priority scheme to order the corrections, and then arbitrarily chooses one of the corrections with the highest priority. You have some control over the priority scheme by being able to name a set of preferred insertions and a set of preferred substitutions. The priorities for corrections, ordered from highest to lowest priority, are preferred insertions, preferred substitutions, insertions, deletions, and substitutions.

The error recovery algorithm is guaranteed to terminate since it always selects fixes which parse through the error token.

The error-correcting LR parser implements the algorithm by keeping a queue of its state stacks before shifting tokens and using a lazy stream for the lexer. This makes it possible to restart the parse from before an error point and try various corrections. The error-correcting LR parser does not defer semantic actions. Instead, ML-Yacc creates semantic actions which are free of side-effects and always terminate. ML-Yacc uses higher-order functions to defer the evaluation of all user semantic actions until the parse is successfully completed without constructing an explicit parse tree. You may declare whether your semantic actions are free of side-effects and always terminate, in which case ML-Yacc does not need to defer the evaluation of your semantic actions.

[15] An LR parser detects a syntax error as soon as possible, but this does not necessarily mean that the token at which the error was detected caused the error.

8.4 Precedence

ML-Yacc uses the same precedence scheme as Yacc for resolving shift/reduce conflicts. Each terminal may be assigned a precedence and associativity. Each rule is then assigned the precedence of its rightmost terminal. If a shift/reduce conflict occurs, the conflict is resolved silently if the terminal and the rule in the conflict have precedences. If the terminal has the higher precedence, the shift is chosen. If the rule has the higher precedence, the reduction is chosen. If both the terminal and the rule have the same precedence, then the associativity of the terminal is used to resolve the conflict. If the terminal is left associative, the reduction is chosen. If the terminal is right associative, the shift is chosen. Terminals may be declared to be non associative, also, in which case an error message is produced if the associativity is needed to resolve the parsing conflict.

If a terminal or a rule in a shift/reduce conflict does not have a precedence, then an error message is produced and the shift is chosen.

In reduce/reduce conflicts, an error message is always produced and the first rule listed in the specification is chosen for reduction.

ML-Yacc does not allow direct specification of non-terminal precedence and non-terminal associativity, however you can get the effect by introducing dummy terminals with the required precedence and associativity. See the `%prec` declaration, chapter 9.5.2 on page 29.

For further discussion of precedence, see [LMB95, p.196].

8.5 Notation

Text surrounded by brackets denotes meta-notation. If you see something like {*parser name*}, you should substitute the actual name of your parser for the meta-notation. Text in a bold-face typewriter font, **`like this`**, denotes text in a specification or ML code.

9 ML-Yacc specifications

An ML-Yacc specification consists of three parts, each of which is separated from the others by a %% delimiter. The general format is:

ML-Yacc user declarations
%%
ML-Yacc declarations
%%
ML-Yacc rules

Comments have the same lexical definition as they do in Standard ML and can be placed in any ML-Yacc section.

Before looking at the three sections, we first review the structure of the `.yacc` file.

9.1 ML-Yacc symbols

After the first %%, the following words and symbols are reserved:

```
of for = { } , * -> :  | ( )
```

The following classes of ML symbols are used:

identifiers: non-symbolic ML identifiers, which consist of an alphabetic character followed by one or more alphabetic characters, numeric characters, primes “'”, or underscores “_”.

type variables: non-symbolic ML identifier starting with a prime “'”

integers: one or more decimal digits.

qualified identifiers: an identifier followed by a period.

The following classes of non-ML symbols are used:

% identifiers: a percent sign followed by one or more lowercase alphabet letters. The valid % identifiers are:

`%arg` `%eop` `%header` `%keyword` `%left` `%name` `%nodefault` `%nonassoc` `%nonterm` `%noshift` `%pos` `%prec` `%prefer` `%pure` `%right` `%start` `%subst` `%term` `%value` `%verbose`

code: This class is meant to hold ML code. The ML code is not parsed for syntax errors. It consists of a left parenthesis followed by all characters up to a balancing right parenthesis. Parentheses in ML comments and ML strings are excluded from the count of balancing parentheses.

9.2 ML-Yacc grammar

This is the grammar for ML-Yacc specifications:

<table>
<tr><td>spec</td><td>::=</td><td>user-declarations %% cmd-list %% rule-list</td></tr>
<tr><td>ML-type</td><td>::=</td><td>non-polymorphic ML types</td></tr>
<tr><td></td><td></td><td>(see the Standard ML manual)</td></tr>
<tr><td>symbol</td><td>::=</td><td>identifier</td></tr>
<tr><td>symbol-list</td><td>::=</td><td>symbol-list symbol</td></tr>
<tr><td></td><td>|</td><td>ϵ</td></tr>
<tr><td>symbol-type-list</td><td>::=</td><td>symbol-type-list <code>|</code> symbol <code>of</code> ML-type</td></tr>
<tr><td></td><td>|</td><td>symbol-type-list <code>|</code> symbol</td></tr>
<tr><td></td><td>|</td><td>symbol <code>of</code> ML-type</td></tr>
<tr><td></td><td>|</td><td>symbol</td></tr>
<tr><td>subst-list</td><td>::=</td><td>subst-list <code>|</code> symbol <code>for</code> symbol</td></tr>
<tr><td></td><td>|</td><td>ϵ</td></tr>
<tr><td>cmd</td><td>::=</td><td><code>%arg</code> (Any-ML-pattern) <code>:</code> ML-type</td></tr>
<tr><td></td><td>|</td><td><code>%eop</code> symbol-list</td></tr>
<tr><td></td><td>|</td><td><code>%header</code> code</td></tr>
<tr><td></td><td>|</td><td><code>%keyword</code> symbol-list</td></tr>
<tr><td></td><td>|</td><td><code>%left</code> symbol-list</td></tr>
<tr><td></td><td>|</td><td><code>%name</code> identifier</td></tr>
</table>

	|	`%nodefault`
	|	`%nonassoc` symbol-list
	|	`%nonterm` symbol-type-list
	|	`%noshift` symbol-list
	|	`%pos` ML-type
	|	`%prefer` symbol-list
	|	`%pure`
	|	`%right` symbol-list
	|	`%start` symbol
	|	`%subst` subst-list
	|	`%term` symbol-type-list
	|	`%value` symbol code
	|	`%verbose`
cmd-list	::=	cmd-list cmd
	|	cmd
rule-prec	::=	`%prec` symbol
	|	ϵ
clause-list	::=	symbol-list rule-prec code
	|	clause-list `|` symbol-list rule-prec code
rule	::=	symbol `:` clause-list
rule-list	::=	rule-list rule
	|	rule

9.3 ML-Yacc user declarations

You can define values available in the semantic actions of the rules in the user declarations section. It is recommended that you keep the size of this section as small as possible and place large blocks of code in other modules.

All characters up to the first occurrence of a delimiting %% outside of a comment are placed in the user declarations section, structure `Header`.

If you have any significant processing to do, it would probably be better placed in some other structure than this section.

9.4 ML-Yacc declarations

The ML-Yacc declarations section is used to make a set of required declarations and a set of optional declarations. You must declare the non-terminals and terminals and the types of the values associated with them there. You must also name the parser and declare the type of position values. You should specify the set of terminals which can follow the start symbol and the set of non-shiftable terminals. You may optionally declare precedences for terminals, make declarations that will improve error-recovery, and suppress the generation of default reductions in the parser. You may declare whether the parser generator should create a verbose description of the parser in a “`.desc`” file such as `my.yacc.desc`. This is useful for debugging your parser and for finding the causes of shift/reduce errors and other parsing conflicts.

You may also declare whether the semantic actions are free of significant side-effects and always terminate. Normally, ML-Yacc delays the evaluation of semantic actions until the completion of a successful parse. This ensures that there will be no semantic actions to "undo" if a syntactic error-correction invalidates some semantic actions. If, however, the semantic actions are free of significant side-effects and always terminate, the results of semantic actions that are invalidated by a syntactic error-correction can always be safely ignored.

Parsers run faster and need less memory when it is not necessary to delay the evaluation of semantic actions. You are encouraged to write semantic actions that are free of side-effects and always terminate and to declare this information to ML-Yacc.

A semantic action is free of significant side-effects if it can be re-executed a reasonably small number of times without affecting the result of a parse. (The re-execution occurs when the error-correcting parser is testing possible corrections to fix a syntax error, and the number of times re-execution occurs is roughly bounded, for each syntax error, by the number of terminals times the amount of lookahead permitted for the error-correcting parser).

9.4.1 `%name`, required declaration

You must specify the name of the parser with command `%name` *name*. If you decide to call your parser "*My*`Parser`" then you will need the declaration:

```
87  %name My
```

This declaration must agree with the ML-Lex command `%header`, chapter 7.2.2. See also the glue code in lines 389, 391 and 393 which must be in agreement.

9.4.2 `%nonterm` and `%term`, required declaration

You must define the terminal and non-terminal sets using the `%term` and `%nonterm` declarations, respectively. These declarations are like an ML datatype definition. The type of the supplemental payload value that a symbol may carry is defined at the same time that the symbol is defined. Each declaration consists of the keyword (`%term` or `%nonterm`) followed by a list of symbol entries separated by a vertical bar "`|`".

Each symbol entry is a symbol name which may be followed by an optional "`of` <*ML-type*>". The types cannot be polymorphic. Those symbol entries without a type carry no supplemental payload; just the basic left and right positions. Non-terminal and terminal names must be disjoint and no name may be declared more than once in either declaration.

The symbol names and types are used to construct a datatype union for the payload values on the semantic stack in the LR parser and to name the values associated with subcomponents of a rule. The names and types of terminals are also used to construct a signature for a structure that may be passed to the lexer functor.

Because the types and names are used in these manners, do not use ML keywords as symbol names. The programs produced by ML-Yacc will not compile if ML keywords are used as symbol names. Make sure that the types specified in the `%term` declaration are fully qualified types or are available in the background environment when the signatures produced by ML-Yacc are loaded. Do not use any locally defined types from the user declarations section of the specification.

These requirements on the types in the `%term` declaration are not a burden. They force the types to be defined in another module[16], which is a good idea since these types will be used in the lexer module.

Let's have a look at some of the terminals and non-terminals defined for the SML/NJ compiler in file `ml.grm`:

```
88  %term EOF | SEMICOLON
89      | ID     of FastSymbol.raw_symbol
90      | INT    of IntInf.int
91      | WORD   of IntInf.int
92      | REAL   of string
93      | STRING of string
94      | CHAR   of string
          ⋮
95  %nonterm elabel of (symbol * exp)
96         | id     of FastSymbol.raw_symbol
97         | int    of IntInf.int
98         | tycon  of symbol list
99         | ty     of ty
100        | match  of rule list
101        | rule   of rule
102        | exp    of exp
             ⋮
```

Note that the terminals defined on lines 88–94 have upper case symbols and the non-terminals defined in lines 95–102 have lower case symbols. We recommend following this well-known convention.

The symbols `EOF` and `SEMICOLON` on line 88 have no supplemental payload; only the basic left and right positions. Terminal `REAL` on line 92 carries a supplemental payload in addition to the positions: a string representing the real number. Remembering from figure 1 on page 2 that the terminal tokens form the output of the lexer, let's check that the lexer is loading up the payload correctly. In file `ml.lex` we see:

```
103 {real} => (Tokens.REAL(yytext,
104                        yypos,
105                        yypos+size yytext));
```

On line 103, the supplemental payload `yytext` is of type `string` as expected. The left and right positions on lines 104 and 105 have the type defined by command `%pos`, which in the SML/NJ compiler is `int`. *Oof!*

On line 95, the supplemental payload is a tuple. The parentheses are required.

9.4.3 `%pos`, required declaration

You must declare the type of basic payload position values using the `%pos` declaration. The syntax is `%pos` *ML-type*. This type MUST be the same type as that which is actually found in the lexer. It cannot be polymorphic.

[16]For example the `DataTypes` structure in file `datatypes.sml` in our working example. See chapter 11.2.2 on page 38.

The basic payload of a token is often two integers, and now is the time to say that they are integers. In the SML/NJ compiler we find:

```
106 %pos int
```

For example, line 106 declares that the basic payload of the token `REAL` on lines 104 and 105 consists of integers.

9.4.4 `%arg`

You may want each invocation of the entire parser to be parameterised by a particular argument, such as the file name of the input being parsed in an invocation of the parser. The `%arg` declaration allows you to specify such an argument. (This is often cleaner than using "global" reference variables.) The declaration

`%arg` *Any-ML-pattern* : *ML-type*

specifies the argument to the parser, as well as its type. If `%arg` is not specified, it defaults to `() : unit`.

Note that ML-Lex also has a `%arg` directive, but the two are independent and may have different types.

For example:

```
%arg (fileName) : string
```

says that the file name used on line 417 on page 43 is a string.

9.4.5 `%eop` and `%noshift`

You should specify the set of terminals that may follow the start symbol, also called end-of-parse symbols, using the `%eop` declaration. The `%eop` keyword should be followed by the list of terminals. This is useful, for example, in an interactive system where you want to force the evaluation of a statement before an end-of-file (remember, a parser delays the execution of semantic actions until a parse is successful).

ML-Yacc has no concept of an end-of-file. You must define an end-of-file terminal (`EOF`, perhaps) in the `%term` declaration. You must declare terminals which cannot be shifted, such as end-of-file, in the `%noshift` declaration. The `%noshift` keyword should be followed by the list of non-shiftable terminals. An error message will be printed if a non-shiftable terminal is found on the right hand side of any rule, but ML-Yacc will not prevent you from using such grammars.

It is important to emphasise that *non-shiftable terminals must be declared.* The error-correcting parser may attempt to read past such terminals while evaluating a correction to a syntax error otherwise. This may confuse the lexer. For example:

```
%eop EOF SEMICOLON
%noshift EOF
```

9.4.6 `%header`

This facility is for advanced users. Novice users, and users of our worked example, chapter 11.2, should omit this directive and take the default value.

You may define code to head the functor {*parser name*}`LrValsFun` here. This may be useful for adding additional parameter structures to the functor. The functor must be parameterised by the `Token` structure, so the `%header` declaration should always have the form:

```
%header (functor MyLrValsFun(structure Token : TOKEN
                              ...)
         )
```

This directive is not used in the SML/NJ compiler, neither is it used in our working example.

9.4.7 `%left`, `%right`, `%nonassoc`

You should list the precedence declarations in order of increasing (tighter-binding) precedence. Each precedence declaration consists of a % keyword specifying associativity followed by a list of terminals. You may place more than one terminal at a given precedence level, but you cannot specify non-terminals. The keywords are `%left`, `%right`, and `%nonassoc`, standing for their respective associativities.

For example, here are the precedence declarations used in the SML/NJ compiler:

```
%nonassoc WITHTYPE
%right AND
%right ARROW
%right DARROW
%left DO
%left ELSE
%left RAISE
%right HANDLE
%left ORELSE
%left ANDALSO
%right AS
%left COLON
```

9.4.8 `%nodefault`

The `%nodefault` declaration suppresses the generation of default reductions. If only one production can be reduced in a given state in an LR table, it may be made the default action for the state. An incorrect reduction will be caught later when the parser attempts to shift the lookahead terminal which caused the reduction. ML-Yacc usually produces programs and verbose files with default reductions. This saves a great deal of space in representing the LR tables, but sometimes it is useful for debugging and advanced uses of the parser to suppress the generation of default reductions. Novice users should omit this declaration.

9.4.9 `%pure`

Include the `%pure` declaration if the semantic actions are free of significant side effects and always terminate. It is suggested that you begin developing your language without this directive.

9.4.10 %start

You may define the start symbol using the %start declaration. Otherwise the non-terminal for the first rule will be used as the start non-terminal. The keyword %start should be followed by the name of the starting non-terminal. This non-terminal should not be used on the right hand side of any rules, to avoid conflicts between reducing to the start symbol and shifting a terminal. ML-Yacc will not prevent you from using such grammars, but it will print a warning message.

The SML/NJ compiler has the declaration:

```
%start interdec
```

but simpler languages often omit the %start directive and start with the non-terminal of the first rule, for example with the begin at line 127 in

```
126 %%
127 begin: procList  ((Pi procList))
    ⋮
```

9.4.11 %verbose

Include the %verbose declaration to produce a verbose description of the LALR parser. The name of this file is the name of the specification file with a ".desc" appended to it, for example pi.yacc.desc.

This file is helpful for debugging, and has the following format:

1. A summary of errors found while generating the LALR tables.
2. A detailed description of all errors.
3. A description of the states of the parser. Each state is preceded by a list of conflicts in the state.

It is instructive to use this directive, and then to trace the operation of the parser, which will lead you through the ".desc" file. To obtain a trace, set DEBUG1 and DEBUG2 to true in the file parser2.sml which you will find in the ml-yacc directory in your SML/NJ distribution.

You can make the trace more agreeable by making the following modification to parser2.sml:

1. Define the function waitForKeyStroke which waits for you to hit the enter key:

   ```
   val waitForKeyStroke:unit->unit = fn ()
      => ignore (TextIO.input1 TextIO.stdIn);
   ```

2. Replace the line

   ```
   case action
   ```

   with

   ```
   (case action
   ```

3. Replace the line

```
      | ACCEPT => println "ACCEPT")
```

with

```
      | ACCEPT => println "ACCEPT");
  waitForKeyStroke()
)
```

This modification will allow you to single step through the trace.

9.4.12 `%keyword`, error recovery

Specify all keywords in your grammar here. The `%keyword` should be followed by a list of terminal names. Fixes involving keywords are generally dangerous; they are prone to substantially altering the syntactic meaning of the program. They are subject to a more rigorous parse check than other fixes.

The `%keyword` declaration for the SML/NJ compiler begins:

```
%keyword ABSTYPE AND AS CASE DATATYPE DOTDOTDOT ELSE END
           ⋮
```

What turns an identifier into a keyword? See chapter 7.1.3 on page 10.

9.4.13 `%prefer`, error recovery

List terminals to prefer for insertion after the command `%prefer`. Corrections which insert a terminal on this list will be chosen over other corrections, all other things being equal.

9.4.14 `%subst`, error recovery

This declaration should be followed by a list of clauses of the form

terminal `for` *terminal*

where items on the list are separated using a `|`. The substitution corrections on this list will be chosen over all other corrections except preferred insertion corrections (see 9.4.13 above), all other things being equal.

9.4.15 `%change`, error recovery

This is a generalisation of `%prefer` and `%subst`. It takes the following syntax:

$$tokens_{1a} \texttt{ -> } tokens_{1b} \texttt{ | } tokens_{2a} \texttt{ -> } tokens_{2b} \texttt{ | } \ldots$$

where each *tokens* is a (possibly empty) sequence of tokens. The idea is that any instance of $tokens_{1a}$ can be "corrected" to $tokens_{1b}$, and so on.

For example, to suggest that a good error correction to try is `IN ID END` (which is useful for the ML parser), write

```
%change  -> IN ID END
```

9.4.16 %value, error recovery

The error-correction algorithm may also insert terminals with values. You must supply a value for such a terminal. The keyword should be followed by a terminal and a piece of code (enclosed in parentheses) that when evaluated supplies the value. There must be a separate %value declaration for each terminal with a value that you wish may be inserted or substituted in an error correction. The code for the value is not evaluated until the parse is successful.

Do not specify a %value for terminals without supplemental payload values. This would result in a type error in the program produced by ML-Yacc.

The %value declarations for the SML/NJ compiler include:

```
%value INT (IntInf.fromInt 1)
%value REAL ("0.0")
%value STRING ("")
```

9.5 ML-Yacc rules

The rules section contains the context-free grammar productions and their associated semantic actions.

All rules are declared in the final section, after the last %% delimiter. A rule consists of a left hand side non-terminal, followed by a colon, followed by a list of right hand side clauses.

The right hand side clauses should be separated by bars ("|"). Each clause consists of a list of non-terminal and terminal symbols, followed by an optional %prec declaration, and then followed by the code to be evaluated when the rule is reduced.

The optional %prec consists of the keyword %prec followed by a terminal whose precedence should be used as the precedence of the rule.

The values of those symbols in a right hand side clause which have values are available inside the code. For example, in rule

```
141 path: IDE   ((Name (IDE,fileName,IDEleft,IDEright)))
```

the non-terminal left hand side is `path` and the right hand side clause is everything after the colon. Within the clause the list of symbols is just the "`IDE`" and the code is a `Name` datatype. The value of `IDE` is a string (see line 354 on page 41) which is used in the `Name` datatype.

Each position value has the general form {*symbol name*}{n+1}, where {n} is the number of occurrences of the symbol to the left of the symbol. If the symbol occurs only once in the rule, {*symbol name*} may also be used. For example, if in rule "`path`" above, there had been two `IDE`'s in the list of symbols, we could have referred to their values as `IDE1` and `IDE2`.

Positions for all the symbols are also available. The payload positions are given by {*symbol name*}{n+1}`left` and {*symbol name*}{n+1}`right`. where {n} is defined as before. For example we see the use of `IDEleft` and `IDEright` on line 141.

If in rule "`path`" above, there had been two `IDE`'s in the list of symbols, we could have referred to their left and right positions as `IDE1left`, `IDE1right`, `IDE2left` and `IDE2right`.

The position for a null right-hand-side of a production is assumed to be the leftmost position of the lookahead terminal which is causing the reduction. This position value is available in `defaultPos`.

The value to which the code evaluates is used as the value of the non-terminal. The type of the value and the non-terminal must match. The value is ignored if the non-terminal has no value, but is still evaluated for side-effects.

An example will make this clearer. Assume that a language contains a statement of people's lifetime, with a starting and ending year. The ML datatype declarations might be:

```
142 datatype Life = Life of Year * Year        * int * int
143      and Year = Year of int * int * int    * int * int
```

The two integers at the end of each of lines 142 and 143 give the position of the constructions in the source file. They will be needed for error messages.

The tokens representing years, months and days will have a supplemental payload of one integer; they are declared in the ML-Yacc declarations section of the file `my.yacc` as:

```
%term YMD of int
```

This means that the tokens `YMD` generated by the lexer will have a payload of type `int * int * int`. The lexer rules in file `my.lex` might be

```
{int} => (T.YMD(stoi yytext,!line,!col))
```

where function `stoi : string -> int` converts a string of digits to the corresponding integer.

Parser rules in file `my.yacc` will pull these three integers together to form the two years and the life:

```
146 life: year HYPHEN year
147              ((Life (year1,year2,year1left,year1right)))
148 year: YMD COLON YMD COLON YMD
149              ((Year (YMD1,YMD2,YMD3,YMD1left,YMD1right)))
```

Note in line 147 how the two (non-terminal) years are addressed, and how the left position and and right position of the first year are picked out. In line 149 similar addressing is used for the `YMD` terminals.

9.5.1 Heavy payload

How does the addressing work in the presence of a heavy payload, i.e. a payload with more than one value?

Imagine the previous example, but in a legacy, pre-Y2K situation. The year is specified by the user as only two digits, with the century added by the lexer. (This is very bad design, but its an example of a situation one may face.)

The tokens representing years, months and days in the ML-Yacc declarations section of the file `my.yacc` become:

```
150 %term Y  of int * int
151     | MD of int
```

Line 150 says that the tokens `Y` generated by the lexer will have a payload of type `(int * int) * int * int`. The lexer rules in file `my.lex` become

```
{int}     => (T.Y((1900,stoi yytext),!line,!col))
"-"{int} => (T.MD(stoi yytext,!line,!col))
```

where the hard-wired value on line 152 for the year *1900* is amongst the worst programming we have seen in a long while.

The parser rules in file `my.yacc` have to be modified to pull these tokens together:

```
154 life: year HYPHEN year
155                 ((Life (year1,year2,year1left,year1right)))
156 year: Y MD MD
157                 ((Year (#1(Y)+#2(Y),MD1,MD2,Yleft,Yright)))
```

Note on line 157 that the components of supplemental payload are addressed as `#1(·)`, `#2(·)`, ..., see [Ull98, ch 7.1.3] to find out why.

9.5.2 `%prec`, precedence and associativity

If the programmer writes `2 * 3 - 5`, we would expect the expression to evaluate to 1, not -4, but how do we tell ML-Yacc that this is what we want?

Common mathematical usage associates a *precedence* with arithmetic operators. We expect a multiplication to be performed before an addition. Even at the same level of precedence, there can be ambiguities: for example, does `2-3-4` equal -5 or 3? In other words, is `2-3-4` equal to `(2-3)-4` or `2-(3-4)`? This is clarified by specifying the *associativity* of the operator "`-`".

ML-Yacc provides declarations to specify both the precedence and the associativity. The declaration is in two parts.

1. Attach precedence and associativity to terminals using the commands `%left`, `%right` and `%nonassoc`. Here are the declarations used in an on-going project.

```
%nonassoc CATCH
%right    EXCL
%right    EQ
%nonassoc DCOLON
%nonassoc ORELSE
%nonassoc ANDALSO
%nonassoc EE NE LE LT GE GT EXACTEE EXACTNE
%right    PLUSPLUS MINUSMINUS RARROW
%left     PLUS MINUS BOR BXOR BSL BSR OR XOR
%left     MULT DIV SOL ASTR REM BAND AND
%nonassoc BNOT NOT
%nonassoc NUMB
%left     FUN_APPL
%nonassoc COLN
```

   The tightest binding operators are at the bottom of the list, so we see that multiplication, `MULT`, binds more tightly than addition, `PLUS`. We see also that subtraction, `MINUS`, is left associative so `2-3-4` is equal to `(2-3)-4`.

2. Add `%prec` declarations to the rules to say which terminal's precedence and associativity are to be used with which rules. The multiplication, addition and subtraction rules in the ongoing project are specified as:

```
| e MULT e    %prec MULT     (Fa ...
| e PLUS e    %prec PLUS     (Fa ...
| e MINUS e   %prec MINUS    (Fa ...
```

The language specified by this project requires that function application be left associative as in ML, i.e. `f g 2` means `(f g) 2`. Now function application is a non-terminal, `fun_appl`, in the ML-Yacc specification, and non-terminals cannot be placed in the `%left`, `%right` and `%nonassoc` declarations — what can we do? The solution is to create a new terminal, `FUN_APPL`, by declaring it in the `%term` declaration, and then defining the required precedence and associativity on line 170. The corresponding rule becomes:

```
175 fun_appl:   (* Function application has precedence
176                and associativity defined by dummy
177                terminal FUN_APPL. *)
178   name e      %prec FUN_APPL  ((name,e))
```

10 Standalone operation of ML-Lex and ML-Yacc

ML-Lex can be run either as a stand-alone program, or it can be integrated into a larger project. This chapter discusses the standalone operation. A complete worked example of ML-Lex integrated into a project is given in chapter 11.2 on page 35.

10.1 ML-Lex as a stand-alone program

Let the name for the parser given in the `%name` declaration, chapter 9.4.1, be denoted by *My*, the ML-Lex specification file name be denoted by `my.lex`, and the ML-Yacc specification file name be denoted by `my.yacc`.

The parser generator creates a functor named *My*`LrValsFun` for the values needed for a particular parser. This functor is placed in file `my.yacc.sml`. It contains a structure `Tokens` which allows you to construct terminals from the appropriate values. The structure has a function for each terminal that takes a tuple consisting of the supplemental payload value for the terminal (if there is any), and then the basic payload, a leftmost and a rightmost position for the terminal, and constructs the terminal from these values.

ML-Yacc also creates a signature *My*`_TOKENS` for the structure `Tokens`, and a signature *My*`_LRVALS` for the structure produced by applying the functor *My*`LrValsFun`. These two signatures are placed in `my.yacc.sig`.

Use the signature *My*`_TOKENS` to create a functor for the lexical analyser which takes the structure `Tokens` as an argument. The signature *My*`_TOKENS` will not change unless the `%term` declaration in a specification is altered by adding terminals or changing the types of terminals. You do not need to recompile the lexical analyser functor each time the specification for the parser is changed if the signature *My*`_TOKENS` does not change.

If you are using ML-Lex to create the lexical analyser, you can turn the lexer structure into a functor using the `%header` declaration. `%header` allows you to define the header for a structure body.

Add the following declaration to the specification `my.lex` for the lexical analyser:

```
179 %header (functor MyLexFun(structure Tokens : My_TOKENS))
```

Now declare the type of position values for terminals. Let's assume that your positions are integers, which is often the case. In the user definitions section of `my.lex`:

```
180 type pos = int
```

and in the ML-Yacc declarations section in `my.yacc`:

```
%pos int
```

These two declaration must be in agreement. Note, however, that this type is not available in the `Tokens` structure that parameterises the lexer functor.

Include the following glue code in the user definitions section of `my.lex`:

```
type svalue = Tokens.svalue
type ('a,'b) token = ('a,'b) Tokens.token
type lexresult  = (svalue,pos) token
```

These types are used to give lexers signatures.

You may use a lexer constructed using ML-Lex with the `%arg` declaration, but you must follow special instructions for tying the parser and lexer together.

10.2 Running ML-Lex standalone

From the Unix shell, run `sml-lex` `my.lex`. You will find the output in file `my.lex.sml`. The extension `.lex` is not required but is recommended.

If you are running ML-Lex within an interactive system (note that this is not the preferred method): Use `lexgen.sml`; this will create a structure `LexGen`. The function `LexGen.lexGen` creates a program for a lexer from an input specification. It takes a string argument – the name of the file containing the input specification. The output file name is determined by appending "`.sml`" to the input file name.

10.3 ML-Yacc as a standalone program

ML-Yacc may be used from the interactive system or built as a stand-alone program which may be run from the Unix command line. See the file `README` in the `mlyacc` directory for directions on installing ML-Yacc. We recommend that ML-Yacc be installed as a stand-alone program.

If you are using the stand-alone version of ML-Yacc, invoke the program "`sml-yacc`" with the name of the specification file, say `my.yacc`. If you are using ML-Yacc in the interactive system, load the file "`smlyacc.sml`". The end result is a structure `ParseGen`, with one value `parseGen` in it. Apply `parseGen` to a string containing the name of the specification file, e.g. `ParseGen.parseGen my.yacc`.

Two files will be created, `my.yacc.sig` and `my.yacc.sml`.

10.4 Using the program produced by ML-Lex

When the output file `my.lex.sml` is loaded, it will create a structure `Mlex` that contains the function `makeLexer` which takes a function from `int` → `string` and returns a lexing function:

```
val makeLexer : (int->string) -> yyarg -> lexresult
```

where `yyarg` is the type given in the ML-Lex `%arg` directive, chapter 7.2.7, or `unit` if there is no `%arg` directive.

For example,

```
val lexer = Mlex.makeLexer (inputc (open_in "f"))
```

creates a lexer that operates on the file whose name is `f`.

When the ML-Lex `%posarg` directive, see chapter 7.2.6, is used, then the type of `makeLexer` is

```
val makeLexer : ((int->string)*int) -> yyarg -> lexresult
```

where the extra `int` argument is one less than the `yypos` of the first character in the input. The value k would be used, for example, when creating a lexer to start in the middle of a file, when k characters have already been read. At the beginning of the file, $k = 0$ should be used.

The `int` $\rightarrow$ `string` function should read (grab) a string of characters from the input stream. It should return a null string to indicate that the end of the stream has been reached. The integer is the number of characters that the lexer wishes to read; the function may return any non-zero number of characters. For example,

```
val lexer =
  let val input_line = fn f =>
        let fun loop result =
           let val c = input (f,1)
               val result = c :: result
           in if String.size c = 0 orelse c = "\n"
              then String.implode (rev result)
              else loop result
           end
        in loop nil
        end
  in Mlex.makeLexer (fn n => input_line std_in)
  end
```

is appropriate for interactive streams where prompting, etc. occurs; the lexer won't care that `input_line` might return a string of more than or less than n characters.

The lexer tries to read a large number of characters from the input function at once, and it is desirable that the input function return as many as possible. Reading many characters at once makes the lexer more efficient. Fewer input calls and buffering operations are needed, and input is more efficient in large block reads. For interactive streams this is less of a concern, as the limiting factor is the speed at which the user can type.

To obtain a value, invoke the lexer by passing it a unit:

```
val nextToken = lexer()
```

If one wanted to restart the lexer, one would just discard `lexer` and create a new lexer on the same stream with another call to `makeLexer`. This is the best way to discard any characters buffered internally by the lexer.

All code that is declared in the ML-Lex user declarations section is placed inside a structure `UserDeclarations`. If you want to access this structure, use the path name `Mlex.UserDeclarations`.

If any input cannot be matched, the program will raise the exception `Mlex.LexError`. An internal error (could be a bug) will cause the exception `Internal.LexerError` to be raised.

If `%structure` is used, chapter 7.2.3, remember that the structure name will no longer be `Mlex`, but the one specified in the `%structure` command.

11 Examples

11.1 A calculator

Here is a sample lexer for a calculator program. First the user declarations:

```
202 datatype lexresult = DIV | EOF | EOS | ID of string
203                    | LPAREN | NUM of int | PLUS | PRINT
204                    | RPAREN | SUB | TIMES
205 val linenum = ref 1
206 val error = fn x => output(std_out,x ^ "\n")
207 val eof = fn () => EOF
```

These are the four basic declarations we expect to see. In line 202 the output is much simpler than the type defined on line 9. The parser will have to be hand-crafted to use this result.

Now the ML-Lex definitions:

```
208 %%
209 %structure CalcLex
210 alpha = [A-Za-z];
211 digit = [0-9];
212 ws    = [\ \t];
```

The character set is 7-bit "ASCII". Here are the rules:

```
213 %%
214 \n        => (inc linenum; lex());
215 {ws}+     => (lex());
216 "/"       => (DIV);
217 ";"       => (EOS);
218 "("       => (LPAREN);
219 {digit}+ => (NUM (revfold (fn(a,r)=>ord(a)-ord("0")+10*r)
220                           (explode yytext) 0));
221 ")"       => (RPAREN);
222 "+"       => (PLUS);
223 {alpha}+ => (if yytext="print" then PRINT else ID yytext);
224 "-"       => (SUB);
225 "*"       => (TIMES);
226 .         => (error ("calc: ignoring bad character "
227                     ^yytext);
228              lex());
```

In line 226 note the practice of placing the catch-all period as the last rule.

Here is the parser for the calculator:

```
(* Sample interactive calculator to demonstrate use of lexer.
   The original grammar was
       stmt_list -> stmt_list stmt
       stmt -> print exp ;  | exp ;
       exp -> exp + t | exp - t | t
       t -> t * f | t/f | f
       f -> (exp) | id | num
  The function parse takes a stream and parses it for
  the calculator program.
```

```
  If a syntax error occurs, parse prints an error message
  and calls itself on the stream.  On this system that has
  the effect of ignoring all input to the end of a line.
*)

structure Calc =
 struct
   open CalcLex
   open UserDeclarations
   exception Error
   fun parse strm =
    let
      val say = fn s => output(std_out,s)
      val input_line = fn f =>
          let fun loop result =
             let val c = input (f,1)
         val result = c :: result
             in if String.size c = 0 orelse c = "\n"
   then   String.implode (rev result)
        else loop result
     end
          in loop nil
          end
      val lexer = makeLexer (fn n => input_line strm)
      val nexttok = ref (lexer())
      val advance = fn () => (nexttok := lexer(); !nexttok)
      val error = fn () => (say ("calc: syntax error on line"
                            ^(makestring(!linenum)) ^ "\n");
                            raise Error)
      val lookup = fn i =>
        if i = "ONE" then 1
        else if i = "TWO" then 2
        else  (say ("calc: unknown identifier '" ^ i ^ "'\n");
               raise Error)
     fun STMT_LIST () =
         case !nexttok of
            EOF => ()
          | _ => (STMT(); STMT_LIST())

     and STMT() =
         (case !nexttok
           of EOS  => ()
            | PRINT => (advance();
                        say ((makestring (E():int)) ^ "\n");
                        ())
            | _ => (E(); ());
         case !nexttok
           of EOS => (advance())
            | _ => error())
     and E () = E' (T())
     and E' (i : int ) =
         case !nexttok of
            PLUS => (advance (); E'(i+T()))
          | SUB => (advance (); E'(i-T()))
```

```
          | RPAREN => i
          | EOF => i
          | EOS => i
          | _ => error()
     and T () =  T'(F())
     and T' i =
        case !nexttok of
            PLUS => i
          | SUB => i
          | TIMES => (advance(); T'(i*F()))
          | DIV => (advance (); T'(i div F()))
          | EOF => i
          | EOS => i
          | RPAREN => i
          | _ => error()
     and F () =
        case !nexttok of
            ID i => (advance(); lookup i)
          | LPAREN =>
              let val v = (advance(); E())
              in if !nexttok = RPAREN
                 then (advance (); v)
                 else error()
              end
          | NUM i => (advance(); i)
          | _ => error()
    in STMT_LIST () handle Error => parse strm
    end
end
```

11.2 ML-Lex and ML-Yacc in a larger project

In this chapter we show a working example of a very small language processed by a lexer and parser produced by ML-Lex and ML-Yacc as part of a larger project.

Joe Grunt works on into the night *'Cos his boss says a raise is in sight!* *But what Joe doesn't know* *Is that Joe Grunt must go.* *He'll be fired once his code's working right.*

We want to emphasise that this is a working example. If you place the code into a directory, start ML and then type `CM.make "pi.cm";` in that directory, you will build the project. You can run the two sample programs by typing `Pi.compile "good.pi";` and `Pi.compile "bad.pi";`.

Let's assume that your term project, your professional deliverable or your lifetime software ambition is large enough to fill several programs. This makes it of interest to use the SML/NJ Compilation Manager (CM). The documentation for the Compilation Manager [Blu02] is included in the distribution as files `CM-new.ps` and `CM-new.pdf`. The syntax has changed with SML/NJ version 110.40, so check to see if the syntax used in your version is the same as used in this chapter.

An added advantage of using CM is that it is aware of ML-Lex and ML-Yacc, and does a very good job of integrating ML-Lex and ML-Yacc into the project's build process.

The project consists of the following files:

`pi.cm` Provides a list of files that the SML/NJ Compilation Manager (CM) will use to build the project, chapter 11.2.1

`datatypes.sml` The ML datatype declarations for the elements in the parse tree, chapter 11.2.2

`pi.lex` The specification for the lexer, chapter 11.2.3

`pi.yacc` The specification for the parser, chapter 11.2.4

`glue.sml` The glue code that ties the lexer and parser to the project, chapter 11.2.5.

`compiler.sml` A simple driver that will read the source program and display the corresponding parse tree, chapter 11.2.6.

`good.pi` An example of a valid program, line 432.

`bad.pi` You guessed, an example of a invalid program, line 434.

We now review the project, file by file.

```
(* pi.cm  Build project *)
Library
   structure Pi
is
   datatypes.sml      (* Lot of datatypes *)
   pi.lex             (* Lexer rules. *)
   pi.yacc:MLYacc     (* Parser rules. *)
   glue.sml           (* Build the parser *)
   compiler.sml       (* Lex, parse, panic... *)
   $/basis.cm         (* SML/NJ's Basis Library. *)
   $/ml-yacc-lib.cm   (* Code written by Lucent. *)
   $smlnj/compiler/compiler.cm  (* Structure Compiler. *)
```

Figure 4: File `pi.cm`.

11.2.1 File `pi.cm`

At its simplest, CM calls for you to place a list of all the files to be compiled in a file with extension `.cm`. We will call this file `pi.cm`. `pi.cm` is known as a *CM description file*. Then all you have to do is type `CM.make "pi.cm";` on the command line, and the compilation of your project files is done for you. *This includes much of the tricky business on integrating ML-Lex and ML-Yacc into your build.* Its not all plain sailing — you will need to

1. Package your program into a set of structures.
2. Point to some system programs that Lucent have provided for you.
3. Specify some glue.

To make things clearer, figure 4 shows an example of a `pi.cm` file. Line 233 will cause ML-Yacc to be run on `pi.yacc`, producing source files `pi.yacc.sig` and `pi.yacc.sml`, and line 232 will cause ML-Lex to be run on `ml.lex`, producing a source file `ml.lex.sml`. Then these files will be compiled after loading the necessary signatures and structures from the ML-Yacc library `ml-yacc-lib.cm`, as specified on line 237. The library `ml-yacc-lib.cm` is a part of the SML/NJ distribution. Lines 234 and 235 will then build the compiler using the parser and lexer.

The "$" sign in the entries on lines 236, 237 and 238 says that these entries are "anchored paths" and are resolved with respect to an "anchor environment" which you can read about in the CM documentation [Blu02]. For our purposes here, "$" means "well known to SML/NJ". You may omit the specification of `$smlnj/compiler/compiler.cm` on line 238. If you set a value for the `printDepth`, the structure `Compiler` will be loaded automatically.

Have a look at the code in `ml-yacc-lib.cm` and feel very glad that someone has done all this hard work for you.

Note on line 233 that the Compilation Manager needs to be told that a `.yacc` file extension is for an ML-Yacc file. CM understands that a `.lex` file extension is for an ML-Lex file.

```
(* datatypes.sml *)
signature DATATYPES =
sig datatype A      = A of Pat * Proc
    and      Pi     = Pi of Proc list
    and      Pat    = Pat of Path
    and      Path   = Name of string * string * int * int
    and      Proc   = New of Path * Proc
                    | Output of Path * V
                    | Input of Path * A
                    | Parallel of Proc list
    and      V      = V of Path
end;
```

Figure 5: File `datatypes.sml`, signature `DATATYPES`.

```
structure DataTypes : DATATYPES =
struct
  datatype  A     = A of Pat*Proc
  and       Pi    = Pi of Proc list
  and       Pat   = Pat of Path
  and       Path  = Name of string * string * int * int
  and       Proc  = New of Path * Proc
                  | Output of Path * V
                  | Input of Path * A
                  | Parallel of Proc list
  and       V     = V of Path
end;
```

Figure 6: File `datatypes.sml`, structure `DataTypes`.

```
263 (* pi.lex *)
264 structure T = Tokens
265 
266 type pos = int
267 type svalue = T.svalue
268 type ('a,'b) token = ('a,'b) T.token
269 type lexresult = (svalue,pos) token
270 type lexarg = string
271 type arg = lexarg
272 
273 val lin = ref 1;
274 val col = ref 0;
275 val eolpos = ref 0;
276 
277 val badCh : string * string * int * int -> unit = fn
278     (fileName,bad,line,col) =>
279     TextIO.output(TextIO.stdOut,fileName^"["
280           ^Int.toString line^"."^Int.toString col
281           ^"] Invalid character \""^bad^"\"\n");
282 val eof = fn fileName => T.EOF (!lin,!col);
```

Figure 7: File pi.lex, user declarations.

11.2.2 File datatypes.sml

The parse tree which you obtain after your source file has been lexed and parsed will represent the language elements. These are best coded using ML `datatype` declarations placed in a separate structure. The signature for the structure may be in the same file or in a separate file. In our case, the two are in the same file. First, we see in figure 5 the signature `DATATYPES`, followed by the structure `DataTypes` in figure 6. Line 254 shows the “top” node of the parse tree. The output of the parser will be of type `DataTypes.Pi`. How do we know this? Look at line 371 on page 42 in file `pi.yacc` and line 399 on page 43 in file `compiler.sml`.

One of the advantages of developing a language processor in a statically typed language such as SML is that when a new language feature is introduced, it is sufficient to make the corresponding change in `datatypes.sml`, type `CM.make "pi.cm";` on the command line, and see in the messages from the compiler, all the places in the project which will need attention.

11.2.3 File pi.lex

The lexer specification in file `pi.lex` is in three sections which we now review.

11.2.3.1 File pi.lex user declarations The ML-Lex user declarations are shown in figures 7 and 8. The abbreviation on line 264 saves a lot of typing. Line 266 declares the type of the position in the tokens. Line 273 declares the line pointer. Line 274 declares a variable to hold the column number, and line 275 declares a variable to hold the character position of most recent newline. On line 277 we find the routine needed to print an error message if an unwelcome character is found in the input stream. Finally,

```
283 (* Keyword management *)
284 structure KeyWord :
285 sig val find : string ->
286                (int * int -> (svalue,int) token) option
287 end =
288 struct
289  val TableSize =  422 (* 211 *)
290  val HashFactor = 5
291  val hash = fn
292      s => List.foldr (fn (c,v) =>
293        (v*HashFactor+(ord c)) mod TableSize) 0 (explode s)
294  val HashTable = Array.array(TableSize,nil) :
295               (string * (int * int -> (svalue,int) token))
296               list Array.array
297  val add = fn
298      (s,v) => let val i = hash s
299               in Array.update(HashTable,i,(s,v)
300                  :: (Array.sub(HashTable, i)))
301               end
302  val find = fn
303      s => let val i = hash s
304               fun f ((key,v)::r) = if s=key then SOME v
305                                    else f r
306                 | f nil = NONE
307           in  f (Array.sub(HashTable, i))
308           end
309  val _ = (List.app add [
310       ("new",        T.NEW)
311      ])
312 end;
313
314 open KeyWord;
```

Figure 8: File pi.lex, user declarations, continued.

line 282 provides end-of-file management. Note that since %arg is specified on line 318, the function eof takes the lexer argument fileName as an argument.

In figure 8 the structure KeyWord defined in lines 284 *et seq.* provides keyword management. The language has only one keyword — yours may have more, but you probably won't have as many as SQL. Line 58 on page 11 offers suggestions for some more possible keywords.

11.2.3.2 File pi.lex definitions The definitions in this section are shown in figure 9, and described in detail in chapter 7.2 on page 11.

From lines 316 and 320, we see that the input language uses the ISO Latin 9 character set; see chapter 2 and appendix A. Line 323 says that white space is made of spaces and horizontal tabs, and line 324 shows three end of line sequences suitable for a variety of operating systems.

```
%%
%full
%header (functor PiLexFun(structure Tokens: Pi_TOKENS));
%arg (fileName:string);
%s PI COMMENT;
alpha          = [A-Za-zŠšŽžŒ-ŸÀ-ÖØ-öø-ÿ]
hexa           = "0"("x"|"X")[0-9A-Fa-f];
digit          = [0-9];
ws             = [\ \t];
eol            = ("\013\010"|"\010"|"\013");
```

Figure 9: File pi.lex, ML-Lex definitions.

```
325 %%
326 <INITIAL>{ws}* => (lin:=1; eolpos:=0;
327                    YYBEGIN PI; continue ());
328 <PI>{ws}*  => (continue ());
329 <PI>{eol}  => (lin:=(!lin)+1;
330                eolpos:=yypos+size yytext; continue ());
331 <PI>{alpha}+ => (case find yytext of
332                     SOME v => (col:=yypos-(!eolpos);
333                                v(!lin,!col))
334                   | _      => (col:=yypos-(!eolpos);
335                                T.IDE(yytext,!lin,!col)));
336 <PI>"%"    => (YYBEGIN COMMENT; continue ());
337 <PI>"="    => (col:=yypos-(!eolpos); T.EQUALS(!lin,!col));
338 <PI>"("    => (col:=yypos-(!eolpos); T.LPAR(!lin,!col));
339 <PI>")"    => (col:=yypos-(!eolpos); T.RPAR(!lin,!col));
340 <PI>"!"    => (col:=yypos-(!eolpos); T.OUTPUT(!lin,!col));
341 <PI>"?"    => (col:=yypos-(!eolpos); T.INPUT(!lin,!col));
342 <PI>"||"   => (col:=yypos-(!eolpos); T.DVBAR(!lin,!col));
343 <PI>.      => (col:=yypos-(!eolpos);
344                badCh (fileName,yytext,!lin,!col);
345                T.ILLCH(!lin,!col));
346 <COMMENT>{eol} => (lin:=(!lin)+1;eolpos:=yypos+size yytext;
347                    YYBEGIN PI; continue ());
348 <COMMENT>. => (continue ());
```

Figure 10: File pi.lex, rules.

11.2.3.3 File pi.lex rules The rules in this section are shown in figure 10, and are described in detail in chapter 7.3 on page 13.

Note in line 326 that we specify that this analysis of white space applies only when the lexer begins. If you remove the "<INITIAL>", the rule will be considered to apply to comments as well, and will wrongly switch the lexer back to state <PI> if white space is met in a comment.

Line 336 specifies the "%" character as the comment marker. Any characters met between this marker and the next end-of-line will be ignored because of the rule on line 348. When the end of line sequence is met, the rule for {eol} on line 346 will switch the lexer back to state <PI>.

Note in lines such as 337 the recalculation of the value of the column position `col`.

The `ILLCH` token returned to the parser on line 345 says that there is no use for this character. The token `ILLCH` will be shown to the user — see line 442 on page 44 for an example — and the user is expected to guess that "`ILLCH`" means "you have a bogus character here; get rid of it". You will probably make life a lot easier for your users if you choose a better name, e.g. `BOGUS_CHARACTER`.

11.2.4 File `pi.yacc`

The parser specification in file `pi.yacc` is in three sections which we now review.

11.2.4.1 File `pi.yacc` user declarations The ML-Yacc user declarations are shown at the top of figure 11. All that is needed is to make the structure `DataTypes` in file `datatypes.sml` available to the parser.

```
349 (* pi.yacc *)
350 open DataTypes
351 %%
352 %name Pi
353 %term CARET | DVBAR | EOF | EQUALS
354     | IDE        of string
355     | ILLCH | INPUT | LPAR | NEW | OUTPUT | RPAR
356 %nonterm abs      of A          | begin    of Pi
357        | procList of Proc list  | parallel of Proc list
358        | pat      of Pat        | path     of Path
359        | pi       of Proc list  | proc     of Proc
360        | value    of V
361 %pos int
362 %eop EOF
363 %noshift EOF
364 %nonassoc DVBAR EOF EQUALS ILLCH INPUT
365           LPAR NEW OUTPUT RPAR
366 %nodefault
367 %verbose
368 %keyword NEW
369 %arg (fileName) : string
```

Figure 11: File `pi.yacc`, user declarations and ML-Yacc declarations.

11.2.4.2 File `pi.yacc` ML-Yacc declarations The ML-Yacc declarations are shown in figure 11 and are described in chapter 9.4. The declaration of the terminals on line 353 is important for the lexer since this defines the set of tokens that the lexer deliver to the parser. The terminal `ILLCH` on line 355 is a replacement for characters which are not used in the language. A better name for this terminal token would help the user to understand the error message.

Note on line 369 that the parser takes an argument. The value is passed on line 417.

11.2.4.3 File `pi.yacc` rules The parser rules are shown in figure 12 and described in chapter 9.5. The top-most rule is on line 371. The other lines provide recursive

```
370 %%
371 begin: procList          ((Pi procList))
372 abs: pat EQUALS proc     ((A (pat,proc)))
373 procList: proc procList ((proc::procList))
374 |                        ([])
375 parallel:
376   proc DVBAR parallel    ((proc::parallel))
377 | proc RPAR              ([proc])
378 | RPAR                   ([])
379 pat: path                ((Pat path))
380 path: IDE                ((Name (IDE,fileName,
381                                  IDEleft,IDEright)))
382 proc:
383   NEW path proc          ((New (path,proc)))
384 | path OUTPUT value      ((Output (path,value)))
385 | path INPUT abs         ((Input (path,abs)))
386 | LPAR parallel          ((Parallel parallel))
387 value: path              ((V path))
```

Figure 12: File `pi.yacc`, rules.

sub-rules for constructing a "`procList`".

Note on line 380 that since the `IDE` token has been defined (line 354) as having a supplemental payload, we may pick up the value of the payload and its position in the code on the right hand side of the rule.

```
388 (* glue.sml  Create a lexer and a parser *)
389 structure PiLrVals = PiLrValsFun(
390               structure Token = LrParser.Token);
391 structure PiLex     = PiLexFun(
392               structure Tokens = PiLrVals.Tokens);
393 structure PiParser = JoinWithArg(
394               structure ParserData = PiLrVals.ParserData
395               structure Lex=PiLex
396               structure LrParser=LrParser);
```

Figure 13: File `glue.sml`.

11.2.5 File `glue.sml`

The glue referred to in line 234 is in file `glue.sml` shown in figure 13. It builds the lexer and parser.

Since we specify the ML-Lex directive `%arg` in `pi.lex` (line 318), see chapter 7.2.7, then it is essential that on line 393 we specify `JoinWithArg` rather than `Join`. The rest shouldn't need much hacking, except for the name of your project which must match your ML-Lex `%header` declaration in `pi.lex`, chapter 7.2.2, and your ML-Yacc `%name` declaration in `pi.yacc`, chapter 9.4.1.

Note that it's on line 393 that the required parser is created as structure `PiParser`.

```
397 (* compiler.sml *)
398 structure Pi :
399 sig val compile : string -> DataTypes.Pi
400 end =
401 struct
402 exception PiError;
403 fun compile (fileName) =
404     let val inStream =  TextIO.openIn fileName;
405         val grab : int -> string = fn
406             n => if TextIO.endOfStream inStream
407                  then ""
408                  else TextIO.inputN (inStream,n);
409         val printError : string * int * int -> unit = fn
410             (msg,line,col) =>
411              print (fileName^"["^Int.toString line^":"
412                    ^Int.toString col^"] "^msg^"\n");
413         val (tree,rem) = PiParser.parse
414                      (15,
415                      (PiParser.makeLexer grab fileName),
416                      printError,
417                      fileName)
418             handle PiParser.ParseError => raise PiError;
419         (* Close the source program file *)
420         val _ = TextIO.closeIn inStream;
421     in tree
422     end
423 end;
```

Figure 14: File `compiler.sml`.

11.2.6 File `compiler.sml`

Now that we have a lexer and a parser, lets look at the front end of a possible compiler. See figure 14.

See chapter 8.3 for a discussion of the value 15 on line 414. On line 415, where we tell our parser to use our lexer, we

1. Specify the function `grab`: `int` → `string` shown at line 405.

2. Specify a value for the argument to our lexer, `fileName` since we have specified the `%arg` directive on line 318 in file `pi.lex`, chapter 7.2.7. Note that these two arguments are curried.

On line 417, we specify a value `fileName` for the argument to the parser, since we have specified the `%arg` directive on line 369 in file `pi.yacc`, chapter 9.4.4.

Although it is possible for the lexer and the parser to receive different arguments, the two are the same in our program.

11.2.7 Sample session

Here is a sample session. The lines of output have been folded if needed to fit on these pages. First we turn on SML and compile our project:

```
$ sml
Standard ML of New Jersey v110.44 [FLINT v1.5],
     November 6, 2003
- CM.make "pi.cm";
 ...
[New bindings added.]
val it = true : bool
-
```

Before we can run our project, we need some sample data: programs in the language defined by the parser rules. The language, if you are curious, is based on the π-calculus, but we expect that you will be replacing such an ivory tower language by real work.

Here are two short programs. The first, "`good.pi`", is valid:

```
%%% good.pi %%%
new a (a!x || a?x=())
```

but the second, "`bad.pi`", is invalid:

```
%%% bad.pi %%%
new new new ..
```

Now let's run the project on the two sample files:

```
436 - Pi.compile "good.pi";
437 GC #0.0.0.2.17.632:   (0 ms)
438 val it = Pi [New (Name #,Parallel #)] : ?.DataTypes.Pi
439 - Pi.compile "bad.pi";
440 bad.pi[2.12] Invalid character "."
441 bad.pi[2.13] Invalid character "."
442 bad.pi[2:12] syntax error: deleting  NEW NEW ILLCH
443 bad.pi[2:3] syntax error found at ILLCH
444 uncaught exception PiError
445   raised at: compiler.sml:44.49-44.56
446  ...
447 -
```

Where do these error messages come from?. The "Invalid character" messages at line 440 are issued by the lexer using function badCh defined at line 277 on page 38 in file pi.lex.

The "syntax error" messages at line 442 are issued by the parser using function printError defined at line 409 on page 43 in file pi.yacc. "ILLCH" is intended to say "this is a bogus character which has no place here".

11.2.7.1 Extra detail in result

If you would like to see more detail in the result, you will need to modify the `printDepth`. Add the following expression to the function `compile` which lives in file `compiler.sml`, just before calling function `PiParser.parse`.

```
val _ = Control.Print.printDepth:=12;
```

Now re-compile and re-run the compiler:

```
449 -  Pi.compile "good.pi";
450 val it =
451   Pi
452     [New
453        (Name ("a","good.pi",2,4),
454         Parallel
455           [Output (Name ("a","good.pi",2,7),
456                    V (Name ("x","good.pi",2,9))),
457            Input (Name ("a","good.pi",2,14),
458                   A (Pat (Name #),Parallel []))])]
459   : ?.DataTypes.Pi
460 -
```

You could also type

```
461 -  Compiler.Control.Print.printDepth:=12;
```

on the SML/NJ command line. With SML/NJ release 110.44 and later you type

```
462 -  Control.Print.printDepth:=12;
```

11.2.7.2 Garbage collection messages

If you don't like the SML/NJ garbage collection messages on line 437, then to turn them off, add the following declaration to the description file `pi.cm`

```
463 $/smlnj-lib.cm     (* SML/NJ goodies *)
```

and then add the following expression to the top of function `compile` in file `compiler.sml`.

```
464 val _ = SMLofNJ.Internals.GC.messages false;
```

11.3 ML-Lex and those exotic character encodings

In the dark days of the past, program files were "ASCII": they used one of the "ASCII" character sets and were encoded one character per byte as defined by whichever "ASCII" was in use. ISO Latin 1 and its relatives added new characters and letters with accents, but the encoding was still one character per byte. There were many attempts to get beyond the limit of 256 characters but this led to a cacophony of different character sets which assigned the same numbers to different characters. The light now shines from the Unicode Consortium which works to provide a unique number for every character. However the question of the encoding for these numbers remains, and a variety of different encodings are in use: e.g. UTF-8, UTF-16. This chapter shows a technique for handling Unicode with 8-bit ML-Lex, which works with a wide variety of encodings.

ML-Lex provides native support for 7-bit and 8-bit character sets but not for larger or more exotic character sets. In other words ML-Lex sees all its input as a stream of 8-bit characters, one per octet. In order to use the full Unicode range [TUC03] we will emulate the UTF-32 character encoding using 4-tuples of 8-bit characters. I.e. we will block the input stream *4 octets at a time*. Each block of 4 octets will be taken to represent a 4×8 bit $= 32$ bit integer which is read as a Unicode code position.

Figure 15 shows the 6 step process:

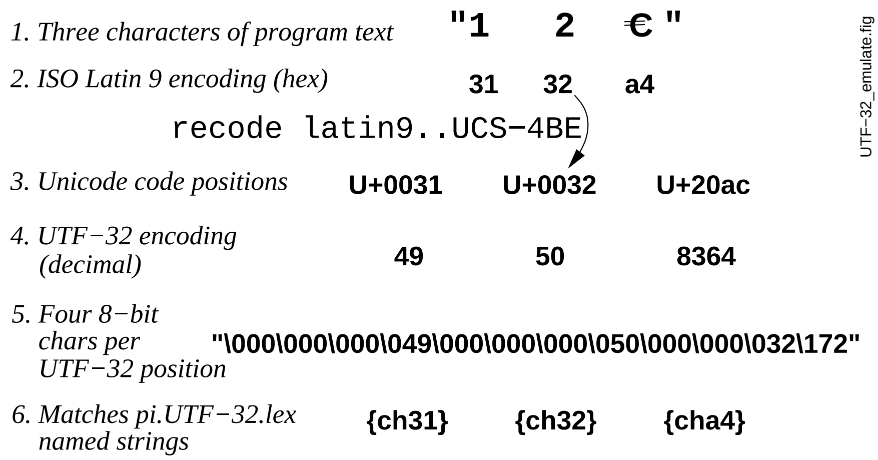

Figure 15: Use 4-character strings to emulate UTF-32 32 bit integers.

Step 1 : It is essential that you get the customer to agree on the *character repertoire* that will be used. For our running example, we continue to use the characters defined by ISO Latin 9.

Step 2 : Try to find what which character encodings the customer is using. It's quite possible that the customer doesn't know, but that's not too big a problem, since the proposed solution covers most encodings in use. Our running example uses ISO Latin 9.

Step 3 : Convert the source file to big-endian UTF-32 using program `recode`[17] available under the GNU GPL from `http://recode.progiciels-bpi.ca/` and included in Linux and other fine operating systems.

> "This recoding library converts files between various coded character sets and surface encodings. When this cannot be achieved exactly, it may get rid of the offending characters or fall back on approximations. The library recognises or produces more than 300 different character sets and is able to convert files between almost any pair."

Step 4 : The choice of big-endian UTF-32 makes it a little easier to set up the emulator, and guarantees that the technique works with all Unicode characters. In many cases[18] it would be possible to use UTF-16, but let's not get involved in whether its big-end or little-end. Figure 15 shows the decimal values that we seek to represent using blocks of four 8-bit characters.

Step 5 : Each character in the source file will now appear in the input stream as a block of four 8-bit characters, each of which we represent using `\ddd` where `ddd` is a 3 digit decimal escape. See chapter 6 on page 7. The four 8-bit characters emulate the 32 bit integer value of the Unicode code position:

$$\text{``}\backslash\text{aaa}\backslash\text{bbb}\backslash\text{ccc}\backslash\text{ddd''} \equiv (\text{bbb}\times 256+\text{ccc})\times 256+\text{ddd}$$

[17] Merci François!

[18] Including the case of our worked example.

since in all cases aaa= 0.

Step 6 : Each of the UTF-32 encoded characters we seek in the input stream has been pre-defined in the lexer specification `pi.UTF-32.lex` where the variable names provide a mnemonic for the UTF-32 character. Ok, using hexadecimal is not a very good mnemonic, and it would have been a lot clearer to write “ch_1”, “ch_2” and “ch_euro”.

Lets look at the changes to the lexer specification. The modified lexer specification is in file `pi.UTF-32.lex` which we now review, section by section. We will discuss only those parts of the file which have changed.

11.3.1 File `pi.UTF-32.lex` — user declarations

```
465 (* pi.UTF-32.lex *)
466 (* This file is to be encoded in emacs iso-latin-1. *)
467 val DEBUG = true;
```

We use emacs to create the file `pi.UTF-32.lex`, and the good news is that emacs now offers the possibility of storing the buffer in a file encoded in a variety of ways. Since the file is to be read by ML-Lex, it must be in ISO Latin 1 and line 466 reminds the programmer of this. To specify the required encoding, use the emacs command `C-x C-m f latin-1 RET`.

On line 467 you should set the variable `DEBUG` to `true` to get a detailed trace of the modified lexer. This trace is produced by a extra function call placed in each ML-Lex rule.

```
468 val col = ref 0;
469 val eolpos = ref 0;
```

On lines 468 and 469 variables `col` and `eolpos` now provide the octet position within the UTF-32 encoded file, which is not what the customer expects.

```
470 val eof = fn fileName => T.EOF (!lin,(!col) div 4);
```

Luckily, the conversion is simple in our example: divide by 4, as seen on line 470. The conversion is more complex if the source file contains combining characters.

```
471 val rec chrLat9Help : int -> char = fn
472       164 => raise ChrLat9Error
473     |  166 => raise ChrLat9Error
          ⋮
474     |  352 => chr(166)   (* Latin capital S with caron *)
475     | 8364 => chr(164)   (* Euro sign *)
476     |     x => if x<0 orelse x>255 then raise ChrLat9Error
477                                   else chr x;
          ⋮
478 val chrsLat9 : string -> string = fn
479    L => implode (chrsLat9Help (explode L));
```

The trace function in `pi.UTF-32.lex` requires a helper function `chrsLat9` shown on line 478 which converts a small subset of UTF-32 to ISO Latin 9. Given a list of integers which when taken 4 by 4 represent the Unicode positions for characters appearing in ISO Latin 9, return the character string. For example [0,0,32,172] represents a string containing the Euro character. This requires care to avoid getting tangled in emac's own character encoding. Naïvely, one would want to convert [0,0,32,172] to the character #"€" which could then be imploded into a string. However writing #"€" in a program, and then saving it as ISO Latin 1 as required by ML-Lex will provoke emacs into issuing an error message. The character € is not a part of ISO Latin 1 and cannot be saved. The proposed alternative is UTF-8 which is not acceptable to ML-Lex.

The solution is to express the character #"€" as chr(164) as shown on line 475.

```
480 val lexDisplay : string*int*int*int*string -> unit = fn
481 (tag,pos,line,col,L) => if DEBUG
482       then print ("lex:"^tag^":\t"^Int.toString pos^" "
483                    ^Int.toString line^" "
484                    ^Int.toString col^" "^chrsLat9 L^" "
485                    ^ppIntList (strToInt L)^"\n")
486       else ();
```

Function lexDisplay on line 480 provides a simple debugging display of the lexer's activity. Line 487 shows a typical line of output:

```
487 lex:PI alpha 1: 74 2 0 new [0,0,0,110,0,0,0,101,0,0,0,119]
```

The rule "<PI>alpha+" has detected a keyword beginning at the 74th octet of the file, i.e. line 2 octet 0. The keyword is "new" and is represented by the list of integers [0,...,119].

11.3.2 File pi.UTF-32.lex — definitions

```
488 ch00 = "\000\000\000\000";
          ⋮
489 cha4 = "\000\000\032\172";
490 cha5 = "\000\000\000\165";
491 cha6 = "\000\000\001\096";
492 cha7 = "\000\000\000\167";
493 cha8 = "\000\000\001\097";
494 chb4 = "\000\000\001\125";
495 chb8 = "\000\000\001\126";
          ⋮
496 chbc = "\000\000\001\082";
497 chbd = "\000\000\001\083";
498 chbe = "\000\000\001\120";
          ⋮
499 chff = "\000\000\000\255";
```

First list your character repertoire, i.e. all the characters that you propose to recognize, and assign them names. The names we have chosen are poor examples. They repeat the hexadecimal value of the ISO Latin 9 encoding, and have little mnemonic value. You should be able to do something much better. For each name, write out the UTF-32 big-end encoding as a string. The easiest way to do this is with the \ddd notation. The eight names which distinguish ISO Latin 9 from ISO Latin 1 begin at line 489. Wouldn't it have been better to write ch_euro?

```
%%
<INITIAL>{ws}* => (lin:=1; eolpos:=0; YYBEGIN PI;
                   lexDisplay ("INITIAL ws",yypos,
                               !lin,!col,yytext);
                   continue ());
<PI>{ws}*     => (continue ());
<PI>{eol}     => (lin:=(!lin)+1;eolpos:=yypos+size yytext;
                  continue ());
<PI>{alpha}+ => (let val yyLat9 = chrsLat9 yytext
                 in case find yyLat9 of
                     SOME v => (col:=yypos-(!eolpos);
                                v(!lin,(!col) div 4))
                   | _ => (col:=yypos-(!eolpos);
                           T.IDE(yyLat9,!lin,(!col) div 4))
                 end);
<PI>{percent}=> (YYBEGIN COMMENT; continue ());
<PI>{equals} => (col:=yypos-(!eolpos);
                 T.EQUALS(!lin,(!col) div 4));
<PI>{lpar}    => (col:=yypos-(!eolpos);
                  T.LPAR(!lin,(!col) div 4));
<PI>{rpar}    => (col:=yypos-(!eolpos);
                  T.RPAR(!lin,(!col) div 4));
<PI>{exclam} => (col:=yypos-(!eolpos);
                 T.OUTPUT(!lin,(!col) div 4));
<PI>{quest}   => (col:=yypos-(!eolpos);
                  T.INPUT(!lin,(!col) div 4));
<PI>{vbar}{vbar} => (col:=yypos-(!eolpos);
                     T.DVBAR(!lin,(!col) div 4));
<PI>{any}     => (col:=yypos-(!eolpos);
                  badCh (fileName,chrsLat9 yytext,
                         !lin,(!col) div 4);
                  T.ILLCH(!lin,(!col) div 4));
<COMMENT>{eol} => (lin:=(!lin)+1;eolpos:=yypos+size yytext;
                   YYBEGIN PI; continue ());
<COMMENT>{any} => (continue ());
```

Figure 16: File pi.UTF-32.lex. The rules, modified for UTF-32 encodings.

```
535 uc = {ch41}|{ch42}|{ch43}|...
536 lc = {ch61}|{ch62}|{ch63}|{ch64}|...
537 dg = {ch30}|{ch31}|{ch32}|{ch33}|{ch34}|...
```

We now define the upper case letters, line 535, the lower case letters, line 536 and the digits, line 537.

```
538 alpha          = {uc}|{lc};
539 digit          = {dg};
540 space          = {ch20};
541 tab            = {ch09};
542 ws             = {space}|{tab};
543 lf             = {ch0a};
544 cr             = {ch0d};
545 eol            = {cr}{lf}|{lf}|{cr};
546 percent        = {ch25};
547 equals         = {ch3d};
548 lpar           = {ch28};
549 rpar           = {ch29};
550 exclam         = {ch21};
551 quest          = {ch3f};
552 vbar           = {ch7c};
553 anyoctet       = \000|[^\000];
554 any            = {anyoctet}{anyoctet}{anyoctet}{anyoctet};
```

We can now write out the names that we would like to use in the rules. Note in line 528, that we do not use a period "." to represent any character except a new-line. This is replaced by `{any}` which will match any character, new line or otherwise.

11.3.3 File `pi.UTF-32.lex` — rules

The modified rules needed for analyzing a UTF-32 encoded file are shown in figure 16, which should be compared with the original rules shown in figure 10 on page 40. To lighten up the code a little, all the calls to the tracing function `lexDisplay` have been removed except for the first on line 502.

The lookup for keywords provided by function `find` assumes that the argument is an ML string, i.e. a sequence of single octet characters. On line 508 we form such a string "`yyLat9`" from the UTF-32 encoded sequence `yytext`. Line 513 handles the case in which we find a non-keyword name. The rule passes the single octet ISO Latin 9 characters in `yytext` to `pi.yacc`. This maintains the interface to the parser and we do not need to modify `pi.yacc` to handle exotic encodings. This rule also passes the column position *as seen by the user*, `(!col) div 4`, to the parser.

On line 554 note the use of `{any}` in place of the period seen in line 343 on page 40.

11.3.4 Sample session

We are now ready for a demonstration. To compile our modified program, we first create a new `pi.UTF-32.cm` from `pi.cm` by updating line 232 on page 36 to read

```
    pi.UTF-32.lex        (* UTF-32 lexer rules. *)
```

Now turn on SML and compile the modified project:

```
$ sml
Standard ML of New Jersey v110.55
- CM.make "pi.UTF-32.cm" ;
...
[New bindings added.]
val it = true : bool
-
```

We need sample data in big-endian UTF-32. The easiest way to get this is with the command `recode latin-9..UCS-4BE < good.pi > good.pi.UTF-32`. Note that `recode` requires that you write "`UCS-4` instead of `UTF-32`. I have added a Euro symbol to the comment to test our handling of non ISO Latin 1 characters. We also manufacture some bad data: `recode latin-9..UCS-4BE < bad.pi > bad.pi.UTF-32`. Now let's run the "UTF-32" project on the "UTF-32" files. The output begins with a trace of the lexer as it analyses the comment:

```
- Pi.compile "good.pi.UTF-32" ;
lex:INITIAL ws: 2 1 0   []
lex:PI %:       2 1 0 % [0,0,0,37]
lex:COMMENT any:        6 1 0 % [0,0,0,37]
lex:COMMENT any:        10 1 0 % [0,0,0,37]
lex:COMMENT any:        14 1 0   [0,0,0,32]
lex:COMMENT any:        18 1 0 g [0,0,0,103]
lex:COMMENT any:        22 1 0 o [0,0,0,111]
lex:COMMENT any:        26 1 0 o [0,0,0,111]
lex:COMMENT any:        30 1 0 d [0,0,0,100]
lex:COMMENT any:        34 1 0 . [0,0,0,46]
lex:COMMENT any:        38 1 0 p [0,0,0,112]
lex:COMMENT any:        42 1 0 i [0,0,0,105]
lex:COMMENT any:        46 1 0   [0,0,0,32]
lex:COMMENT any:        50 1 0 €  [0,0,32,172]
lex:COMMENT any:        54 1 0   [0,0,0,32]
lex:COMMENT any:        58 1 0 % [0,0,0,37]
lex:COMMENT any:        62 1 0 % [0,0,0,37]
lex:COMMENT any:        66 1 0 % [0,0,0,37]
lex:COMMENT eol:        70 2 0
 [0,0,0,10]
```

The trace continues with the second line. Note that the lexer status returns to `PI`.

```
lex:PI alpha 1: 74 2 0 new [0,0,0,110,0,0,0,101,0,0,0,119]
lex:PI ws:      86 2 0   [0,0,0,32]
lex:PI alpha 2: 90 2 16 a [0,0,0,97]
lex:PI ws:      94 2 16   [0,0,0,32]
lex:PI (:       98 2 24 ( [0,0,0,40]
lex:PI alpha 2: 102 2 28 a [0,0,0,97]
lex:PI !:       106 2 32 ! [0,0,0,33]
lex:PI alpha 2: 110 2 36 x [0,0,0,120]
lex:PI ws:      114 2 36   [0,0,0,32]
lex:PI ||:      118 2 44 || [0,0,0,124,0,0,0,124]
lex:PI ws:      126 2 44   [0,0,0,32]
lex:PI alpha 2: 130 2 56 a [0,0,0,97]
lex:PI ?:       134 2 60 ? [0,0,0,63]
lex:PI alpha 2: 138 2 64 x [0,0,0,120]
lex:PI =:       142 2 68 = [0,0,0,61]
lex:PI (:       146 2 72 ( [0,0,0,40]
lex:PI ):       150 2 76 ) [0,0,0,41]
lex:PI ):       154 2 80 ) [0,0,0,41]
lex:PI eol:     158 3 80
 [0,0,0,10]
```

The result returned by the parser is:

```
val it =
  Pi
    [New
       (Name ("a","good.pi.UTF-32",2,4),
        Parallel
          [Output
             (Name ("a","good.pi.UTF-32",2,7),
              V (Name ("x","good.pi.UTF-32",2,9))),
           Input
             (Name ("a","good.pi.UTF-32",2,14),
                    A (Pat (Name #),Parallel []))])]
  : ?.DataTypes.Pi
-
```

The result, omitting the trace of the comment, produced by the bad data is:

```
- Pi.compile "bad.pi.UTF-32" ;
lex:PI alpha 1: 62 2 0 new [0,0,0,110,0,0,0,101,0,0,0,119]
lex:PI ws:      74 2 0    [0,0,0,32]
lex:PI alpha 1: 78 2 16 new [0,0,0,110,0,0,0,101,0,0,0,119]
lex:PI ws:      90 2 16    [0,0,0,32]
lex:PI alpha 1: 94 2 32 new [0,0,0,110,0,0,0,101,0,0,0,119]
lex:PI ws:      106 2 32    [0,0,0,32]
lex:PI any:     110 2 48 . [0,0,0,46]
bad.pi.UTF-32[2.12] Invalid character "."
lex:PI any:     114 2 52 . [0,0,0,46]
bad.pi.UTF-32[2.13] Invalid character "."
bad.pi.UTF-32[2:12] syntax error: deleting  NEW NEW ILLCH
bad.pi.UTF-32[2:2] syntax error found at ILLCH

uncaught exception PiError
  raised at: compiler.sml:45.49-45.56
-
```

This output should be compared with the earlier output at line 440 on page 44.

We hope that you have been convinced that it is relatively easy to write lexers for UTF-32 encoded character sets. It is also possible to modify ML-Lex to do the addditional character manipulation. Such modification requires a lot of engineering which I don't propose to discusss here.

12 Hints

This chapter describes techniques of interest to advanced users, and may be omitted at a first reading.

12.1 Multiple start symbols

With the little green book in their hands,
They set out to conquer all lands.
But Java and C,
And much bigotry,
Still stifle those noblest of plans.

To have multiple start symbols, define a dummy token for each start symbol. Then define a start symbol which derives the multiple start symbols with dummy tokens placed in front of them. When you start the parser you must place a dummy token on the front of the lexer stream to select a start symbol from which to begin parsing.

Assuming that you have followed the naming conventions used before, create the lexer using the `makeLexer` function in the structure *My*`Parser`[19]. Then, place the dummy token on the front of the lexer:

```
val dummyLexer =
    MyParser.Stream.cons
        (MyLrVals.Tokens.dummy token name
                 (dummy lineno,dummy lineno),
        lexer)
```

You have to pass a `Tokens` structure to the lexer. This `Tokens` structure contains functions which construct tokens from values and line numbers. So to create your dummy token just apply the appropriate token constructor function from this `Tokens` structure to a value (if there is one) and the line numbers. This is exactly what you do in the lexer to construct tokens.

Then you must place the dummy token on the front of your lex stream. The structure *My*`Parser` contains a structure `Stream` which implements lazy streams. So you just cons the dummy token on to stream returned by `makeLexer`.

12.2 Functorizing things further

You may wish to functorize things even further. Two possibilities are turning the lexer and parser structures into closed functors, that is, functors which do not refer to types or values defined outside their body or outside their parameter structures (except for pervasive types and values), and creating a functor which encapsulates the code necessary to invoke the parser.

Use the `%header` declaration in ML-Lex and the `%header` declaration in ML-Yacc to create closed functors. See chapters 7.2.2 and 9.4.6 for complete descriptions of these declarations. If you do this, you should also parameterise these structures by the types of line numbers. The type will be an abstract type, so you will also need to define all the valid operations on the type. The signature `INTERFACE`, defined below, shows one possible signature for a structure defining the line number type and associated operations.

If you wish to encapsulate the code necessary to invoke the parser, your functor generally will have form:

```
functor Encapsulate(
     structure Parser : PARSER
     structure Interface : INTERFACE
         sharing type Parser.arg = Interface.arg
         sharing type Parser.pos = Interface.pos
         sharing type Parser.result = ...
     structure Tokens : {parser name}_TOKENS
         sharing type Tokens.token = Parser.Token.token
         sharing type Tokens.svalue = Parser.svalue) =
  struct
        ...
  end
```

[19]Do you remember that it was created on line 393?

The signature `INTERFACE`, defined below, is a possible signature for a structure defining the types of line numbers and arguments (types `pos` and `arg`, respectively) along with operations for them. You need this structure because these types will be abstract types inside the body of your functor.

```
signature INTERFACE =
sig
   type pos
   val line : pos ref
   val reset : unit -> unit
   val next : unit -> unit
   val error : string * pos * pos -> unit

   type arg
   val nothing : arg
end
```

The directory `example/fol` contains a sample parser in which the code for tying together the lexer and parser has been encapsulated in a functor.

13 Acknowledgements

Nick Rothwell wrote an SLR table generator in 1988 which inspired the initial work on an ML parser generator. Bruce Duba and David MacQueen made useful suggestions about the design of the error-correcting parser. Thanks go to all the users at Carnegie Mellon who beta-tested this version. Their comments and questions led to the creation of this manual and helped improve it.

14 Bugs

1. There is a slight difference in syntax between ML-Lex and ML-Yacc. In ML-Lex, semantic actions must be followed by a semicolon but in ML-Yacc semantic actions cannot be followed by a semicolon. The syntax should be the same. ML-Lex also produces structures with two different signatures, but it should produce structures with just one signature. This would simplify some things.

2. The position of the first character in the file is wrongly reported as 2, unless the `%posarg` feature is used, chapter 7.2.6. To preserve compatibility, this bug has not been fixed.

   You can fix it yourself if you want to. In our running example, in the file `pi.lex.sml`, in function `makeLexer` change[20] `val yygone0=1` to `val yygone0=~1`.

15 Questions and answers

This description of ML-Lex leaves some questions unanswered:

15.1 Why does "\␣" mean "a single space"?

By what rule does the "\␣" in the named expression `ws` in line 212 mean "a single space"?

[20] It would of course be better to make a permanent change in `src/ml-lex/lexgen.sml`.

15.2 Why is the basic payload of a token two integers?

By what declaration is the basic payload of a token set to _*two*_ integers?

Answer Hans Leiss, 2009-03-20

It's _two_ (but not necessarily integers) by the signature declaration in `lib/base.sig`:

```
signature TOKEN =
  sig
    structure LrTable : LR_TABLE
    datatype ('a,'b) token = TOKEN of LrTable.term * ('a * 'b * 'b)
    val sameToken : ('a,'b) token * ('a,'b) token -> bool
  end
```

The two positions mark token left end and token right end (relative to the initial position in a file), and are used to give `leftPos` and `rightPos` to each element on the parse stack by the semantic actions in `lib/parsern.sml.` The types are

```
type ('a,'b) elem = (state * ('a * 'b * 'b))
type ('a,'b) stack = ('a,'b) elem list

saction : int * '_c * ('_b,'_c) stack * 'a ->
          nonterm * ('_b * '_c * '_c) * ('_b,'_c) stack,
```

In the end, the two positions are used to limit the phrase where an error occured.

15.3 Why is there a question mark on line 438?

Where does the question mark "?" on line 438 on page 44 come from?

Answer Hans Leiss, 2009-03-20

From a missing `"structure PiLrVals"` or `"structure PiParser"` in the `Library exports` of `pi.cm`, 11.2.1 on page 36.

The "?" on line 459 on page 45 and on line614 on page 52 also come from the missing export.

15.4 How can a string v be used as a function in v(!lin,!col)?

Program lines 331-335 on page 40:

```
  (case find yytext of
        SOME v => (col:=yypos-(!eolpos);
                   v(!lin,!col))
      | _      => (col:=yypos-(!eolpos);
                   T.IDE(yytext,!lin,!col)));
```

Here `v` is the matched string `yytext`, not a token constructor. So how can it be applied to `(!lin,!col)` ???

Answer The function `find` has type

```
string -> (int * int -> (svalue,int) token) option
```

The value returned by `find` is itself a function which maps `(!lin,!col)` to a token. See line 285 on page 39.

A ISO Latin 9

000	*0*	00$_x$	NUL	020	*16*	10$_x$	DLE
001	*1*	01$_x$	STX	021	*17*	11$_x$	DC1
002	*2*	02$_x$	SOT	022	*18*	12$_x$	DC2
003	*3*	03$_x$	ETX	023	*19*	13$_x$	DC3
004	*4*	04$_x$	EOT	024	*20*	14$_x$	DC4
005	*5*	05$_x$	ENQ	025	*21*	15$_x$	NAK
006	*6*	06$_x$	ACK	026	*22*	16$_x$	SYN
007	*7*	07$_x$	BEL	027	*23*	17$_x$	ETB
010	*8*	08$_x$	BS	030	*24*	18$_x$	CAN
011	*9*	09$_x$	HT	031	*25*	19$_x$	EM
012	*10*	0a$_x$	LF	032	*26*	1a$_x$	SUB
013	*11*	0b$_x$	VT	033	*27*	1b$_x$	ESC
014	*12*	0c$_x$	FF	034	*28*	1c$_x$	FS
015	*13*	0d$_x$	CR	035	*29*	1d$_x$	GS
016	*14*	0e$_x$	SO	036	*30*	1e$_x$	RS
017	*15*	0f$_x$	SI	037	*31*	1f$_x$	US

040	*32*	20$_x$	SP	060	*48*	30$_x$	0
041	*33*	21$_x$	!	061	*49*	31$_x$	1
042	*34*	22$_x$	"	062	*50*	32$_x$	2
043	*35*	23$_x$	#	063	*51*	33$_x$	3
044	*36*	24$_x$	$	064	*52*	34$_x$	4
045	*37*	25$_x$	%	065	*53*	35$_x$	5
046	*38*	26$_x$	&	066	*54*	36$_x$	6
047	*39*	27$_x$	'	067	*55*	37$_x$	7
050	*40*	28$_x$	(	070	*56*	38$_x$	8
051	*41*	29$_x$	)	071	*57*	39$_x$	9
052	*42*	2a$_x$	*	072	*58*	3a$_x$	:
053	*43*	2b$_x$	+	073	*59*	3b$_x$	;
054	*44*	2c$_x$	,	074	*60*	3c$_x$	<
055	*45*	2d$_x$	-	075	*61*	3d$_x$	=
056	*46*	2e$_x$	.	076	*62*	3e$_x$	>
057	*47*	2f$_x$	/	077	*63*	3f$_x$	?

continued on next page

Here is the character set specified by International Standard 8859-15 [ISO99], known as ISO Latin 9 which is a variant of ISO Latin 1, which in turn is subset of the Unicode [TUC03] characters. Each entry gives the octal code, the decimal code used by ML-Lex in the `\ddd` notation, i.e. the character position, the value expressed in hexadecimal, and an approximation for the glyph.

Note that the Euro symbol in position *164* has Unicode value U+20AC.

continuing from previous page

100	*64*	40_x	@	120	*80*	50_x	P
101	*65*	41_x	A	121	*81*	51_x	Q
102	*66*	42_x	B	122	*82*	52_x	R
103	*67*	43_x	C	123	*83*	53_x	S
104	*68*	44_x	D	124	*84*	54_x	T
105	*69*	45_x	E	125	*85*	55_x	U
106	*70*	46_x	F	126	*86*	56_x	V
107	*71*	47_x	G	127	*87*	57_x	W
110	*72*	48_x	H	130	*88*	58_x	X
111	*73*	49_x	I	131	*89*	59_x	Y
112	*74*	$4a_x$	J	132	*90*	$5a_x$	Z
113	*75*	$4b_x$	K	133	*91*	$5b_x$	[
114	*76*	$4c_x$	L	134	*92*	$5c_x$	\
115	*77*	$4d_x$	M	135	*93*	$5d_x$	]
116	*78*	$4e_x$	N	136	*94*	$5e_x$	ˆ
117	*79*	$4f_x$	O	137	*95*	$5f_x$	_

140	*96*	60_x	‘	160	*112*	70_x	p
141	*97*	61_x	a	161	*113*	71_x	q
142	*98*	62_x	b	162	*114*	72_x	r
143	*99*	63_x	c	163	*115*	73_x	s
144	*100*	64_x	d	164	*116*	74_x	t
145	*101*	65_x	e	165	*117*	75_x	u
146	*102*	66_x	f	166	*118*	76_x	v
147	*103*	67_x	g	167	*119*	77_x	w
150	*104*	68_x	h	170	*120*	78_x	x
151	*105*	69_x	i	171	*121*	79_x	y
152	*106*	$6a_x$	j	172	*122*	$7a_x$	z
153	*107*	$6b_x$	k	173	*123*	$7b_x$	{
154	*108*	$6c_x$	l	174	*124*	$7c_x$	|
155	*109*	$6d_x$	m	175	*125*	$7d_x$	}
156	*110*	$6e_x$	n	176	*126*	$7e_x$	~
157	*111*	$6f_x$	o	177	*127*	$7f_x$	$\boxed{\text{DEL}}$

continued on next page

continuing from previous page

200	*128*	80_x	CRTL	220	*144*	90_x	CRTL
201	*129*	81_x	CRTL	221	*145*	91_x	CRTL
202	*130*	82_x	CRTL	222	*146*	92_x	CRTL
203	*131*	83_x	CRTL	223	*147*	93_x	CRTL
204	*132*	84_x	CRTL	224	*148*	94_x	CRTL
205	*133*	85_x	CRTL	225	*149*	95_x	CRTL
206	*134*	86_x	CRTL	226	*150*	96_x	CRTL
207	*135*	87_x	CRTL	227	*151*	97_x	CRTL
210	*136*	88_x	CRTL	230	*152*	98_x	CRTL
211	*137*	89_x	CRTL	231	*153*	99_x	CRTL
212	*138*	$8a_x$	CRTL	232	*154*	$9a_x$	CRTL
213	*139*	$8b_x$	CRTL	233	*155*	$9b_x$	CRTL
214	*140*	$8c_x$	CRTL	234	*156*	$9c_x$	CRTL
215	*141*	$8d_x$	CRTL	235	*157*	$9d_x$	CRTL
216	*142*	$8e_x$	CRTL	236	*158*	$9e_x$	CRTL
217	*143*	$8f_x$	CRTL	237	*159*	$9f_x$	CRTL

240	*160*	$a0_x$	NBSP	260	*176*	$b0_x$	°
241	*161*	$a1_x$	¡	261	*177*	$b1_x$	$\pm$
242	*162*	$a2_x$	¢	262	*178*	$b2_x$	^{2}
243	*163*	$a3_x$	£	263	*179*	$b3_x$	^{3}
244	*164*	$a4_x$	€	264	*180*	$b4_x$	Ž
245	*165*	$a5_x$	¥	265	*181*	$b5_x$	μ
246	*166*	$a6_x$	Š	266	*182*	$b6_x$	¶
247	*167*	$a7_x$	§	267	*183*	$b7_x$	·
250	*168*	$a8_x$	š	270	*184*	$b8_x$	ž
251	*169*	$a9_x$	©	271	*185*	$b9_x$	^{1}
252	*170*	aa_x	ª	272	*186*	ba_x	º
253	*171*	ab_x	$\ll$	273	*187*	bb_x	$\gg$
254	*172*	ac_x	$\neg$	274	*188*	bc_x	Œ
255	*173*	ad_x	SHY	275	*189*	bd_x	œ
256	*174*	ae_x	®	276	*190*	be_x	Ÿ
257	*175*	af_x	¯	277	*191*	bf_x	¿

continued on next page

continuing from previous page							
300	*192*	c0$_x$	À	320	*208*	d0$_x$	Ð
301	*193*	c1$_x$	Á	321	*209*	d1$_x$	Ñ
302	*194*	c2$_x$	Â	322	*210*	d2$_x$	Ò
303	*195*	c3$_x$	Ã	323	*211*	d3$_x$	Ó
304	*196*	c4$_x$	Ä	324	*212*	d4$_x$	Ô
305	*197*	c5$_x$	Å	325	*213*	d5$_x$	Õ
306	*198*	c6$_x$	Æ	326	*214*	d6$_x$	Ö
307	*199*	c7$_x$	Ç	327	*215*	d7$_x$	×
310	*200*	c8$_x$	È	330	*216*	d8$_x$	Ø
311	*201*	c9$_x$	É	331	*217*	d9$_x$	Ù
312	*202*	ca$_x$	Ê	332	*218*	da$_x$	Ú
313	*203*	cb$_x$	Ë	333	*219*	db$_x$	Û
314	*204*	cc$_x$	Ì	334	*220*	dc$_x$	Ü
315	*205*	cd$_x$	Í	335	*221*	dd$_x$	Ý
316	*206*	ce$_x$	Î	336	*222*	de$_x$	Þ
317	*207*	cf$_x$	Ï	337	*223*	df$_x$	ß

340	*224*	e0$_x$	à	360	*240*	f0$_x$	ð
341	*225*	e1$_x$	á	361	*241*	f1$_x$	ñ
342	*226*	e2$_x$	â	362	*242*	f2$_x$	ò
343	*227*	e3$_x$	ã	363	*243*	f3$_x$	ó
344	*228*	e4$_x$	ä	364	*244*	f4$_x$	ô
345	*229*	e5$_x$	å	365	*245*	f5$_x$	õ
346	*230*	e6$_x$	æ	366	*246*	f6$_x$	ö
347	*231*	e7$_x$	ç	367	*247*	f7$_x$	÷
350	*232*	e8$_x$	è	370	*248*	f8$_x$	ø
351	*233*	e9$_x$	é	371	*249*	f9$_x$	ù
352	*234*	ea$_x$	ê	372	*250*	fa$_x$	ú
353	*235*	eb$_x$	ë	373	*251*	fb$_x$	û
354	*236*	ec$_x$	ì	374	*252*	fc$_x$	ü
355	*237*	ed$_x$	í	375	*253*	fd$_x$	ý
356	*238*	ee$_x$	î	376	*254*	fe$_x$	þ
357	*239*	ef$_x$	ï	377	*255*	ff$_x$	ÿ

B ML-Lex and ML-Yacc internals

B.1 Summary of signatures and structures

This chapter introduces the internal structure of ML-Lex and ML-Yacc and may be omitted at a first reading.

The following outline summarises the ML signatures and structures used to build a parser. First, the signatures available in file `base.sig` which is part of the ML-Yacc library `ml-yacc-lib.cm`.

```
661 signature STREAM = ...    (* Lazy stream *)
662 signature LR_TABLE = ... (* LR table *)
663 signature TOKEN = ...    (* Internal structure of token *)
664 signature LR_PARSER = ...(* Polymorphic LR parser *)
665 signature PARSER_DATA = ...(* ParserData structure * )
```

On line 665, `PARSER_DATA` is the signature of the `ParserData` structure in *My*`LrValsFun` produced by ML-Yacc.

Next, a structure in file `join.sml` which is part of the ML-Yacc library `ml-yacc-lib.cm`.

```
structure LrParser : LR_PARSER
```

The following signatures are written into file `my.yacc.sig` by ML-Yacc:

```
signature My_TOKENS =
sig
  structure Token : TOKEN
  type svalue
  val PLUS : 'pos * 'pos -> (svalue,'pos) Token.token
  val INTLIT : int * 'pos * 'pos
              -> (svalue,'pos) Token.token
  ...
end

signature My_LRVALS =
sig
  structure Tokens : My_TOKENS
  structure ParserData : PARSER_DATA
  sharing ParserData.Token = Tokens.Token
  sharing type ParserData.svalue = Tokens.svalue
end
```

The following functor is written into file `my.lex.sml` by ML-Lex:

```
functor MyLexFun(structure Tokens : My_TOKENS) =
struct
  ...
end
```

The following functor and structure are written into file `my.yacc.sml` by ML-Yacc:

```
functor MyLrValsFun(structure Token : TOKENS) =
struct
  structure ParserData =
  struct
    structure Token = Token

    (* Code from header section of my.yacc *)

    structure Header = ...
    type svalue = ...
    type result = ...
    type pos = ...
    structure Actions = ...
    structure EC = ...
    val table = ...
  end

  structure Tokens : My_TOKENS =
  struct
    structure Token = ParserData.Token
    type svalue = ...
    fun PLUS(p1,p2) = ...
    fun INTLIT(i,p1,p2) = ...
  end

end
```

You then glue these component structures together to create the operational structures *My*`LrVals` *My*`Lex` and *My*`Parser` as shown in chapter 11.2.5.

B.1.1 Parser structure signatures

The final structure created will have the signature `PARSER`:

```
signature PARSER =
sig
  structure Token : TOKEN
  structure Stream : STREAM
  exception ParseError

  type pos    (* pos is the type of line numbers *)
  type result (* Value returned by the parser *)
  type arg    (* Type of the user-supplied argument  *)
  type svalue (* The types of semantic values *)

  val makeLexer : (int -> string) ->
          (svalue,pos) Token.token Stream.stream
  val parse :
      int * ((svalue,pos) Token.token Stream.stream)
          * (string * pos * pos -> unit) * arg ->
      result * (svalue,pos) Token.token Stream.stream
  val sameToken :
      (svalue,pos) Token.token * (svalue,pos) Token.token
      -> bool
end
```

or the signature ARG_PARSER if you used the ML-Lex command %arg to create the lexer. This signature differs from ARG in that it has an additional type lexarg and a different type for makeLexer:

```
type lexarg
val makeLexer : (int -> string) -> lexarg
                -> (svalue,pos) token stream
```

The signature STREAM which provides lazy streams is:

```
signature STREAM =
sig
  type 'a stream
  val streamify : (unit -> 'a) -> 'a stream
  val cons : 'a * 'a stream -> 'a stream
  val get : 'a stream -> 'a * 'a stream
end
```

B.2 Using the parser structure

This chapter describes the internal operation of ML-Lex and ML-Yacc and may be omitted at a first reading.

The parser structure converts the lexing function produced by ML-Lex into a function which creates a lazy stream of tokens.

The function makeLexer takes the same values as the corresponding makeLexer created by ML-Lex, but returns a stream of tokens instead of a function which yields tokens.

The function parse takes the token stream and some other arguments that are described below and parses the token stream. It returns a pair composed of the value associated with the start symbol and the rest of the token stream. The rest of the token stream includes the end-of-parse symbol which caused the reduction of some rule to the start symbol. The function parse raises the exception ParseError if a syntax error occurs which it cannot fix.

In Glasgow a programmer who
Slept whíle others wére eager to
Get on with the work
Awoke with a jerk
"You can do that in Haskell too!"

The lazy stream is implemented by the Stream structure. In this structure the function streamify converts a conventional implementation of a stream into a lazy stream. In a conventional implementation of a stream, a stream consists of a position in a list of values. Fetching a value from a stream returns the value associated with the position and updates the position to the next element in the list of values. The fetch is a side-effecting operation. In a lazy stream, a fetch returns a value and a new stream, without a side-effect which updates the position value. This means that a stream can be repeatedly re-evaluated without affecting the values that it returns. If f is the function that is passed to streamify, f is called only as many times as necessary to construct the portion of the list of values that is actually used.

The function parse also takes an integer giving the maximum amount of lookahead permitted for the error-correcting parse, a function to print error messages, and a value of type arg. The maximum amount of lookahead for interactive systems should be zero. In this case, no attempt is made to correct any syntax errors. For non-interactive systems, try 15. The function to print error messages takes a tuple of values consisting of the left position and right position of the terminal which caused the error and an error message. If the %arg declaration is not used, the value of type arg should be a value of type unit.

The function sameToken can be used to see if two tokens denote the same terminal, irregardless of any values that the tokens carry. It is useful if you have multiple end-of-parse symbols and must check which end-of-parse symbol has been left on the front of the token stream.

The types have the following meanings. The type arg is the type of the additional argument to the parser, which is specified by the %arg declaration in file my.yacc. The type lexarg is the optional argument to lexers, and is specified by the %arg declaration in file my.lex. The type pos is the type of line numbers, and is specified by the %pos declaration in file my.yacc and defined in the user declarations section of file my.lex. The type result is the type associated with the start symbol in file my.yacc.

C Signatures

This chapter contains material for advanced users, and may be omitted at a first reading.

This chapter contains signatures used by ML-Yacc for structures in the file base.sml, functors and structures that it generates, and for the signatures of lexer structures supplied by you.

C.1 Parsing structure signatures

STREAM is a signature for a lazy stream.

```
signature STREAM =
sig
  type 'a stream
  val streamify : (unit -> 'a) -> 'a stream
  val cons : 'a * 'a stream -> 'a stream
  val get : 'a stream -> 'a * 'a stream
end
```

LR_TABLE is a signature for an LR Table.

```
signature LR_TABLE =
sig
  datatype ('a,'b) pairlist
    = EMPTY
    | PAIR of 'a * 'b * ('a,'b) pairlist
  datatype state = STATE of int
  datatype term = T of int
  datatype nonterm = NT of int
  datatype action = SHIFT of state
                  | REDUCE of int
                  | ACCEPT
                  | ERROR
  type table

  val numStates : table -> int
```

```
  val numRules : table -> int
  val describeActions : table -> state ->
                          (term,action) pairlist * action
  val describeGoto : table -> state ->
                       (nonterm,state) pairlist
  val action : table -> state * term -> action
  val goto : table -> state * nonterm -> state
  val initialState : table -> state
  exception Goto of state * nonterm

  val mkLrTable :
      {actions : ((term,action) pairlist * action) array,
       gotos : (nonterm,state) pairlist array,
       numStates : int, numRules : int,
       initialState : state} -> table
end
```

TOKEN is a signature for the internal structure of a token.

```
signature TOKEN =
sig
  structure LrTable : LR_TABLE
  datatype ('a,'b) token = TOKEN of LrTable.term *
                                     ('a * 'b * 'b)
  val sameToken : ('a,'b) token * ('a,'b) token -> bool
end
```

LR_PARSER is a signature for a polymorphic LR parser.

```
signature LR_PARSER =
sig
  structure Stream: STREAM
  structure LrTable : LR_TABLE
  structure Token : TOKEN

  sharing LrTable = Token.LrTable

  exception ParseError

  val parse:
       {table : LrTable.table,
        lexer : ('b,'c) Token.token Stream.stream,
        arg: 'arg,
        saction : int *
                  'c *
                  (LrTable.state * ('b * 'c * 'c)) list *
                  'arg ->
                   LrTable.nonterm *
                   ('b * 'c * 'c) *
```

```
                  ((LrTable.state *('b * 'c * 'c)) list),
        void : 'b,
        ec: {is_keyword : LrTable.term -> bool,
             noShift : LrTable.term -> bool,
             preferred_subst:LrTable.term -> LrTable.term list,
             preferred_insert : LrTable.term -> bool,
             errtermvalue : LrTable.term -> 'b,
             showTerminal : LrTable.term -> string,
             terms: LrTable.term list,
             error : string * 'c * 'c -> unit
            },
        lookahead : int (* max amount of lookahead used in
                         * error correction *)
       } -> 'b * (('b,'c) Token.token Stream.stream)
end
```

C.2 Lexers

Lexers for use with ML-Yacc's output must match one of these signatures.
Signature LEXER:

```
signature LEXER =
sig
  structure UserDeclarations :
    sig
      type ('a,'b) token
      type pos
      type svalue
    end
  val makeLexer : (int -> string) -> unit ->
       (UserDeclarations.svalue, UserDeclarations.pos)
       UserDeclarations.token
end
```

In signature `ARG_LEXER` the `%arg` option of ML-Lex allows users to produce lexers which also take an argument before yielding a function from unit to a token.

```
signature ARG_LEXER =
sig
  structure UserDeclarations :
    sig
      type ('a,'b) token
      type pos
      type svalue
      type arg
    end
  val makeLexer :
      (int -> string) ->
      UserDeclarations.arg ->
```

```
      unit ->
       (UserDeclarations.svalue, UserDeclarations.pos)
       UserDeclarations.token
end
```

C.3 Signatures for the functor produced by ML-Yacc

The following signature is used in signatures generated by ML-Yacc. The signature `PARSER_DATA` is the signature of `ParserData` structures in the *My*`LrValsFun` functor produced by ML-Yacc. All such structures match this signature.

```
signature PARSER_DATA =
sig
  type pos       (* the type of line numbers *)
  type svalue    (* the type of semantic values *)
  type arg       (* the type of the user-supplied *)
 (* argument to the parser *)
  type result

  structure LrTable : LR_TABLE
  structure Token : TOKEN
  sharing Token.LrTable = LrTable

  structure Actions :
    sig
      val actions : int * pos *
       (LrTable.state * (svalue * pos * pos)) list * arg ->
       LrTable.nonterm * (svalue * pos * pos) *
     ((LrTable.state *(svalue * pos * pos)) list)
      val void : svalue
      val extract : svalue -> result
    end
```

Structure `EC` contains information used to improve error recovery in an error-correcting parser.

```
  structure EC :
    sig
      val is_keyword : LrTable.term -> bool
      val noShift : LrTable.term -> bool
      val preferred_subst: LrTable.term -> LrTable.term list
      val preferred_insert : LrTable.term -> bool
      val errtermvalue : LrTable.term -> svalue
      val showTerminal : LrTable.term -> string
      val terms: LrTable.term list
    end

  (* table is the LR table for the parser *)
```

```
  val table : LrTable.table
end
```

ML-Yacc generates signatures: *My*_TOKENS which is printed out in the .sig file created by parser generator, and *My*_LRVALS:

```
signature My_TOKENS =
sig
  type ('a,'b) token
  type svalue
  ...
end

signature My_LRVALS =
sig
  structure Tokens : My_TOKENS
  structure ParserData : PARSER_DATA
  sharing type ParserData.Token.token = Tokens.token
  sharing type ParserData.svalue = Tokens.svalue
end
```

C.4 User parser signatures

Parsers created by applying the Join functor will match the signature PARSER:

```
signature PARSER =
sig
  structure Token : TOKEN
  structure Stream : STREAM
  exception ParseError

  type pos    (* pos is the type of line numbers *)
  type result (* value returned by the parser *)
  type arg    (* type of the user-supplied argument  *)
  type svalue (* the types of semantic values *)

  val makeLexer : (int -> string) ->
   (svalue,pos) Token.token Stream.stream

  val parse :
      int * ((svalue,pos) Token.token Stream.stream) *
      (string * pos * pos -> unit) * arg ->
  result * (svalue,pos) Token.token Stream.stream
  val sameToken :
    (svalue,pos) Token.token * (svalue,pos) Token.token ->
bool
end
```

The parsers which are created by applying the JoinWithArg functor will match the signature ARG_PARSER:

```
signature ARG_PARSER =
sig
  structure Token : TOKEN
  structure Stream : STREAM
  exception ParseError

  type arg
  type lexarg
  type pos
  type result
  type svalue

  val makeLexer : (int -> string) -> lexarg ->
     (svalue,pos) Token.token Stream.stream
  val parse : int *
      ((svalue,pos) Token.token Stream.stream) *
      (string * pos * pos -> unit) *
      arg ->
       result * (svalue,pos) Token.token Stream.stream
  val sameToken :
      (svalue,pos) Token.token * (svalue,pos) Token.token ->
   bool
end
```

C.5 Sharing constraints

Let the name of the parser be denoted by `My`. If you have not created a lexer which takes an argument, and you have followed the directions given earlier for creating the parser, you will have the following structures with the following signatures:

These signatures are always present:

```
signature TOKEN
signature LR_TABLE
signature STREAM
signature LR_PARSER
signature PARSER_DATA
structure LrParser : LR_PARSER
```

These signatures are generated by ML-Yacc:

```
signature My_TOKENS
signature My_LRVALS
```

These structures created by you:

```
structure MyLrVals : My_LRVALS
structure Lex : LEXER
structure MyParser : PARSER
```

The following sharing constraints will exist:

```
sharing MyParser.Token = LrParser.Token =
          MyLrVals.ParserData.Token
sharing MyParser.Stream = LrParser.Stream

sharing type MyParser.arg = MyLrVals.ParserData.arg
sharing type MyParser.result = MyLrVals.ParserData.result
sharing type MyParser.pos = MyLrVals.ParserData.pos =
                Lex.UserDeclarations.pos
sharing type MyParser.svalue = MyLrVals.ParserData.svalue =
        MyLrVals.Tokens.svalue = Lex.UserDeclarations.svalue
sharing type MyParser.Token.token =
           MyLrVals.ParserData.Token.token =
           LrParser.Token.token =
           Lex.UserDeclarations.token

sharing MyLrVals.LrTable = LrParser.LrTable
```

If you used a lexer which takes an argument, then you will have:

```
structure Lex: ARG_LEXER
structure MyParser : PARSER
```

with the additional sharing constraint:

```
sharing type MyParser.lexarg = Lex.UserDeclarations.arg
```

References

[App98] Andrew W. Appel. *Modern compiler implementation in ML*. Cambridge University Press, first edition, January 1998.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, first edition, 1986.

[BF87] M. Burke and G. Fisher. A practical method for LR and LL syntactic error diagnosis and recovery. *ACM Transactions on Programming Languages and Systems*, 9(2):164–167, April 1987.

[Blu02] Matthias Blume. CM, the SML/NJ compilation and library manager. User manual, Lucent Technologies, Bell Labs, May 2002. For SML/NJ version 110.40 and later. <http://www.smlnj.org/doc/CM/index.html/>.

[ISO87] International Organization for Standardization, Geneva. *ISO 8859:1987, 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1*, first edition, February 1987.

[ISO99] International Organization for Standardization, Geneva. *ISO 8859:1999, 8-bit single-byte coded graphic character sets — Part 15: Latin alphabet No. 9*, first edition, 1999. Draft available at <http://www.evertype.com/standards/iso8859/8859-15-en.pdf>. See <www.cs.tut.fi/~jkorpela/latin9.html> for a comparison with ISO Latin 1.

[LMB95] John R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc*. O'Reilly & Associates, second edition, 1995.

[Pau96] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge, second edition, 1996.

[TUC03] The Unicode Consortium. *The Unicode Standard, Version 4.0*. Addison-Wesley, Boston, MA, 2003.
The online edition of the Unicode Standard, Version 4.1.0 is available at <http://www.unicode.org>.

[Ull98] Jeffrey D. Ullman. *Elements of ML Programming*. Prentice Hall, ML 97 edition, 1998.

A don on the banks of the Cam
Wrote a book — for the labouring man!
If you can hack code,
And your functors look good,
Isabelle offers her hand.

Index