卐 Compiler Short Revision Notes 卐

Sourabh Aggarwal

Compiled on February 6, 2019

Contents

1 Intro And ML Lex

$(a \odot b)|\epsilon$ represents the language {"", "ab"}. In writing regular expressions, we will sometimes omit the concatenation symbol or the epsilon, and we will assume that Kleene closure "binds tighter" than concatenation, and concatenation binds tighter than alternation; so that $ab|c$ means $(a \odot b)|c$, and $(a|)$ means $(a|\epsilon)$. Let us introduce some more abbreviations: [abed] means $(a|b|c|d)$, [b-g] means [bcdefg], [b-gM-Qkr] means [bcdefgMNOPQkr], M? means $(M|\epsilon)$, and M^+ means $(M \odot M^*)$.

```
if                                      (IF);
[a-z][a-z0-9]*                          (ID);
[0-9]+                                  (NUM);
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)     (REAL);
("--"[a-z]*"\n")|(" "|"\n"|"\t")+       (continue());
.                                       (error(); continue());
```

FIGURE 2.2. Regular expressions for some tokens.

Longest match: The longest initial substring of the input that can match any regular expression is taken as the next token.
Rule priority: For a **particular** longest initial substring, the first regular expression that can match determines its token type. This means that the order of writing down the regular-expression rules has significance.
So according to the rules, if8 match as a single identifier and not as the two tokens if and 8. And "if 89" begin with a reserved word and not by an identifier by rule priority rule.

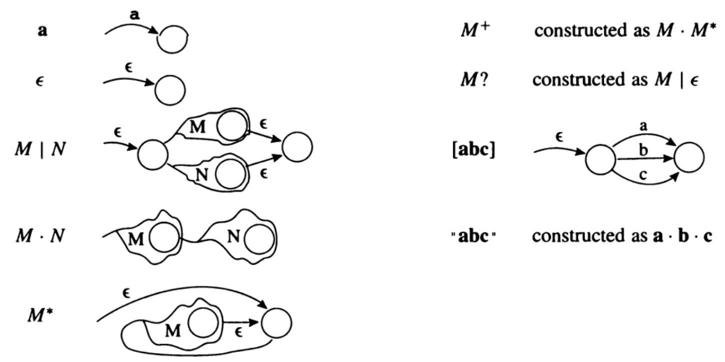


FIGURE 2.6. Translation of regular expressions to NFAs.

$$\textbf{DFAedge}(d, c) = \textbf{closure}\Big(\bigcup_{s \in d} \textbf{edge}(s, c)\Big)$$

Using **DFAedge**, we can write the NFA simulation algorithm more formally. If the start state of the NFA is s_1, and the input string is $c_1, \dots, c_k$, then the algorithm is:

```
$d \leftarrow$ closure($\{s_1\}$)
for $i \leftarrow 1$ to $k$
  $d \leftarrow$ DFAedge($d, c_i$)
```

[redacted] Abstractly, there is an edge from d_i to d_j labeled with c if d_j = **DFAedge**(d_i, c). We let Σ be the alphabet.

```
states[0] $\leftarrow$ {};    states[1] $\leftarrow$ closure($\{s_1\}$)
$p \leftarrow 1$;    $j \leftarrow 0$
while $j \le p$
  foreach $c \in \Sigma$
    $e \leftarrow$ DFAedge(states[$j$], $c$)
    if $e$ = states[$i$] for some $i \le p$
      then trans[$j, c$] $\leftarrow i$
      else $p \leftarrow p + 1$
           states[$p$] $\leftarrow e$
           trans[$j, c$] $\leftarrow p$
  $j \leftarrow j + 1$
```

DFA construction is a mechanical task easily performed by computer, so it makes sense to have an automatic lexical analyzer generator to translate regular expressions into a DFA.
The output of ML-Lex is a program in ML - a lexical analyzer that interprets a DFA using the algorithm described in Section 2.3 and executes

the action fragments on each match. The action fragments are just ML statements that return token values.

```
(* ML Declarations: *)
    type lexresult = Tokens.token
    fun eof() = Tokens.EOF(0,0)
    %%
(* Lex Definitions: *)
    digits=[0-9]+
    %%
(* Regular Expressions and Actions: *)
    if                  => (Tokens.IF(yypos,yypos+2));
    [a-z][a-z0-9]*      => (Tokens.ID(yytext,yypos,yypos+size yytext));
    {digits}            => (Tokens.NUM(Int.fromString yytext,
                                 yypos,yypos+size yytext));
    ({digits}"."[0-9]*)|([0-9]*"."{digits})
                        => (Tokens.REAL(Real.fromString yytext,
                                        yypos, yypos+size yytext));
    ("--"[a-z]*"\n")|(" "|"\n"|"\t")+
                        =>    (continue());
                        =>    (ErrorMsg.error yypos "illegal character";
                               continue());
```

PROGRAM 2.9. ML-Lex specification of the tokens from Figure 2.2.

The format is: user declarations %% ML-Lex definitions %% rules
The first part of the specification, above the first %% mark, contains functions and types written in ML. These must include the type lexresult, which is the result type of each call to the lexing function; and the function eof, which the lexing engine will call at end of file. This section can also contain utility functions for the use of the semantic actions in the third section. It is called with the same argument as lex (see %arg, below), and must return a value of type lexresult.
The second part of the specification contains regular-expression abbreviations and state declarations. For example, the declaration digits=$[0-9]^+$ in this section allows the name {digits} to stand for a nonempty sequence of digits within regular expressions.
In the definitions section, the user can define named regular expressions, a set of start states, and specify which of the various bells and whistles of ML-Lex are desired.
The third part contains regular expressions and actions. The actions are fragments of ordinary ML code. Each action must return a value of type lexresult. In this specification, lexresult is a token from the Tokens structure.
In the action fragments, several special variables are available. The string matched by the regular expression is yytext. The file position of the beginning of the matched string is yypos. The function continue () calls the lexical analyzer recursively.

In this particular example, each token is a data constructor parameterized by two integers indicating the position – in the input file – of the beginning and end of the token.

```
structure Tokens =
struct
    type pos = int
    datatype token = EOF of pos * pos
                   | IF of pos * pos
                   | ID of string * pos * pos
                   | NUM of int * pos * pos
                   | REAL of real * pos * pos
                        ⋮
end
```

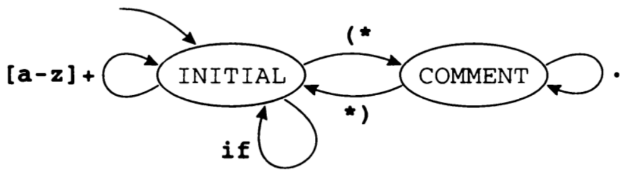
Arguments given to token are called payload.
The tokens are defined by the combined effect of
1. The %term commands used in the ML-Yacc declaration section of your ML-Yacc specification. These may add extra values to the token function's argument and thus extend the payload.
2. The lexresult type declaration in the user declarations of your ML-Lex specification
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:

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

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));
```

But sometimes the step-by-step, state-transition model of automata is appropriate. ML-Lex has a mechanism to mix states with regular expressions. One can declare a set of start states; each regular expression can be prefixed by the set of start states in which it is valid. The action fragments can explicitly change the start state. In effect, we have a finite automaton whose edges are labeled, not by single symbols, but by regular expressions. This example shows a language with simple identifiers, if tokens, and comments delimited by (* and *) brackets:



The ML-Lex specification corresponding to this machine is

```
        the usual preamble ...
%%
%s COMMENT
%%
<INITIAL>if          => (Tokens.IF(yypos,yypos+2));
<INITIAL>[a-z]+      => (Tokens.ID(yytext,yypos,
                                  yypos+size(yytext)));
<INITIAL>"(*"        => (YYBEGIN COMMENT; continue());
<COMMENT>"*)"        => (YYBEGIN INITIAL; continue());
<COMMENT>.           => (continue();)
```

This example can be easily augmented to handle nested comments, via a global variable that is incremented and decremented in the semantic actions.
Any regular expression not prefixed by a $<$ *state* $>$ operates in all states; this feature is rarely useful.
Certain rules

- An individual character stands for itself, except for the reserved characters

  ```
  ? * + | ( ) ^ $ / ; . = < > [ { " \
  ```

- A backslash followed by one of the reserved characters stands for that character.
- Inside the brackets, only the symbols

  ```
  \ - ^
  ```

  are reserved. An initial up-arrow ˆ stands for the complement of the characters listed, e.g. [ˆabc] stands any character except a, b, or c.
- To include ˆ 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 for any character except newline, i.e. the same as

  ```
  [^\n]
  ```

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

  ```
  \b backspace
  \n newline
  \t horizontal tab
  \ddd where ddd is a 3 digit decimal escape
  ```

- Any regular expression may be enclosed in parentheses ( ) for syntactic (but, as usual, not semantic) effect
- A sequence of characters will stand for itself (reserved characters will be taken literally) if it is enclosed in double quotes " ".
- A postfix repetition range {a, b} where a and b are small integers stands for any number of repetitions between a and b of the preceding expression. The notation {a} stands for exactly a repetitions. Ex: [0-9]{3} Any three-digit decimal number.
- 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.
- The user may recursively call the lexing function with lex(). (If %arg is used, the lexing function may be re-invoked with the same argument by using continue().) This is convenient for ignoring white space or comments silently:

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

- To switch start states, the user may call YYBEGIN with the name of a start state.

- 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
- Running ML - Lex file: From the Unix shell, run

```
sml-lex myfile.lex
The output file will be myfile.lex.sml. The
extension .lex is not required but is recommended.
```

To get messages for lexer errors and unwelcome characters (note: l1 is the lineno. and l2 is the position in that line):

```
val error : string * int * int -> unit = fn
(e,l1,l2) => TextIO.output(TextIO.stdOut,"lex:line "
^Int.toString l1^" l2="^Int.toString l2
^": "^e^"\n")
val badCh : string * string * int * int -> unit = fn
(fileName,bad,line,col) =>
TextIO.output(TextIO.stdOut,fileName^"["
^Int.toString line^"."^Int.toString col
^"] Invalid character \""^bad^"\"\n");
```

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.

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

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
The ML-Lex definitions section provides the following commands. They are all terminated with a semicolon ;
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 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
%count counts newlines using yylineno
%posarg Pass an initial-position argument to function makeLexer. See 10.4. %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:

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

The argument value is passed in the call to the parser. See line 415 on page 43
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
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 nn and have an action that increments a line-number variable. For example, see chapter 11.2.3 on page 38 in our working example.

```
datatype lexresult= DIV | EOF | EOS | ID of string | LPAREN |
NUM of int | PLUS | PRINT | RPAREN | SUB | TIMES

val linenum = ref 1
val error = fn x => output(std_out,x ^ "\n")
val eof = fn () => EOF
%%
%structure CalcLex
alpha=[A-Za-z];
digit=[0-9];
ws = [\ \t];
%%
\n        => (inc linenum; lex());
{ws}+     => (lex());
"/"       => (DIV);
";"       => (EOS);
"("       => (LPAREN);
(* revfold ((('a * 'b)->'b) ->'a list ->'b -> 'b) is like fold
done from backwards (in this case from left) explode will
convert the string to list of characters. *)
{digit}+ => (NUM (revfold (fn(a,r)=>ord(a)-ord("0")+10*r)
(explode yytext) 0));
")"       => (RPAREN);
"+"       => (PLUS);
{alpha}+ => (if yytext="print" then PRINT else ID yytext);
"-"       => (SUB);
"*"       => (TIMES);
.         => (error ("calc: ignoring bad character "^yytext);
lex());
```

2 Parsing

The parser returns an abstract syntax tree of the expression being evaluated. The parser gets tokens from the scanner to parse the input and build the AST. When an AST is returned by the parser, the compiler calls the code generator to evaluate the tree and produce the target code.
There are two main parts to a compiler, the front end and back end. The front end reads the tokens and builds an AST of a program. The back end generates the code given the AST representation of the program. As presented in earlier chapters, the front end consists of the scanner and the parser.

1 $S \rightarrow S ; S$	4 $E \rightarrow \text{id}$	8 $L \rightarrow E$
2 $S \rightarrow \text{id} := E$	5 $E \rightarrow \text{num}$	9 $L \rightarrow L , E$
3 $S \rightarrow \text{print} (L)$	6 $E \rightarrow E + E$	
	7 $E \rightarrow (S , E)$	

GRAMMAR 3.1. A syntax for straight-line programs.

As before, we say that a language is a set of strings; each string is a finite sequence of symbols taken from a finite alphabet. For parsing, the strings are source programs, the symbols are lexical tokens, and the alphabet is the set of token types returned by the lexical analyzer.
A leftmost derivation is one in which the leftmost nonterminal symbol is always the one expanded; in a rightmost derivation, the rightmost nonterminal is always next to be expanded.
A parse tree is made by connecting each symbol in a derivation to the one from which it was derived, as shown in Figure 3.3. Two different derivations can have the same parse tree.
A grammar is ambiguous if it can derive a sentence with two different parse trees.
Parsers must read not only terminal symbols such as +, -, num, and so on, but also the end-of-file marker. We will use $ to represent end of file.

Suppose S is the start symbol of a grammar. To indicate that \$ must come after a complete S-phrase, we augment the grammar with a new start symbol S' and a new production S' → S\$.

Predictive Parsing: Some grammars are easy to parse using a simple algorithm known as recursive descent. Predictive parsing works only on grammars where the first terminal symbol of each subexpression provides enough information to choose which production to use.

$S \to E\ \$$

$E \to E + T$ $\quad T \to T * F$ $\quad F \to \text{id}$

$E \to E - T$ $\quad T \to T\ /\ F$ $\quad F \to \text{num}$

$E \to T$ $\quad T \to F$ $\quad F \to (\ E\ )$

GRAMMAR 3.10.

$S \to \text{if } E \text{ then } S \text{ else } S$ $\quad L \to \text{end}$

$S \to \text{begin } S\ L$ $\quad L \to \ ;\ S\ L$

$S \to \text{print } E$ $\quad E \to \text{num} = \text{num}$

GRAMMAR 3.11.

```
datatype token = IF | THEN | ELSE | BEGIN | END | PRINT
               | SEMI | NUM | EQ

val tok = ref (getToken())
fun advance() =  tok := getToken()
fun eat(t) = if (!tok=t) then advance() else error()

fun S() = case !tok
           of IF => (eat(IF); E(); eat(THEN); S();
                     eat(ELSE); S())
            | BEGIN => (eat(BEGIN); S(); L())
            | PRINT => (eat(PRINT); E())
and L() = case !tok
           of END => (eat(END))
            | SEMI => (eat(SEMI); S(); L())
and E() = (eat (NUM); eat(EQ); eat(NUM))
```

ith suitable definitions of `error` and `getToken`, this program will parse ry nicely.

Emboldened by success with this simple method, let us try it with Gramar 3.10:

```
fun S() = (E(); eat(EOF))
and E() = case !tok
           of ? => (E(); eat(PLUS); T())
            | ? => (E(); eat(MINUS); T())
            | ? => (T())
and T() = case !tok
           of ? => (T(); eat(TIMES); F())
            | ? => (T(); eat(DIV); F())
            | ? => (F())
and F() = case !tok
           of ID => (eat(ID))
            | NUM => (eat(NUM))
            | LPAREN => (eat(LPAREN); E(); eat(RPAREN))
```

Given a string γ of terminal and nonterminal symbols, FIRST(γ) is the set of all terminal symbols that can begin any string derived from γ. For example, let $\gamma = T * F$. Any string of terminal symbols derived from γ must start with `id`, `num`, or `(`. Thus,

FIRST($T * F$) = {`id`, `num`, `(`}.

If two different productions $X \to \gamma_1 and X \to \gamma_2$ have the same left- hand-side symbol (X) and their right-hand sides have overlapping FIRST sets, then the grammar cannot be parsed using predictive parsing.

With respect to a particular grammar, given a string γ of terminals and nonterminals,

- nullable(X) is true if X can derive the empty string.
- FIRST(γ) is the set of terminals that can begin strings derived from γ.
- FOLLOW(X) is the set of terminals that can immediately follow X. That is,

$t \in$ FOLLOW(X) if there is any derivation containing Xt. This can occur i the derivation contains $X\,Y\,Z\,t$ where Y and Z both derive ϵ.

A precise definition of FIRST, FOLLOW, and nullable is that they are the smallest sets for which these properties hold:

```
For each terminal symbol $Z$, FIRST[$Z$] = {$Z$}.
for each production $X \to Y_1 Y_2 \cdots Y_k$
    if $Y_1 \ldots Y_k$ are all nullable (or if $k = 0$)
        then nullable[$X$] = true
    for each $i$ from 1 to $k$, each $j$ from $i + 1$ to $k$
        if $Y_1 \cdots Y_{i-1}$ are all nullable (or if $i = 1$)
            then FIRST[$X$] = FIRST[$X$] ∪ FIRST[$Y_i$]
        if $Y_{i+1} \cdots Y_k$ are all nullable (or if $i = k$)
            then FOLLOW[$Y_i$] = FOLLOW[$Y_i$] ∪ FOLLOW[$X$]
        if $Y_{i+1} \cdots Y_{j-1}$ are all nullable (or if $i + 1 = j$)
            then FOLLOW[$Y_i$] = FOLLOW[$Y_i$] ∪ FIRST[$Y_j$]
```

The set of sym- bols is the union of the non-terminal and terminal sets. 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 corre- sponding 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.
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
After the first %%, the following words and symbols are reserved:

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

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.
In ML-YACC user declarations section 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.
The ML-Yacc declarations section is used to make a set of required declarations and a set of optional declarations.
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.
You may declare whether the parser generator should create a verbose description of the parser in a ".desc" file. 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 eval- uation 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).
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
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 types cannot be polymorphic. Do not use any locally defined types from the user declarations section of the specification Terminals are written in Capitals whereas non terminals are written in small.
Consider
%nonterm elabel of (symbol * exp)
Here since the supplemented payload is a tuple, parenthesis is required.
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:
107 %arg (fileName) : string
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.
You should list the precedence declarations in order of increasing (tighter-binding) prece- dence. 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 prece- dence level, but you cannot specify non-terminals. The keywords are %left , %right , and %nonassoc , standing for their respective associativities.
The %nodefault declaration suppresses the generation of default reductions
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
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.
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.
Specify all keywords in your grammar here. The %keyword should be followed by a list of terminal names
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.
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.

ML-YACC Rules

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

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

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.

```
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 con- structions 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:

```
144 %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

```
145 {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)
```

Add %prec declarations to the rules to say which terminal's precedence and as- sociativity are to be used with which rules
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))
```