卐 Compiler Short Revision Notes 卐

Sourabh Aggarwal

Compiled on January 12, 2019

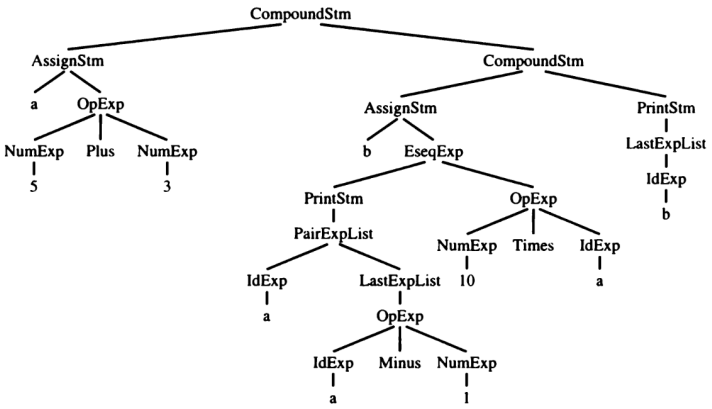
Contents

1 Intro

Lexical means relating to the words or vocabulary of a language.
Most useful abstraction are context free grammer for parsing and regular expressions for lexical analysis. Yacc which converts a grammer into a parsing program, Lex which converts a declarative specification into lexical analysis program.
Useful resource: Click
Language of straight line programs:
The informal semantics of the language is as follows. Each *Stm* is a statement, each *Exp* is an expression. $s_1; s_2$ executes statement s_1, then statement s_2. i:=e evaluates the expression e, then "stores" the result in variable i.
print $(e_1, e_2, \ldots, e_n)$ displays the values of all the expressions, evaluated left to right, separated by spaces, terminated by a newline. An identifier expression, such as i, yields the current contents of the variable i. A number evaluates to the named integer. An operator expression e_1 op e_2 evaluates e1, then e2, then applies the given binary operator. And an expression sequence (s, e) behaves like the C-language "comma" operator, evaluating the statement s for side effects before evaluating (and returning the result of) the expression e. For example, executing this program a := 5+3; b := (print(a, a-1), 10*a); print(b) prints
8 7
80

Stm → *Stm* ; *Stm*	(CompoundStm)
Stm → `id :=` *Exp*	(AssignStm)
Stm → `print (` *ExpList* `)`	(PrintStm)
Exp → `id`	(IdExp)
Exp → `num`	(NumExp)
Exp → *Exp Binop Exp*	(OpExp)
Exp → (*Stm* , *Exp*)	(EseqExp)

ExpList → *Exp* , *ExpList*	(PairExpList)
ExpList → *Exp*	(LastExpList)
Binop → +	(Plus)
Binop → −	(Minus)
Binop → ×	(Times)
Binop → /	(Div)




```
type id = string
datatype binop = Plus | Minus | Times | Div
datatype stm = CompoundStm of stm * stm
             | AssignStm of id * exp
             | PrintStm of exp list
     and exp = IdExp of id
             | NumExp of int
             | OpExp of exp * binop * exp
             | EseqExp of stm * exp
```

To translate a program from one language into another, a compiler must first pull it apart and understand its structure and meaning, then put it together in a different way. The front end of the compiler performs analysis; the back end does synthesis. The analysis is usually broken up into Lexical analysis: breaking the input into individual words or "tokens";The lexical analyzer takes a stream of characters and produces a stream of names, keywords, and punctuation marks; it discards white space and comments between the tokens.
Syntax analysis: parsing the phrase structure of the program; and
Semantic analysis: calculating the program's meaning.
we will specify lexical tokens using the formal language of regular expressions, implement lexers using deterministic finite automata, and use mathematics to connect the two. This will lead to simpler and more readable lexical analyzers. $(a \odot b)|\epsilon$ 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 [bedefg], [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.

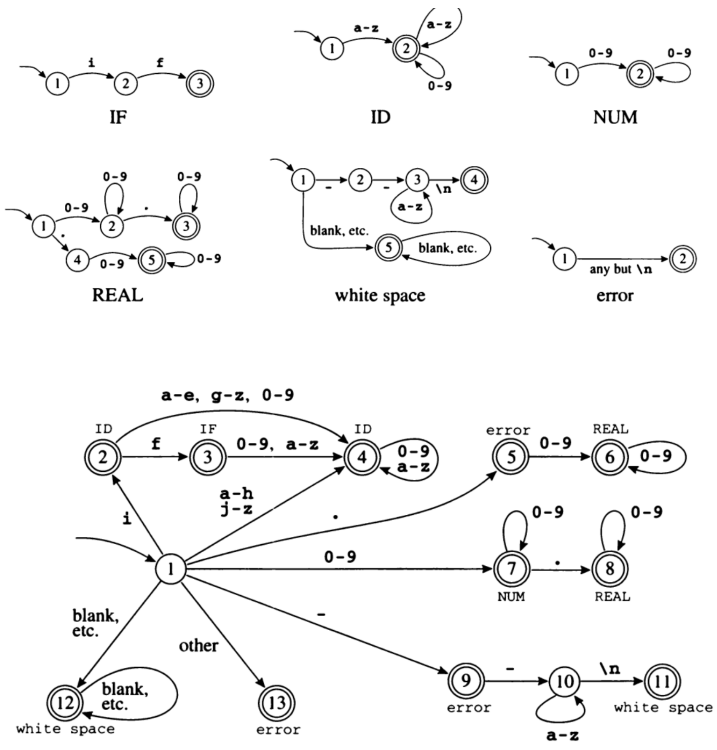


FIGURE 2.4. Combined finite automaton.

We can encode this machine as a transition matrix: a two-dimensional array (a vector of vectors), subscripted by state number and input character. There will be a "dead" state (state 0) that loops to itself on all characters; we use this to encode the absence of an edge.

```
val edges =
 vector[          (*       ···0 1 2···-···e f g h i j···*)
(* state 0 *) vector[0,0,···0,0,0···0···0,0,0,0,0,0···],
(* state 1 *) vector[0,0,···7,7,7···9···4,4,4,4,2,4···],
(* state 2 *) vector[0,0,···4,4,4···0···4,3,4,4,4,4···],
(* state 3 *) vector[0,0,···4,4,4···0···4,4,4,4,4,4···],
(* state 4 *) vector[0,0,···4,4,4···0···4,4,4,4,4,4···],
(* state 5 *) vector[0,0,···6,6,6···0···0,0,0,0,0,0···],
(* state 6 *) vector[0,0,···6,6,6···0···0,0,0,0,0,0···],
(* state 7 *) vector[0,0,···7,7,7···0···0,0,0,0,0,0···],
   et cetera
       ]
```

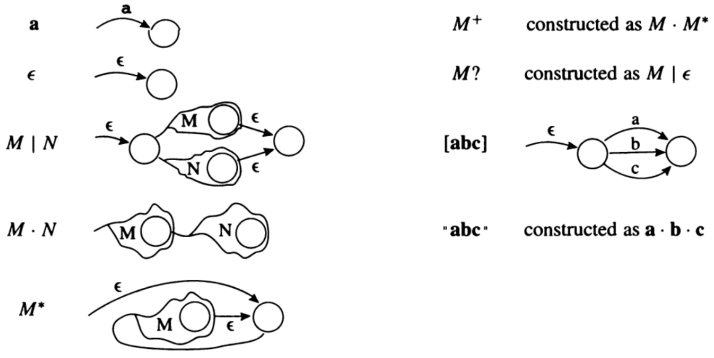


FIGURE 2.6. Translation of regular expressions to NFAs.

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

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 $c_1, \ldots, c_k$, then the algorithm is:

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

Abstractly, there is an edge from d_i to d_j labeled with c if $d_j = \textbf{DFAedge}(d_i, c)$. We let Σ be the alphabet.

states[0] $\leftarrow$ {}; states[1] $\leftarrow$ **closure**($\{s_1\}$)
$p \leftarrow 1$; $j \leftarrow 0$
while $j \leq p$
  foreach $c \in \Sigma$
    $e \leftarrow \textbf{DFAedge}(\text{states}[j], c)$
    if $e = \text{states}[i]$ for some $i \leq p$
      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.
ML-Lex is a lexical analyzer generator that produces an ML program from a lexical specification.
For each token type in the programming language to be lexically analyzed, the specification contains a regular expression and an action. The action communicates the token type (perhaps along with other information) to the next phase of the compiler.
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.

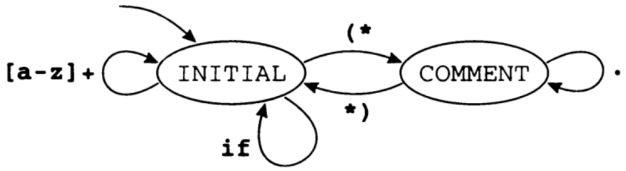
The first part of the specification, above the first %% mark, contains functions and types written in ML. These must include the type lexresult, which is the result type of each call to the lexing function; and the function eof, which the lexing engine will call at end of file. This section can also contain utility functions for the use of the semantic actions in the third section.
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.
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
```

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.