Lex and Yacc: Tools for Generating Compiler Frontends

Matthai Philipose
October 13, 2000

Overview

input program

lexical analyzer

generated by Lex

token

get_next_token()

parser

generated by Yacc

parse tree

parse()

rest of compiler

Interface to the Lex & Yacc

RE -> Lex -> Lexical Analyzer
CFG -> Yacc -> Parser

%ls

pl_1.l pl_1.y

%lex pl_1.l; ls

pl_1.l pl_1.y lex.yy.c

%yacc pl_1.y; ls

pl_1.l pl_1.y lex.yy.c

y.tab.c y.tab.h

An Example Language: PL/-1

factorial in PL/-1:

#Input assumed to # be in “arg”
n := arg;
result:=1;
if arg < 0 then
  #error condition
  return -1
else
  while n > 1 do
    result:=result*n;
    n := n -1;
  return result
Grammar for PL/-1

cmd ::= cmd ; cmd
   | id  := expr
   | if  expr then cmd else cmd end
   | while expr do cmd end
   | return expr

expr ::= id
   | num
   | expr binop expr
   | ( expr )

binop ::= + | - | * | /
   | = | < | >

Tokens for PL/-1

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Token</th>
<th>Attribute Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;whitespace&gt;</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>;</td>
<td>&quot;;&quot;</td>
<td>-</td>
</tr>
<tr>
<td>+</td>
<td>&quot;+&quot;</td>
<td>-</td>
</tr>
<tr>
<td>#</td>
<td>&quot;#&quot;</td>
<td>-</td>
</tr>
<tr>
<td>:=</td>
<td>COLON_EQ</td>
<td>-</td>
</tr>
<tr>
<td>if</td>
<td>IF</td>
<td>-</td>
</tr>
<tr>
<td>then</td>
<td>THEN</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;identifiers&gt;</td>
<td>ID</td>
<td>string</td>
</tr>
<tr>
<td>&lt;nat. number&gt;</td>
<td>NUM</td>
<td>int</td>
</tr>
</tbody>
</table>

Format of Lex Specification

declarations
%%

translation rules
%%

helper functions

Declarations include C declarations and regular expression (RE) shorthands

Translation rules have form

    RE1{action 1}
    ...
    RE{n}{action n}

Helper functions are C functions invoked from the actions

Lex: Declarations

{%
#define COLON_EQ 256
#define IF 257
#define THEN 258
...

typedef union{
    int i;
    char *s;
} attribute;

attribute yylval;
}%

ws   = [\t\b\n]+
alpha = [a-zA-Z]
num   = [0-9]
alpha_num = (alpha|num)
ident  = alpha(alpha_num)+
cmnt   = \\
    \#.\.*$
Lex RE Syntax

<table>
<thead>
<tr>
<th>Expr.</th>
<th>Matches</th>
<th>Eg</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>non-operator character c</td>
<td>x</td>
</tr>
<tr>
<td>\c</td>
<td>any character c</td>
<td>\</td>
</tr>
<tr>
<td>&quot;s&quot;</td>
<td>string s literally</td>
<td>&quot;foo\n&quot;</td>
</tr>
<tr>
<td>.</td>
<td>any character but newline</td>
<td>z.*</td>
</tr>
<tr>
<td>^ or $</td>
<td>beginning/end of line</td>
<td>^.*$</td>
</tr>
<tr>
<td>[s]</td>
<td>any character in s</td>
<td>[abc]</td>
</tr>
<tr>
<td>[^s]</td>
<td>any character not in s</td>
<td>[^abc]</td>
</tr>
<tr>
<td>r*</td>
<td>zero or more r's</td>
<td>[abc]*</td>
</tr>
<tr>
<td>r+</td>
<td>one or more r's</td>
<td>[abc]+</td>
</tr>
<tr>
<td>r?</td>
<td>zero or one r</td>
<td>x?y*</td>
</tr>
<tr>
<td>r{m,n}</td>
<td>m to n r's</td>
<td>a[2,7]</td>
</tr>
<tr>
<td>r{r2}</td>
<td>r then r2</td>
<td>x<em>y</em></td>
</tr>
<tr>
<td>r/r2</td>
<td>r1 or r2</td>
<td>x</td>
</tr>
<tr>
<td>(r)</td>
<td>r</td>
<td>a(x*)</td>
</tr>
<tr>
<td>[c1-c2]</td>
<td>one of characters of numbers</td>
<td>[a-2]</td>
</tr>
<tr>
<td>[m-n]</td>
<td>in specified ranges</td>
<td>[2-6]</td>
</tr>
</tbody>
</table>

Lex: Translation Rules

```
{ws}+ {}  
{cmnt} {}  
";" {return ‘;’;}  
"+" {return ‘+’;}  
...  
":=" {return COLON_EQ;}  
"if" {return IF;}  
...  
{id} {set_id_attr(); return ID;}  
{num}+ {set_num_attr(); return NUM;} 
. {error();}
```

Lex: Helper Functions

```
void set_id_attr(){
  yylval.s=mk_yy_str();}

void set_num_attr(){
  int i= atoi(mk_yy_str());
  yylval.s = i;}

char *mk_yy_str(){
  char * c=
    malloc(sizeof(char)*
         (yyleng+1));
  strncpy(c, yytext, yyleng);
  c[yyleng] = ‘\0’;
  return c;}

void error(){
  fprintf(stderr,”\n”);
  exit(-1);}
```

Lex: A Tricky Common Case

Consider comments of the form:

```
/* ... */
```

Attempted solution 1:

```
cmnt = /\*(\n| .)*\*/
%
...
{cmnt}{}
...
```

Possible solution 2:

```
"/*" {find_cmnt_end()}
%
void find_cmnt_end(){
... while (!c_end_found){
  ...getc()...
  }
}
Lex: Using Explicit States

...%state COMMENT
%{
{ws}+
{}<YYINITIAL> "/*"
{yybegin(COMMENT);}
</COMMENT> "/*"
{error();}
</COMMENT> . {}

</COMMENT> "*/"
{yybegin(YYINITIAL);}

<YYINITIAL> ";"
{return '";'}
...

<YYINITIAL> .
{error();}

Lex: The Big Picture

Invoking lex on a .l file produces lex.yy.c file

lex.yy.c contains:

• Function token yylex(void)
• yylex() also defines (by side-effect) the union yylval
• Declarations and helper functions copied verbatim

yylex() consists of:

transition table for finite automaton (FA)
C-code to simulate FA, with action code invoked at accept states

Format of YACC Specification

declarations
%%
translations
%%
helper functions

Declarations contain
C decl’s
Yacc-specific decl’s

Translations have form
\( p_1 \{a_1\} \)
\[
\ldots
\]

\( p_n \{a_n\} \)

where \( p_i \) are productions
and \( a_i \) are C-code actions

Helper functions are C-code

YACC: Declarations

%{/* Type declarations */
typedef union expr_s{
char *id; int num;
struct{binop op,
union expr_s* e1,
union expr_s* e2}* op_expr
}* expr;

typedef union cmd_u{
struct {union cmd_u* c1,
union cmd_u* c2}* seq;
struct {char* id,
expr e}* asst;
\[
\ldots
\]
}* cmd;

/*Type constructor fwd decl’s*/
expr mk_id(char *);
expr mk_seq(cmd,cmd);
}%

%token
ID COLON_EQ IF THEN ELSE WHILE DO END RETURN NUM
YACC: Translation

```c
expr mk_id(char *s){
    expr e= malloc(sizeof(union expr_u));
    e->s = s;
    return e;
}
```

```c
cmd mk_seq(cmd c1,cmd c2){
    cmd c3= malloc(sizeof(union cmd_u));
    c3->seq.c1=c1;
    c3->seq.c2=c2;
    return c3;
}
```

YACC: Helper Functions

```
expr mk_id(char *s){
    expr e = malloc(
        sizeof(union expr_u));
    e->s = s;
    return e;
}
```

```
cmd mk_seq(cmd c1,cmd c2){
    cmd c3 = malloc(
        sizeof(union cmd_u));
    c3->seq.c1=c1;
    c3->seq.c2=c2;
    return c3;
}
```

Could define `yylex()` here
Commonly just link with `lex.yy.c`

Common Problem: Ambiguities

Yacc (in verbose mode) generates `y.output` file to report conflicts

A somewhat cleaned up entry:
S/R conflict (shift LESS_THAN, reduce by rule 5)
S/R conflict (shift EQUALS, reduce by rule 5)
S/R conflict (shift DIVIDE, reduce by rule 5)
S/R conflict (shift TIMES, reduce by rule 5)
S/R conflict (shift MINUS, reduce by rule 5)
S/R conflict (shift PLUS, reduce by rule 5)

```
E : E . BINOP E
E : E BINOP E .
    (reduce by rule 5)
```

Associativity-Related Ambiguity

```
x - y * - z
```

Shift or reduce on seeing second `-` ?
Two legal parse trees:

Need notion of associativity:

```
+, _, /, * left associative
=, < not associative
```
Precedence-Related Ambiguity

\[ x - y \cdot * z \]

Shift or reduce on seeing ‘∗’?
Two legal parse trees:

Need notion of precedence
\{ /,∗ \} > \{ +,- \} > \{ >,= \}

Solution Attempt: Do Nothing

Yacc has default conflict resolution:
Shift when shift/reduce conflict
Reduce when reduce/reduce conflict

Parse tree on string \( x - y \cdot * z \)?

Parse tree on string \( x - y - z \)?

Solution 1: Rewrite Grammar

\[
\begin{align*}
E & : E_{as} \\
E_{as} & : E_m \mid E_{as} '-' E_m \mid E_{as} '+' E_m \\
E_m & : Einp \mid E_m '\cdot' Einp \mid E_m '/' Einp \\
Einp & : ID \mid NUM \mid '(' Einp ')' \\
\end{align*}
\]

Parse tree on string \( x - y \cdot * z \)?

Solution 2: Associativity and Precedence Declarations

\{ * . . . * \}

%token . . .
%left ‘;’
%noassoc ‘<’, ‘>’, ‘=’
%left ‘-’, ‘+’
%left ‘/’, ‘∗’
%
. . .
%
. . .

Later associativity declarations have higher precedence
Summary

If you ever need to parse character strings into datastructures, think Lex and Yacc.

Very often, tokens are (almost) REs, grammar is CFG.