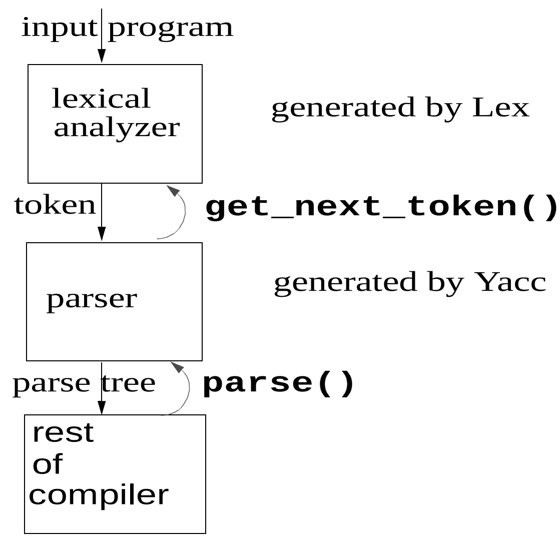
Lex and Yacc: Tools for Generating Compiler Frontends

Matthai Philipose
October 13, 2000

Overview



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

Regular Expression	Token	Attribute Value Type
<whitespace>	-	-
;	‘;’	-
+	‘+’	-
#	‘#”	-
...		
:=	COLON_EQ	-
if	IF	-
then	THEN	-
...		
<identifiers>	ID	string
<nat. number>	NUM	int

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}
...
REn{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

Expr.	Matches	Eg
c	non-operator character *c*	x
\c	any character *c*	\\
"*s*"	string *s* literally	"foo*\"
.	any character but newline	z.*
^ or $	beginning/end of line	^.*$
[*s*]	any character in *s*	[abc]
[^*s*]	any character not in *s*	[^abc]
*r**	zero or more r's	[abc]*
r+	one or more r's	[abc]+
r?	zero or one r	x?y*
r{*m*,*n*}	*m* to *n* *r*'s	a{2,7}
r_1r_2	*r1* then *r2*	x*y*
$r_1\|r_2$	*r1* or *r2*	x*|y*
(*r*)	r	a(x*)
[c1-c2] or [m-n]	one of characters of numbers in specified ranges	[a-z] or [2-6]

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 $\{a_1\}$
...
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;
...
}* cmd;

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

%token
ID COLON_EQ IF THEN ELSE WHILE
DO END RETURN NUM
```

YACC: Translation

```
cmd :
 cmd ';' cmd
  {$$ = mk_seq($1,$3);}
|ID COLON_EQ expr
  {$$ = mk_asst($1,$3);}
|IF expr THEN cmd ELSE cmd END
  {$$ = mk_if($2,$4,$6);}
|WHILE expr DO cmd END
  {$$ = mk_while($2,$4);}
|RETURN expr
  {$$ = mk_ret($2);}

expr:
 ID    {$$ = mk_id($1)}
|NUM   {$$ = mk_num($1)}
|expr binop expr
  {$$ = mk_binop($2,$1,$3);}
|'(' expr ')'

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


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 . * 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 * z ?

Parse tree on string x - y - z ?

Solution 1: Rewrite Grammar

E : E_as

E_as : Em | Eas '-' Em | Eas '+' Em

E_m : Einp | E_m * Einp | E_m / Einp

Einp: ID | NUM | '(' Einp ')'

(actions omitted)

Parse tree on string x - y * 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.