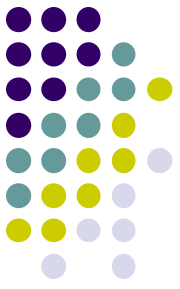


LEX & YACC Tutorial

Nov, 2009

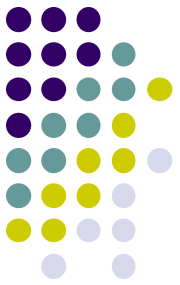
Gil Kulish

Outline



- Overview of Lex and Yacc
- Structure of Lex Specification
- Structure of Yacc Specification
- Some Hints for Lab1

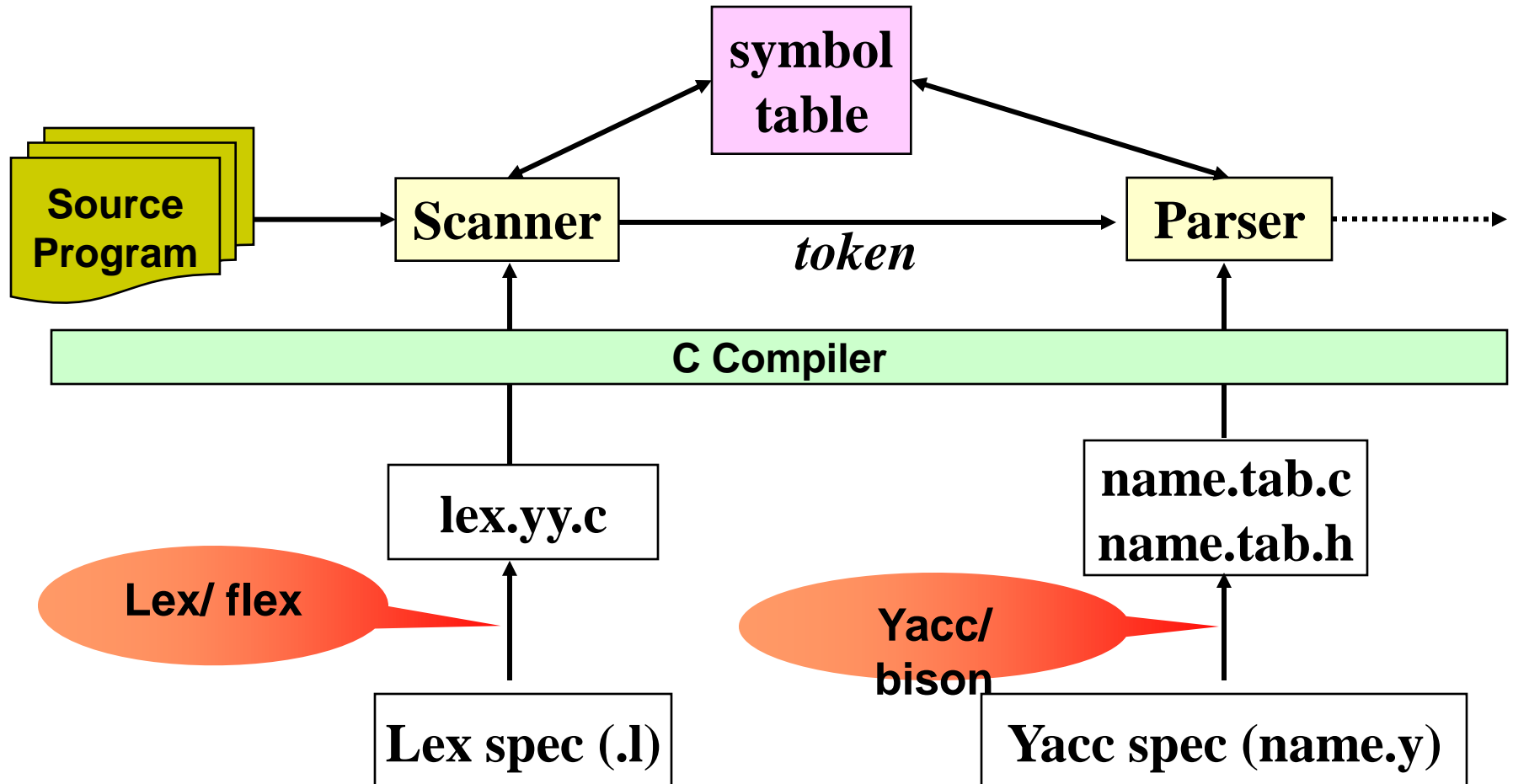
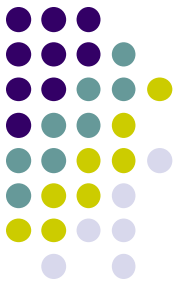
Overview



- Lex (A LEXical Analyzer Generator)
generates lexical analyzers (scanners or Lexers)
- Yacc (Yet Another Compiler-Compiler)
generates parser based on an analytic grammar
- Flex is Free fast scanner alternative to Lex
<http://flex.sourceforge.net/>
- Bison is Free parser generator program
written for the GNU project alternative to Yacc



Scanner, Parser, Lex and Yacc



Skeleton of a Lex Specification (.l file)

x.l

lex.yy.c is generated after running

```
> lex x.l
```

```
%{
```

```
< C global variables, prototypes, comments, ect' >
```

```
%}
```

→ This part will be embedded into lex.yy.c

```
[flex DEFINITION SECTION]
```

```
%%
```

```
[flex RULES SECTION]
```

→ Define how to scan and what action to take for each token

```
%%
```

C auxiliary subroutines

→ Any user code.

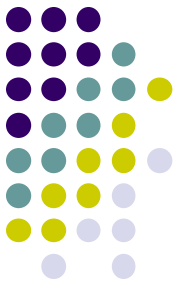
Lex Specification: Definition Sections

```
%{  
#include "y.tab.h"  
#include <stdlib.h>  
int res=0;  
char operation='+';  
void someFuncThatIsDefinedLater();  
  
%}
```

→ User-defined header file

```
DIGIT [0-9]  
NUMBER [1-9]{DIGIT}*  
%%
```

Lex Specification: Rules Section



- Format

```
pattern      { corresponding actions }  
...  
pattern      { corresponding actions }
```

↑
Regular
Expression

↑
C Expression

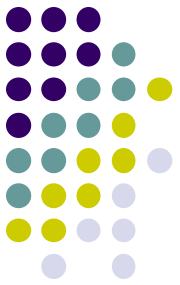
- Example

```
\n          printf("end of input");  
[1-9][0-9]*      {  
                  if (operation=='+')  
                    res+=atoi(ytext);  
                  }  
.  
                /*do nothing*/
```

↙
Unsigned integer will be
accepted as a token

Instead of `[1-9][0-9]*`, could have
used `{NUMBER}`

Two Notes on Using Lex



1. Lex matches token with **longest match**

Input: *abc*

Rule: `[a-z]+`

→ Token: *abc* (not “a” or “ab”)

2. Lex uses the **first applicable rule**

for the Input: *post*

Rule1: `“post”` `{printf (“Hello,”) ; }`

Rule2: `[a-zA-Z]+` `{printf (“World!”) ; }`

→ It will print Hello, (not “World!”)

Flex Code compilation



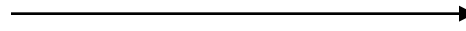
- Flex filename.l
- Gcc -o executableName lex.yy.c -lfl
- * when connecting to yacc (bison), other functions might be needed:

```
int yywrap(void) {  
    return 1;  
}
```

- Also, the .y file needs to be compiled first (-d)

Skeleton of a Yacc Specification (.y file)

x.y



x.tab.c is generated after running

```
> yacc x.y
```

```
%{
```

< C global variables, prototypes, comments >

```
%}
```

[DEFINITION SECTION]

Declaration of tokens recognized in Parser (Lexer), assuming parser declared

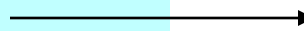
```
#include "y.tab.h"
```

and that the .y file is compiled first using:

```
bison -d filename.y
```

```
%%
```

[PRODUCTION RULES SECTION]



· **How to understand** the input, and **what actions** to take for each "sentence".

```
%%
```

C auxiliary subroutines

Yacc Specification: Definition Section (1)

tree.l

```
[1-9][0-9]*      { yylval.dval = atoi (yytext);  
                    return NUMBER;  
                  }
```

tree.y

```
%{  
#include <string.h>  
int flag = 0;  
  
%}
```

```
%union {  
    int dval; ...  
}
```

```
%token <dval> NUMBER
```

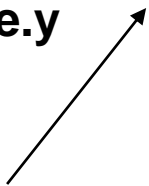
Yacc Specification: Definition Section (2)

tree.l

```
[a-zA-Z]*      { yylval= yytext;  
                  return ID;  
                }
```

```
%{  
#include <string.h>  
int flag = 0;  
  
%}  
  
#define YYSTYPE char*
```

tree.y



An alternative to the %union.
The default YYSTYPE type is int

Yacc Specification: Definition Section (3)

Define operator's precedence and associativity

- We can solve problem in slide 15

```
%left '-' '+'  
%left '*' '/' '%'\  
%right '='
```

```
%type <dval> expression statement statement_list  
%type <dval> logical_expr
```

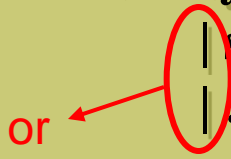
Define nonterminal's name

-With this name, you will define rules in rule section
-This definition is not mandatory (so does the <dval>),
no need to use in the HW

Yacc Specification: Production Rule Section (1)

- Format

```
nontermname : symbol1 symbol2 ... { corresponding actions }  
             | symbol3 symbol4 ... { corresponding actions }  
             | ...  
             ;  
nontermname2 : ...
```



Regular expression

C expression

Yacc Specification: Production Rule Section (2)

- Example

```
statement : expression { printf (" = %g\n", $1); }
expression : expression '+' expression { $$ = $1 + $3; }
           | expression '-' expression { $$ = $1 - $3; }
           | expression '*' expression { $$ = $1 * $3; }
           | NUMBER
```

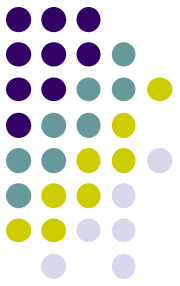
\$\$: final value by performing non-terminal's action, Only for writing, not reading
\$n: value of the nth concatenated element

→ What will happen if we have input “2+3*4”?

Avoiding Ambiguous Expression

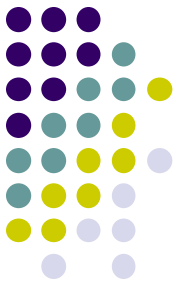
That's the reason why we need to define operator's precedence in definition section

Just one more reserved symbol - start



- Bison assumes by default that the start symbol for the grammar is the first nonterminal specified in the grammar specification section. The programmer may override this restriction with the `%start` declaration as follows:
- `%start symbol`

Hints for Lab1



Exercise 1, question 3

- **Q: How to recognize “while”, “for” and “break” in Lexer?**
- **A: Step1: Add these rules to your .l file:**

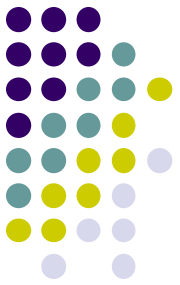
```
%%  
“break”      { return BREAK;}  
“while”      { return WHILE;}  
“for”        { return FOR;}  
...  
%%
```

→ Should be put in the rule section

→ Case-sensitive

Step2: declare WHILE, FOR and BREAK as “token” in your .y file

Hints for Lab1

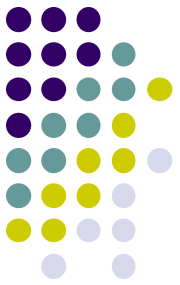


Exercise 1 question 3

- **Q: How would I create a tree from the Grammar?**
- **A: Think Recursively**

```
%%  
..  
expression:      ...  
                |  
                expression '+' expression{  
                ...  
                $$->left=$1;  
                $$->right=$3;  
                }  
                |  
                ...
```

Hints for Lab1



Exercise 1 question 3

Q: How to build up and print AST

1. Define the struct for AST and linked list structure having AST nodes.

```
typedef struct treeNode{  
    sometype left_exp;  
    sometype right_exp;  
    anothertype operator;  
  
} AST;
```

2. In y file, your statement and expressions should be 'ast' type.

A case study – The Calculator



zcalc.l

zcalc.y

Yacc -d zcalc.y

```
%{
#include "zcalc.tab.h"
#include "y.tab.h"

}%

%%
([0-9]+|([0-9]*\.[0-9]+)([eE][+-]?[0-9]+)?
    { yylval.dval = atof(yytext);
      return NUMBER; }

[ \t] ;
[a-zA-Z][a-zA-Z0-9-()]*
    { struct symtab *sp = symlook(yytext);
      yylval.symp = sp;
      return NAME;
    }

%%
```

```
%{
#include "zcalc.h"
}%

%union { double dval; struct symtab *symp; }

%token <symp> NAME
%token <dval> NUMBER

%left '+' '-'

%type <dval> expression

%%

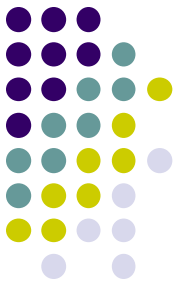
statement_list : statement '\n' | statement_list statement '\n'
statement : NAME '=' expression { $1->value = $3; }
           | expression { printf (" = %g\n", $1); }

expression : expression '+' expression { $$ = $1 + $3; }
            | expression '-' expression { $$ = $1 - $3; }
            | NUMBER { $$ = $1; }
            | NAME { $$ = $1->value; }

%%

struct symtab * symlook( char *s )
{ /* this function looks up the symbol table and check whether the
   symbol s is already there. If not, add s into symbol table. */
}

int main() {
    yyparse();
    return 0;
}
```



References

- Lex and Yacc Page

<http://dinosaur.compilertools.net>