LEX & YACC Tutorial

February 28, 2008

Tom St. John

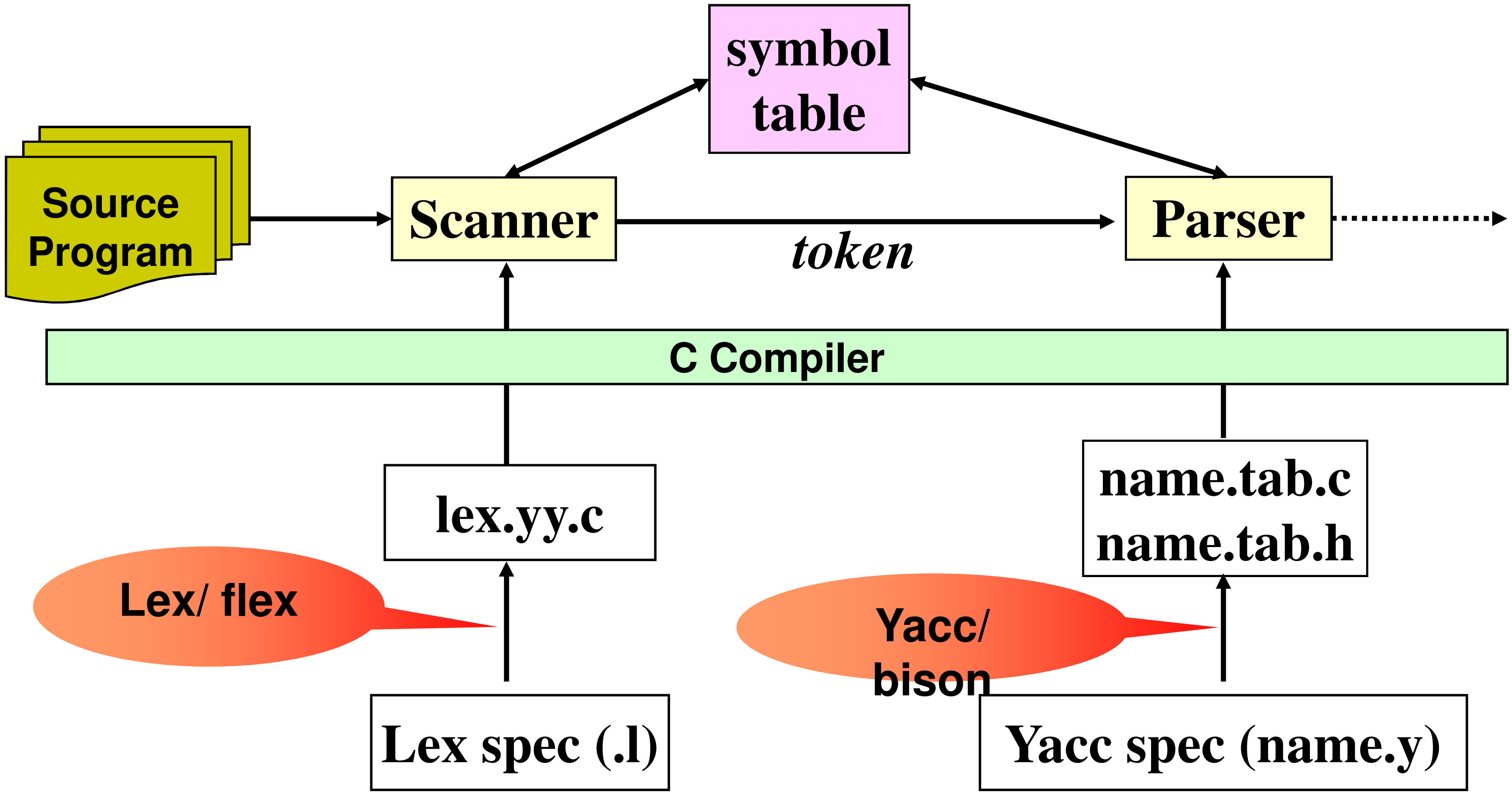
Outline

- 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 scanner alternative to Lex
- 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 >

%}

[DEFINITION SECTION]

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

%%

[RULES SECTION] → Define how to scan and what action to take for each token

%%

C auxiliary subroutines → Any user code.

Lex Specification: Definition Section

```
%{

#include "zcalc.tab.h"
#include "zcalc.h“
#include <math.h>



%}
```

- `#include "zcalc.tab.h"` → You should include this! Yacc will generate this file automatically.
- `#include "zcalc.h"` → User-defined header file

Lex Specification: Rules Section

- Format

```
pattern          { corresponding actions }
…
pattern          { corresponding actions }
```

↑ Regular Expression (pattern) ↑ C Expression (corresponding actions)

- Example

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

Unsigned integer will be accepted as a token

You need to define these two in .y file (dval, NUMBER)

Two Notes on Using Lex

1. Lex matches token with longest match

Input: *abc*
Rule: `[a-z]+`

→ Token: *abc* (not "*a*" or "a*b*")

2. Lex uses the first applicable rule

Input: *post*
Rule1: `"post"          {printf ("Hello,"); }`
Rule2: `[a-zA-Z]+       {printf ("World!"); }`

→ It will print Hello, (not "World!")

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).

%%

[PRODUCTION RULES SECTION] → How to understand the input, and what actions to take for each "sentence".

%%

C auxiliary subroutines

Yacc Specification: Definition Section (1)

zcalc.l

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

zcalc.y

```
%{

#include "zcalc.h"
#include <string.h>

int flag = 0;

%}

%union {
  int dval; …
}

%token <dval> NUMBER
```

Yacc Specification: Definition Section (2)

Define operator's precedence and associativity
- We can solve problem in slide 13

```
%left '-' '+'
%left '*' '/' '%'
```

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

Define nonterminal's name
- With this name, you will define rules in rule section

Yacc Specification: Production Rule Section (1)

- Format

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

or

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 n^{th} 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

Hints for Lab1

Exercise 2

- **Q: How to recognize “prefix”, “postfix” and “infix” in Lexer?**
- **A: Step1: Add these rules to your .l file:**

```
%%
    “prefix”            { return PREFIX;}
    “postfix”           { return POSTFIX; }
    “infix”             { return INFIX;}
…
%%
```

→ Should be put in the rule section

→ Case-sensitive

Step2: declare PREFIX, POSTFIX and INFIX as “token” in your .y file

Hints for Lab1

Exercise 2

- **Q: How to combine three modes together?**
- **A: You may have following grammar in your yacc file**

```
int flag = 0;  // Default setting

%%
..
statement: PREFIX { flag = 0; }       |
                  INFIX  { flag = 1; }       |
                  POSTFIX { flag = 2; }      |
             expression
                 …
expression: expr_pre | expr_in | expr_post;

expr_pre: '+' expr_pre expr_pre { if(flag == 0) $$ = $2 + $3; }
…
expr_in: expr_in ‘+’ expr_in    { if(flag == 1) $$ = $1 + $3; }
…
```

Hints for Lab1

Exercise 3

- **Q: What action do we use to define the octal and hexadecimal token?**

- A: You can simply use ‘strtol’ functions for this.

```
long strtol(const char *nptr, char **endptr, int base);
```

Hints for Lab1

Exercise 4-5

Q: How to build up and print AST

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

```
typedef struct EXP{ struct EXP* exp1;
                    struct EXP* exp2;
                    struct OP    operator;
 } AST;
```

→ Instead of using struct,
if you use union here,
It's easier to handle the terminal nodes (name and numbers)

2. In yacc file, your statement and expressions should be 'ast' type (no longer dval type).

Hints for Lab1

Exercise 4-5

3. Functions for making expression. It can be different functions by the type of the node (kinds of expression, number, name and so on). You can make functions like,

```
makeExpression(struct EXP* exp1, struct EXP* exp2, struct OP operator)
```

→ The action field for each production in your yacc file can call any function you have declared. Just as a sentence is recursively parsed, your AST is recursively built-up and traversed.

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-(]*
                    { 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