

**课 程 实 验 报 告**

**课程名称： 编 译 原 理**

**简单词法分析器实验**

**LR分析器构造实验**

**专业班级： 计算机1012班**

**学 号： U201014592**

**姓 名： 苏宙行**

**指导教师： 徐丽萍**

**报告日期： 2013年6月9日**

**计算机科学与技术学院**

目录

[1. 实验一 词法分析器实验 1](#_Toc358323875)

[1.1. 实验内容 1](#_Toc358323876)

[1.2. 实验要求 1](#_Toc358323877)

[1.3. 程序概要设计 1](#_Toc358323878)

[1.4. 输入输出设计 1](#_Toc358323879)

[1.5. 源程序及注释 1](#_Toc358323880)

[1.6. 程序测试与结果 6](#_Toc358323881)

[1.7. 小结 6](#_Toc358323882)

[2. 实验二 LR分析器构造实验 7](#_Toc358323883)

[2.1. 实验内容 7](#_Toc358323884)

[2.2. 实验要求 7](#_Toc358323885)

[2.3. 程序概要设计 7](#_Toc358323886)

[2.4. 输入输出设计 7](#_Toc358323887)

[2.5. 源程序及注释 9](#_Toc358323888)

[2.6. 程序测试与结果 19](#_Toc358323889)

[2.7. 小结 19](#_Toc358323890)

[参考文献 21](#_Toc358323891)

# 实验一 词法分析器实验

## 实验内容

调研词法分析程序的自动生成工具LEX或FLEX，设计一个类C语言的单词描述文法，设计并实现一个能够输出单词序列的词法分析器。

## 实验要求

(1)理解单词的分类和形式化描述；

(2)掌握从文法到最小确定有穷自动机的转换方法；

(3)用软件实现有穷自动机来识别标识符和常数；

(4)定义保留字和操作符、界符的内部码；

(5)实现一个完整的词法分析器。

## 程序概要设计

采用自定义状态的方法实现对块注释的识别；

通过对关键字表进行二分查找判断一个符合标识符命名规则的字符串是否为关键字，如是返回非负数，否则返回-1，然后根据返回值进行相应的输出。

使用SetConsoleTextAttribute调整控制台输出的字符的颜色属性；

## 输入输出设计

输入：一段标准的C语言源代码，涉及注释、预处理语句、标识符、关键字、界符、运算符、常量等词法元素。（测试样例参见2.4节）

输出：在控制台输出根据词法元素分析得出的语法高亮标记，在result.txt输出词法元素的类型、内部编号、值等信息，每个词法元素占一行。

## 源程序及注释

文件：“src/CCompiler/LexicalElementHighlighter.l”【lex源文件】

digit [0-9]

letter [\_A-Za-z]

PreprocessStatment ^#.\*

word {letter}({letter}|{digit})\*

constant ({digit}\*\.?{digit}\*|\".\*\"|\'(.|\\.)\')

operator ([~!%^&\*\-+=<>/()]|"<="|">="|"=="|"!="|"++"|"--"|"<<"|">>"|"<<="|">>="|"+="|"-="|"\*="|"/="|"%="|"&="|"|="|"^="|"&&"|"||"|sizeof)

token [\{\}\[\]\;\,]

comment "//".\*

%x BLOCKCOMMENT

%{

#include <windows.h>

#include <wincon.h>

#include <stdlib.h>

#include <string.h>

#define TOKEN\_ATTR (14)

#define KEYWORD\_ATTR (9)

#define OPERATOR\_ATTR (7)

#define CONSTANT\_ATTR (13)

#define IDENTIFIER\_ATTR (11)

#define PREPROCESS\_ATTR (4)

#define COMMENT\_ATTR (2)

#define ERROR\_ATTR (12)

int yywrap(void);

int resKeyword( char\* word );

int wordCmp( const void \*a, const void \*b );

int tokenNum = 0;

int keywordNum = 0;

int operatorNum = 0;

int constantNum = 0;

int identifierNum = 0;

int preprocStmtNum = 0;

int commentNum = 0;

HANDLE Ghstdout;

FILE \*resultFile;

%}

%%

{comment} {

SetConsoleTextAttribute( Ghstdout , COMMENT\_ATTR );

printf( "%s", yytext );

fprintf( resultFile, "注释：%s\n", yytext );

commentNum++;

}

"/\*" {

BEGIN BLOCKCOMMENT;

}

<BLOCKCOMMENT>"\*/" {

BEGIN 0;

}

<BLOCKCOMMENT>. {

;

}

<BLOCKCOMMENT>"\n" {

;

}

{PreprocessStatment} {

SetConsoleTextAttribute( Ghstdout , PREPROCESS\_ATTR );

printf( "%s", yytext );

fprintf( resultFile, "预处理语句：%s\n", yytext );

preprocStmtNum++;

}

{constant} {

SetConsoleTextAttribute( Ghstdout , CONSTANT\_ATTR );

printf( "%s", yytext );

fprintf( resultFile, "常量：%s\n", yytext );

operatorNum++;

}

{operator} {

int id = 0;

SetConsoleTextAttribute( Ghstdout , OPERATOR\_ATTR );

while( yyleng > 0 ) {

id += yytext[--yyleng];

}

printf( "%s", yytext );

fprintf( resultFile, "操作符(%d)：%s\n", id, yytext );

constantNum++;

}

{word} {

int id;

if ( (id = resKeyword( yytext )) >= 0 ) {

SetConsoleTextAttribute( Ghstdout , KEYWORD\_ATTR );

printf( "%s", yytext );

fprintf( resultFile, "关键字(%d)：%s\n", id, yytext );

keywordNum++;

} else {

SetConsoleTextAttribute( Ghstdout , IDENTIFIER\_ATTR );

printf( "%s", yytext );

fprintf( resultFile, "标识符：%s\n", yytext );

identifierNum++;

}

}

{token} {

SetConsoleTextAttribute( Ghstdout , TOKEN\_ATTR );

printf( "%s", yytext );

fprintf( resultFile, "界符：%s\n", yytext );

tokenNum++;

}

[^ \t\n] {

SetConsoleTextAttribute( Ghstdout , ERROR\_ATTR );

printf( "%s", yytext );

fprintf( resultFile, "非法字符：%s\n", yytext );

}

%%

int main(void) {

freopen( "in.txt", "r", stdin );

Ghstdout = GetStdHandle( STD\_OUTPUT\_HANDLE );

resultFile = fopen( "result.txt", "w" );

yylex();

printf("\n-------------------------------");

SetConsoleTextAttribute( Ghstdout , PREPROCESS\_ATTR );

printf("\npreprocess statment: %d\n", preprocStmtNum);

SetConsoleTextAttribute( Ghstdout , TOKEN\_ATTR );

printf("token: %d\n", tokenNum);

SetConsoleTextAttribute( Ghstdout , KEYWORD\_ATTR );

printf("keyword: %d\n",keywordNum);

SetConsoleTextAttribute( Ghstdout , OPERATOR\_ATTR );

printf("operator: %d\n",operatorNum);

SetConsoleTextAttribute( Ghstdout , CONSTANT\_ATTR );

printf("constant: %d\n",constantNum);

SetConsoleTextAttribute( Ghstdout , IDENTIFIER\_ATTR );

printf("identifier: %d\n",identifierNum);

SetConsoleTextAttribute( Ghstdout , COMMENT\_ATTR );

printf("comment: %d\n",commentNum);

return 0;

}

int yywrap(void) {

return 1;

}

int wordCmp( const void \*a, const void \*b ) {

return strncmp( (const char\*)a, (const char\*)b, 9 );

}

int resKeyword( char\* word ) {

static const char keywords[32][10] = {"auto", "break", "case", "char", "const", "continue", "default", "do", "double",

"else","enum", "extern","float", "for", "goto", "if", "int", "long","register", "return", "short", "signed", "sizeof",

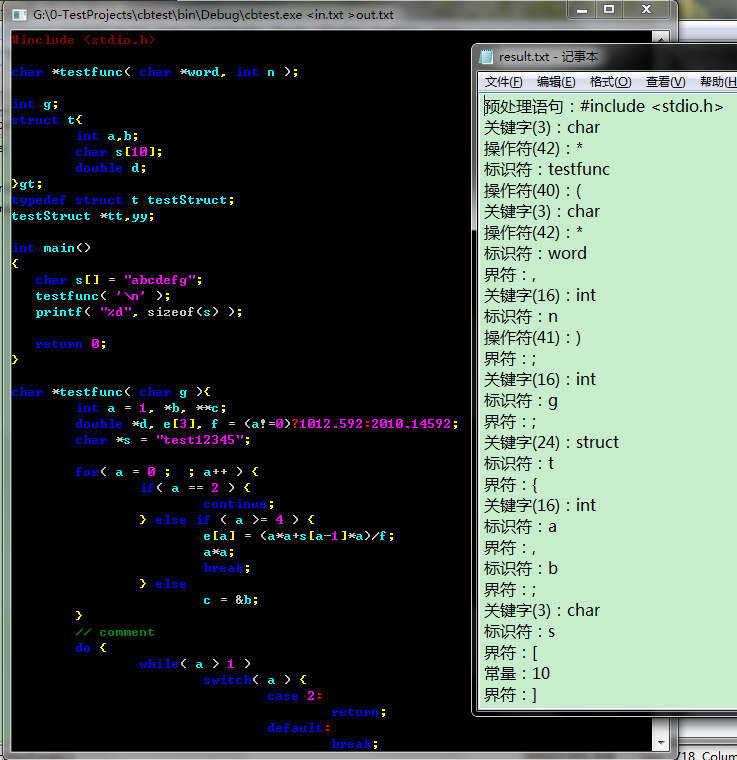
"static","struct", "switch", "typedef", "union", "unsigned", "void","volatile", "while" };

char \* p = (char \*)bsearch( word, keywords, 32, 10, wordCmp );

return (p==NULL)?(-1):(( p - keywords[0])/10);

}

## 程序测试与结果



## 小结

以下为遇到的问题：

处理块注释时，需要用到下面的两个特性：

多行匹配：(.|\n)\* 成功解决该问题；

最小匹配：(.|\n)\*? 未实现懒惰匹配（最小匹配）。

如上所述，flex似乎不支持懒惰匹配，于是，参照《Lex And Yacc Tutorial》中的自定义状态的方法，解决了块注释的问题。但是在阅读过程中发现，其实字符串常量也会有类似的问题，由于时间紧迫，我暂时还没有着手解决。

另外，在常量匹配中，暂时无法判断字符（串）中的转义序列是否正确。

# 实验二 LR分析器构造实验

## 实验内容

调研语法分析器的自动生成工具YACC的功能与工作原理，使用YACC生成一个自底向上的语法分析器。

## 实验要求

(1)调研语法生成工具YACC；

(2)根据实验2定义的语法规则，编写YACC源程序；

(3)用YACC生成语法分析程序；

(4)测试语法分析器的功能。

## 程序概要设计

根据需要重写lex的规则，适当去除一些不必要的规则，补充之前未考虑到的规则和词法元素，然后将实验一中的输出信息的语句改成return词法元素的标号（实际上是一个枚举变量）。

对于yacc的规则文件，先参照C语言教材上的运算符优先级与结合性表将所有操作符定义好准确的优先级与结合性（另外还有针对if-else语句的冲突的优先级规定）。然后整个语法以program为开始符，产生函数声明、变量定义、函数实现、类型定义。然后再逐层展开，例如最核心的部分——函数实现（C语言的所有逻辑和操作都在函数中实现）——由函数原型和一个block组成，block中有多条statement，每个statement可以是表达式、流程控制语句、类型定义、标签，每个表达式可以是表达式和运算符进行运算……

定义好规则后，在适当的地方加入对应的C语言语句，执行符合实验要求的输出。由于实验要求识别不同的语句的类型，这些输出大多在以stmt为左部的产生式中、流程控制语句的产生式中。

最后，在main函数中调用yyparse()函数，yyparse函数会自动调用yylex函数获取词法元素的类型，完成对C语言代码的语法分析。

本分析器未实现的功能参见lex、yacc源文件开头的注释（英文）。

## 输入输出设计

文件：“src/CCompiler/in.txt”【输入文件】

#include <stdio.h>

char \*testfunc( char \*word, int n );

int g;

struct t{

int a,b;

char s[10];

double d;

}gt;

typedef struct t testStruct;

testStruct \*tt,yy;

int main()

{

char s[] = "abcdefg";

testfunc( '\n' );

printf( "%d", sizeof(s) );

return 0;

}

char \*testfunc( char g ){

int a = 1, \*b, \*\*c;

double \*d, e[3], f = (a!=0)?1012.592:2010.14592;

char \*s = "test12345";

/\* testStruct \*tt;

this statement will be recognized as expression because of conflict ! \*/

for( a = 0 ; ; a++ ) {

if( a == 2 ) {

continue;

} else if ( a >= 4 ) {

e[a] = (a\*a+s[a-1]\*a)/f;

a\*a;

break;

} else

c = &b;

}

// comment

do {

while( a > 1 )

switch( a ) {

case 2:

return;

default:

break;

}

} while ( --a );

return s;

}

文件：out.txt / stdout【输出文件】

向控制台或out.txt输出每一条语句的类型。

## 源程序及注释

文件：“src/CCompiler/lex.l”【lex源文件】

/\*

stringconst and (charconst?) did not consider the escape sequences such as \n and \"

also, stringconst do not support cross-line strings and can not recognize "xx"xx" as an error

do not recognize enum value as constant

\*/

digit [0-9]

letter [\_A-Za-z]

numconst {digit}\*\.?{digit}\*

stringconst \".\*\"

charconst (\'(.|\\.)\')

PreprocessStatment ^#.\*

word {letter}({letter}|{digit})\*

constant ({numconst}|{stringconst}|{charconst})

singleCharToken ({singleCharOperator}|{singleCharDelimiter})

singleCharOperator [~!%^&\*\-+=<>/()\,\[\]]

singleCharDelimiter [\?\:\{\}\;]

%x BLOCKCOMMENT

%{

#include <stdlib.h>

#include <string.h>

#include "yacc.tab.h"

int yywrap(void);

int lineno = 1;

%}

%%

"//".\* { ; }

"/\*" { BEGIN BLOCKCOMMENT; }

<BLOCKCOMMENT>"\*/" { BEGIN 0; }

<BLOCKCOMMENT>. { ; }

<BLOCKCOMMENT>\n { ++lineno; }

{PreprocessStatment} { ;/\* return PREP\_STMT; \*/ }

{constant} { return CONSTANT; }

{singleCharToken} { return \*yytext; }

"<=" { return LE; }

">=" { return GE; }

"==" { return EQ; }

"!=" { return NEQ; }

"++" { return INC; }

"--" { return DEC; }

"<<" { return SHL; }

">>" { return SHR; }

"<<=" { return SHLASS; }

">>=" { return SHRASS; }

"+=" { return ADDASS; }

"-=" { return SUBASS; }

"\*=" { return MULASS; }

"/=" { return DIVASS; }

"%=" { return MODASS; }

"&=" { return ANDASS; }

"|=" { return ORASS; }

"^=" { return XORASS; }

"&&" { return AND; }

"||" { return OR; }

"sizeof" { return SIZEOF; }

"auto" { return KW\_AUTO; }

"const" { return KW\_CONST; }

"extern" { return KW\_EXTERN; }

"register" { return KW\_REGISTER; }

"static" { return KW\_STATIC; }

"volatile" { return KW\_VOLATILE; }

"typedef" { return KW\_TYPEDEF; }

"enum" { return KW\_ENUM; }

"struct" { return KW\_STRUCT; }

"union" { return KW\_UNION; }

"char" { return KW\_CHAR; }

"double" { return KW\_DOUBLE; }

"float" { return KW\_FLOAT; }

"int" { return KW\_INT; }

"long" { return KW\_LONG; }

"short" { return KW\_SHORT; }

"signed" { return KW\_SIGNED; }

"unsigned" { return KW\_UNSIGNED; }

"void" { return KW\_VOID; }

"break" { return KW\_BREAK; }

"case" { return KW\_CASE; }

"continue" { return KW\_CONTINUE; }

"default" { return KW\_DEFAULT; }

"do" { return KW\_DO; }

"else" { return KW\_ELSE; }

"for" { return KW\_FOR; }

"goto" { return KW\_GOTO; }

"if" { return KW\_IF; }

"return" { return KW\_RETURN; }

"switch" { return KW\_SWITCH; }

"while" { return KW\_WHILE; }

{word} {

return ID;

}

\n { ++lineno; }

[ \t] { ; }

. { printf( "line %d : invalid character(s)：%s\n", lineno, yytext ); }

%%

int yywrap(void) {

return 1;

}

文件：“src/CCompiler/yacc.y”【yacc源文件】

/\*

int switch clause, constant after "case" can only be constant parsed by lex

and regard constant expression as error,

but think real number and string constant correct .

varinit can be extend to fit the initialization of array and struct .

do not support explicit type cast like "int i = (int)c;" in the "expr" production .

the error "void a" in "void f(void a)" may not be recognize .

do not support type modifiers such as "const" .

"name" needs to be modified to fit the rule that "int a[][3]" is right and "int a[3][]" is wrong .

\*/

%{

#include <stdio.h>

#include "yacc.tab.h"

#define log(msg) syntaxAnalasisLog(msg,0)

int yylex(void);

void yyerror(const char \*s);

void syntaxAnalasisLog( const char \*msg, int linenoShift );

extern int lineno;

%}

// -------------------------

%start program

// -------------------------

%token KW\_AUTO KW\_CONST KW\_EXTERN KW\_REGISTER KW\_STATIC KW\_VOLATILE KW\_TYPEDEF

%token KW\_CHAR KW\_DOUBLE KW\_FLOAT KW\_INT KW\_LONG KW\_SHORT KW\_SIGNED KW\_UNSIGNED KW\_VOID

%token KW\_STRUCT KW\_UNION KW\_ENUM

%token KW\_BREAK KW\_CASE KW\_CONTINUE KW\_DEFAULT KW\_DO KW\_ELSE KW\_FOR KW\_GOTO KW\_IF KW\_RETURN KW\_SWITCH KW\_WHILE

%token ID

%token CONSTANT

// -------------------------

%left ','

%right '=' ADDASS SUBASS MULASS DIVASS MODASS ANDASS XORASS ORASS SHLASS SHRASS

%right '?' ':'

%left OR

%left AND

%left '|'

%left '^'

%left '&'

%left EQ NEQ // "==" "!="

%left '<' LE '>' GE // '<' "<=" '>' ">="

%left SHL SHR // "<<" ">>"

%left '+' '-'

%left '\*' '/' '%'

%right '!' '~' INC DEC POS NEG DEREF ADDROF SIZEOF // '!' '~' "++" "--" '+' '-' '\*' '&' "sizeof" //

%left ARROW '.' '[' ']' '(' ')' // "->"

%nonassoc IFX

%nonassoc KW\_ELSE

%% // =================================

program:

program typeDefineStmt

| program varDeclStmt { log("global variable definition"); }

| program funcDecl

| program funcImplement

|

| error ';'

| error '}'

;

// -------------------------

typeDefineStmt:

typeDefine ';'

;

typeDefine:

KW\_TYPEDEF varType ID { log("user type definition"); }

| compondTypeDef

;

compondTypeDef:

KW\_STRUCT ID { log("struct definition"); } '{' varDecls '}'

| KW\_UNION ID { log("union definition"); } '{' varDecls '}'

| KW\_ENUM ID { log("enum definition"); } '{' enumList '}'

;

varDecls:

varDecls varDeclStmt

|

;

enumList:

ID

| enumList ',' ID

;

// -------------------------

name: // in variable declarations and function definition ( as '\*' is with ID but not simple type )

ID

| '\*' name

| name '[' expr ']'

| name '[' ']'

;

varDeclStmt:

varDecl ';'

;

varDecl:

varType varInit

| varDecl ',' varInit

;

varInit:

name

| name '=' expr

| name '=' '{' expr '}'

;

varType:

atomType

| userDefType

;

userDefType:

compondTypeDef

| KW\_STRUCT ID

| KW\_UNION ID

| KW\_ENUM ID

| ID // 2 reduce/reduce conflicts

;

atomType:

KW\_CHAR

| KW\_DOUBLE

| KW\_FLOAT

| KW\_INT

| KW\_LONG

| KW\_SHORT

| KW\_SIGNED

| KW\_UNSIGNED

| KW\_VOID

;

// -------------------------

funcDecl:

funcDef ';' { log("function declaration"); }

;

funcDef:

varType name '(' argvList ')'

;

argvList: // used in function declarations

varType name

| argvList ',' varType name

| KW\_VOID

|

;

// -------------------------

funcImplement:

funcDef { syntaxAnalasisLog("function implementation", -1); } block

;

block:

'{' stmts '}'

;

stmts:

stmts stmt

|

;

stmt:

block

| expr ';' { log("expression"); }

| typeDefineStmt

| varDeclStmt { log("local variable definition"); }

| for

| while

| dowhile

| if

| switch

| ';' { log("empty statement"); }

| KW\_RETURN expr ';' { log("return statement"); }

| KW\_RETURN ';' { log("return statement"); }

| KW\_BREAK ';' { log("break statement"); }

| KW\_CONTINUE ';' { log("continue statement"); }

| KW\_GOTO ID ';' { log("goto statement"); }

| ID ':' /\* label of "goto" \*/ { log("lable of goto"); }

| error ';'

| error '}'

;

expr:

// either left or right value expression

ID

| expr '[' expr ']'

| '\*' expr %prec DEREF

| expr ARROW expr

| expr '.' expr

| '(' expr ')'

// left value expression

| expr '=' expr

| expr ADDASS expr

| expr SUBASS expr

| expr MULASS expr

| expr DIVASS expr

| expr MODASS expr

| expr ANDASS expr

| expr XORASS expr

| expr ORASS expr

| expr SHLASS expr

| expr SHRASS expr

| INC expr

| DEC expr

// right value expression

| CONSTANT

| SIZEOF expr

| SIZEOF '(' varType ')'

| funcCall

| '&' expr %prec ADDROF

| expr '?' expr ':' expr

| expr ',' expr

| expr OR expr

| expr AND expr

| expr '|' expr

| expr '^' expr

| expr '&' expr

| expr EQ expr

| expr NEQ expr

| expr '<' expr

| expr LE expr

| expr '>' expr

| expr GE expr

| expr SHL expr

| expr SHR expr

| expr '+' expr

| expr '-' expr

| expr '\*' expr

| expr '/' expr

| expr '%' expr

| '!' expr

| '~' expr

| expr INC

| expr DEC

| '+' expr %prec POS

| '-' expr %prec NEG

;

funcCall:

ID '(' expr ')' { log("function call"); }

;

if:

KW\_IF '(' expr ')' stmt KW\_ELSE stmt { log("if-else clause"); }

| KW\_IF '(' expr ')' stmt %prec IFX { log("if clause"); }

;

while:

KW\_WHILE '(' expr ')' { log("while clause"); } stmt

;

dowhile:

KW\_DO stmt KW\_WHILE '(' expr ')' ';' { log("do-while clause"); }

;

for:

KW\_FOR '(' forExpr ';' forExpr ';' forExpr ')' { log("for clause"); } stmt

;

forExpr:

expr

|

;

switch:

KW\_SWITCH '(' expr ')' { log("switch clause"); } '{' cases defaultCase '}'

;

cases:

case

| case cases

;

case:

KW\_CASE CONSTANT ':' { log("case in switch"); } stmts

;

defaultCase:

KW\_DEFAULT ':' { log("default case in switch"); } stmts

|

;

/\*

argv: // used in function calls

expr

| argv ',' expr // replaced by expr with comma

;

\*/

%% // =================================

void yyerror(const char \*msg) {

printf("Error: %s (line %d)\n", msg, lineno);

}

void syntaxAnalasisLog( const char \*msg, int linenoShift ) {

printf( "line %5d : %s\n", (lineno + linenoShift), msg );

}

int main() {

freopen( "in.txt", "r", stdin );

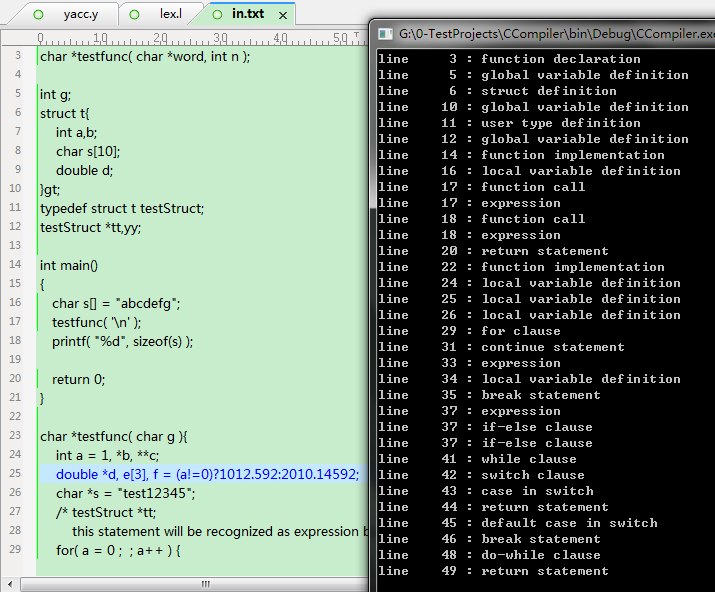
//freopen( "out.txt", "w", stdout );

yyparse();

return 0;

}

## 程序测试与结果



## 小结

一开始把表达式的产生式直接使用OP（用%token定义）表示所有运算符，结果有大量shift/reduce和reduce/reduce冲突，后来经过多次尝试发现了上述问题。由于这样的冲突无法避免，我只能将所有运算符分开定义规则，然后为所有运算符定义结合性和优先级，终于解决了这个问题。

但是，最后还是剩下了少量冲突，经检查发现是if-else的产生式的问题，上网一查发现这是一个很经典的冲突的例子。在老师给的文档《Lex And Yacc Tutorial》中的“计算器”的例子中我找到了类似的解决方案——定义一个新的if为IFX用于无ELSE的if语句，然后通过设置这个IFX的优先级低于ELSE即可在这个冲突的地方按我们需要的情况进行分析。

后来，我扩充了表达式的产生式，将C语言教材上的运算符优先级表的多有运算符按优先级和结合性添加了进来，结果对于三元操作符“?:”总是拿不准。当时觉得问号和冒号是分开的，不知道怎么给这个操作符整体定义优先级和结合性，于是就没做相关工作，直接在产生式中添加了这个操作符，结果一下多出了77个shift/reduce冲突，后来只是简单地将问号和冒号分别在对应的优先级的位置声明就解决了这个问题。

之后，我又想到了左值表达式和右值表达式的问题，一开始想直接通过语法来解决，结果不仅产生了大量冲突（主要集中在跟赋值语句有关的产生式），而且事实上很多表达式无法在定义语法时判断是否为左值，必须等到分析时甚至到程序执行时才能确定，因为存在可能是左值也可能是右值的表达式，比如间访操作符‘\*’，不过后来才想起来其实GCC等C语言编译器也不能处理这种问题）。但是对于有const修饰的变量、数组、struct等，仍然有些麻烦。于是放弃了在语法规则中确定左值右值的问题的方案，改为将类型定义成语法树中节点的属性，在语义分析中检查相关错误。

在编写变量定义语句的语法的时候，我想让这个语法能识别指针，结果发现如果不将变量类型做区分的话，会出现一些不符合C语言规范的问题，比如：

（1）char \*s, c, \*\*s; 和 （2）char \*f(void);

变量定义时‘\*’是在每个变量名之前的，而不是所有变量均为最左侧声明的类型，如果不将（1）和（2）的使用的“类型”识别语法作区分，将会产生语义上的错误，而如果进行区分，则会在扫描完“char”（或者“char\*”？）之后发生shift/reduce冲突，目前我采用了不作区分的设计。但是后来，我发现在函数声明时下面的语法也是成立的：

int \*f(void),\*\*g(void);

这似乎意味着‘\*’在各种情况下是跟着其后的标识符的，而不是和前面的类型在一起，于是我用一个name来表示零个或若干个‘\*’加上一个标识符，将类型统一成简单类型（包括原子类型和用户定义类型如struct），于是解决了这个问题。但是，我暂时没有在语法中支持上面的语句的识别，因为上面的声明方式我几乎从来没有见过，相信有也是一种糟糕的代码风格。

在处理错误检测时，我参考了《Lex And Yacc Tutorial》中的在相关位置加上

| error ';'

| error '}'

的方法，结果发现在行尾缺少分号的错误会多出一些报错。后来想，一直清空到换行符似乎解决了这个问题。结果又发现在我的规则里面flex根本没有把换行符返回给bison过……于是有定义遇到换行符会返回，结果就第二行就报错了，因为实际上没有任何语法规则跟换行符有关。最后一想其实真正的编译器也经常在不着边际的地方报错，于是就没有再纠结这个问题。

在显示行号的时候，由于总是在分析完整个语句之后才调用产生式之后的C语句，此时记录行号的全局变量已经随之改变，对于有多行的语句如for循环，显示出来的行号总是在结束的‘}’所在行。后来，我通过调整C语句的位置使大多数语句可以在出现的第一行被识别，但是if-else语句是个例外。因为if-else语句本身存在冲突，所以无法在遇到else之前确定状态，所以也无法在遇到if之后就输出于是，if-else语句被识别后显示的行号是该语句的最后一行。

后来，我在处理用户自定义的变量类型时又遇到了冲突——假设"UserDefinedType"是一个用户通过typedef自定义的类型，“a”、“b”是两个变量。假设不做语义分析，三者均只能被识别成标识符，那么，"UserDefinedType \*a"和"a\*b"都是 “标识符 '\*' 标识符”，而前者应该被规约成变量定义，后者应该被规约成一个表达式！在对第一个标识符进行归约时，到底把它规约成一个类型还是一个表达式产生了冲突，而且，向后看一个字符都是‘\*’，还是冲突！

当然，一开始我还找不到冲突到底是那些规则产生的。后来在群里发问，有同学回复了输出调试信息的方法，通过加“-v”选项，输出警告、冲突、规则编号、状态及对应的操作到yacc.output文件之后，我很轻松地找到了发生冲突的产生式，于是发现了上一段中提到的问题。

# 参考文献

[1] Tom Niemann .《Lex And Yacc Tutorial》

[2] URL: <http://www.gnu.org/software/bison/manual/html_node/index.html>

[3] URL: <https://www.ibm.com/developerworks/cn/linux/sdk/lex/>