**电子科技大学 信息与软件工程 学院**

**标 准 实 验 报 告**

**（实验）课程名称 编译技术**

**电子科技大学教务处制表**

**电 子 科 技 大 学**

**实 验 报 告**

**组队排序1 学生姓名：王乐卿 学 号：2018091602007**

**指导教师：周尔强**

**实验地点：信软楼304/304 实验时间：2019.10.26**

**一、实验室名称：信软 303**

**二、实验项目名称：递归下降语法分析**

**三、实验学时：4学时**

**四、实验内容及步骤：**

**1. 学习所提供的“表达式文法”的递归下降处理**

**理解 lex.l、rdparser.c的内容**

**在eclipse中建立工程并调试运行**

**2. 学习rdgram.txt所提供的文法**

**与词法分析所提供的文法作比较**

**3. 编写rdgram所提供文法的递归下降程序**

**(1)编写不生成“语法树”的递归下降程序 rdcheck.c**

**(2)将rdcheck.c改造为生成语法树的递归下降程序rdparser.c**

**(3)改进 词法分析程序、showAst函数、main函数等，使递归下降程序rdparser最终从命令行读取要分析的程序test.c,分析后调用showAst打印该程序的结构。**

**c4\_lex.l的代码：**

%{

enum yytokentype{

NUMBER=258,STRING,ID,

INT,STR,VOID,IF,ELSE,WHILE,RETURN,PRINT,SCAN,

CMP,

ASSIGN,

FEND

};

int yylval;

%}

%x comment string

%%

\/\/.+ {}

[ \t\n\r\f\v]+ {}

"#".+ {}

"int" {return INT;}

"str" {return STR;}

"void" {return VOID;}

"if" {return IF;}

"else" {return ELSE;}

"while" {return WHILE;}

"return" {return RETURN;}

"print" {return PRINT;}

"scan" {return SCAN;}

"=="|"!="|">="|"<="|"<"|">" {return CMP;}

"+="|"-="|"\*="|"/="|"%=" {return ASSIGN;}

\+ {return '+';}

\- {return '-';}

\\* {return '\*';}

\/ {return '/';}

"%" {return '%';}

";" {return ';';}

"," {return ',';}

"{" {return '{';}

"}" {return '}';}

"(" {return '(';}

")" {return ')';}

"[" {return '[';}

"]" {return ']';}

"=" {return '=';}

<<EOF>> {return FEND;}

[0-9]\* {yylval = atoi(yytext);return NUMBER;}

"/\*" BEGIN(comment);

<comment>[^\*]\* {}

<comment>"\*"+[^\*/]\* {}

<comment>\\*\/ BEGIN(INITIAL);

\" BEGIN(string);

<string>[^"]\* {return STRING;}

<string>\" BEGIN(INITIAL);

([\_a-zA-Z][a-z\_A-Z0-9]\*) {return ID;}

%%

int yyerror(char \*s)

{

fprintf(stderr, "error: %s\n", s);

return 0;

}

int yywrap()

{

return 1;

}

**rdcheck.c的代码：**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

enum yytokentype {

NUMBER = 258, STRING, ID,

INT, STR, VOID, IF, ELSE, WHILE, RETURN, PRINT, SCAN,

CMP, //< > = >= <= !=

ASSIGN, //+= -= \*= /=

FEND //end of file

};

// if match(),then advance()

extern int yylex();

extern int yylval;

extern char\* yytext;

extern FILE\* yyin;

int tok;

void advance();

bool match(int);

int program();

int external\_declaration();

int decl\_or\_stmt();

int declarator\_list();

int intstr\_list();

int initializer();

int declarator();

int parameter\_list();

int parameter();

int type();

int statement();

int statement\_list();

int expression\_statement();

int expr();

int cmp\_expr();

int add\_expr();

int mul\_expr();

int primary\_expr();

int expr\_list();

int id\_list();

void advance()

{

tok = yylex();

printf("tok: %s\n", yytext);

}

bool match(int terminal\_symbol)

{

if (tok == terminal\_symbol)

return true;

else

return false;

}

int program()

{

external\_declaration();

while (tok != FEND) {

external\_declaration();

}

return 1;

}

int external\_declaration()

{

if (match(INT) || match(STR) || match(VOID)) //FIRST

type();

else

{

printf("ERROR:external\_declaration()\n");

return 0;

}

if (match(ID))

declarator();

else

{

printf("ERROR:external\_declaration()\n");

return 0;

}

if (match('{') || match(',') || match(';'))

decl\_or\_stmt();

else

{

printf("ERROR:external\_declaration()\n");

return 0;

}

return 1;

}

int decl\_or\_stmt()

{

if (match('{'))

{

advance();

if (match('}'))

{

advance();

return 1;

} //statement' first

else if(match(SCAN)||match(PRINT)||match(RETURN)||match(WHILE)||match(IF)||match('{')||match(VOID)||match(STR)||match(INT)||match(';')||match('(')||match(ID)||match(STRING)||match(NUMBER)||match('-'))

{

statement\_list();

if (!match('}'))

{

printf("ERROR:decl\_or\_stmt()1\n");

return 0;

}

advance();

return 1;

}

else

{

printf("ERROR:decl\_or\_stmt()2\n");

return 0;

}

}

else if (match(','))

{

advance();

if (match(ID))

{

declarator\_list();

if (!match(';'))

{

printf("ERROR:decl\_or\_stmt()3\n");

return 0;

}

advance();

return 1;

}

else

{

printf("ERROR:decl\_or\_stmt()4\n");

return 0;

}

}

else if (match(';'))

{

advance();

return 1;

}

else

{

printf("ERROR:decl\_or\_stmt5\n");

return 0;

}

}

int declarator\_list()

{

if (match(ID))

{

declarator();

while (match(','))

{

advance();

if (match(ID))

{

declarator();

}

else

{

printf("ERROR: declarator\_list()\n");

return 0;

}

}

return 1;

}

else

{

printf("ERROR: declarator\_list()\n");

return 0;

}

}

int intstr\_list()

{

if (match(NUMBER)||match(STRING))

{

initializer();

while (match(','))

{

advance();

if (match(NUMBER) || match(STRING))

{

initializer();

}

else

{

printf("ERROR: intstr\_list()\n");

return 0;

}

}

return 1;

}

else

{

printf("ERROR: intstr\_list()\n");

return 0;

}

}

int initializer()

{

if (match(NUMBER))

{

advance();

return 1;

}

else if (match(STRING))

{

advance();

return 1;

}

else

{

printf("ERROR:False Match initializer:Expect NUMBER or STRING\n");

return 0;

}

}

int declarator()

{

if (!match(ID))

{

printf("ERROR:False Match declarator:Expect ID\n");

return 0;

}

else

{

advance();

if (match('='))

{

advance();

if (match(ID) || match('(') || match(NUMBER) || match(STRING) || match('-'))

{

expr();

return 1;

}

else

{

printf("ERROR:declarator()\n");

return 0;

}

}

else if (match('('))

{

advance();

if (match(')'))

{

advance();

return 1;

}

else if(match(INT)||match(STR)||match(VOID))

{

parameter\_list();

if (match(')'))

{

advance();

return 1;

}

else

{

printf("ERROR:declarator()\n");

return 0;

}

}

else

{

printf("ERROR:declarator()\n");

return 0;

}

}

else if (match('['))

{

advance();

if (match(']'))

{

advance();

if (match('='))

{

advance();

if (match('{'))

{

advance();

if (match(NUMBER) || match(STRING))

{

intstr\_list();

if (match('}'))

{

advance();

return 1;

}

else

{

printf("ERROR:declarator()\n");

return 0;

}

}

else

{

printf("ERROR:declarator()\n");

return 0;

}

}

else

{

printf("ERROR:declarator()\n");

return 0;

}

}

return 1; //没匹配到=，说明是匹配了语句 ID'[' ']'

}

else if (match(ID) || match('(') || match(NUMBER) || match(STRING) || match('-'))

{

expr();

if (match(']'))

{

advance();

if (match('='))

{

advance();

if (match('{'))

{

advance();

if (match(NUMBER) || match(STRING))

{

intstr\_list();

if (match('}'))

{

advance();

return 1;

}

else

{

printf("ERROR:declarator()\n");

return 0;

}

}

else

{

printf("ERROR:declarator()\n");

return 0;

}

}

else

{

printf("ERROR:declarator()\n");

return 0;

}

}

return 1; //没匹配到=，说明是匹配了语句 ID'['expr() ']'

}

else

{

printf("ERROR:declarator()\n");

return 0;

}

}

else

{

printf("ERROR:declarator()\n");

return 0;

}

}

else

{

return 1;

//表示只匹配了ID

}

}

}

int parameter\_list()

{

if (match(INT) || match(STR)||match(VOID))

{

parameter();

while (match(','))

{

advance();

if (match(INT) || match(STR) || match(VOID))

{

parameter();

}

else

{

printf("ERROR:parameter\_list()\n");

return 0;

}

}

return 1;

}

else

{

printf("ERROR:parameter\_list())\n");

return 0;

}

}

int parameter()

{

if (match(INT) || match(STR) || match(VOID))

{

type();

if (!match(ID))

{

printf("ERROR:parameter()\n");

return 0;

}

advance();

return 1;

}

else

{

printf("ERROR:parameter()\n");

return 0;

}

}

int type()

{

if (match(INT))

{

advance();

return 1;

}

else if (match(STR))

{

advance();

return 1;

}

else if (match(VOID))

{

advance();

return 1;

}

else

{

printf("ERROR:type():Expect INT or STR or VOID\n");

return 0;

}

}

int statement()

{

if (match(INT) || match(STR) || match(VOID))

{

type();

if (match(ID))

{

declarator\_list();

if (match(';'))

{

advance();

return 1;

}

else

{

printf("ERROR:statement()1\n");

return 0;

}

}

else

{

printf("ERROR:statement()2\n");

return 0;

}

}

else if (match('{'))

{

advance();

if (match(SCAN) || match(PRINT) || match(RETURN) || match(WHILE)||match(IF) || match('{') || match(VOID) || match(STR) || match(INT) || match(';') || match('(') || match(ID) || match(STRING) || match(NUMBER) || match('-'))

{

statement\_list();

if (match('}'))

{

advance();

return 1;

}

else

{

printf("ERROR:statement()3\n");

return 0;

}

}

else

{

printf("ERROR:statement()4\n");

return 0;

}

}

else if (match(';') || match('-') || match(ID) || match(NUMBER) || match(STRING))

{

expression\_statement();

return 1;

}

else if (match(IF))

{

advance();

if (match('('))

{

advance();

if (match('-') || match(ID) || match(NUMBER) || match(STRING)||match('('))

{

expr();

if (match(')'))

{

advance();

if (match(SCAN) || match(PRINT) || match(RETURN)||match(WHILE) || match(IF) || match('{') || match(VOID) || match(STR) || match(INT) || match(';') || match('(') || match(ID) || match(STRING) || match(NUMBER) || match('-'))

{

statement();

if (match(ELSE))

{

advance();

if (match(SCAN) || match(PRINT) || match(RETURN) ||match(WHILE)|| match(IF) || match('{') || match(VOID) || match(STR) || match(INT) || match(';') || match('(') || match(ID) || match(STRING) || match(NUMBER) || match('-'))

{

statement();

return 1;

}

else

{

printf("ERROR:statement()5\n");

return 0;

}

}

else

{

return 1; //IF '(' expr ')' statement

}

}

else

{

printf("ERROR:statement()6\n");

return 0;

}

}

else

{

printf("ERROR:statement()7\n");

return 0;

}

}

else

{

printf("ERROR:statement()8\n");

return 0;

}

}

else

{

printf("ERROR:statement()9\n");

return 0;

}

}

else if (match(WHILE))

{

advance();

if (match('('))

{

advance();

if (match('-') || match(ID) || match(NUMBER) || match(STRING)||match('('))

{

printf("3\n");

expr();

if (match(')'))

{

advance();

if (match(SCAN) || match(PRINT) || match(RETURN) ||match(WHILE)|| match(IF) || match('{') || match(VOID) || match(STR) || match(INT) || match(';') || match('(') || match(ID) || match(STRING) || match(NUMBER) || match('-'))

{

statement();

return 1;

}

else

{

printf("ERROR:statement()10\n");

return 0;

}

}

else

{

printf("ERROR:statement()11\n");

return 0;

}

}

else

{

printf("ERROR:statement()12\n");

return 0;

}

}

else

{

printf("ERROR:statement()13\n");

return 0;

}

}

else if (match(RETURN))

{

advance();

if (match(';'))

{

advance();

return 1;

}

else if (match('-') || match(ID) || match(NUMBER) || match(STRING)|| match('('))

{

expr();

if (match(';'))

{

advance();

return 1;

}

else

{

printf("ERROR:statement()14\n");

return 0;

}

}

else

{

printf("ERROR:statement()15\n");

return 0;

}

}

else if (match(PRINT))

{

advance();

if (match(';'))

{

advance();

return 1;

}

else if (match('-') || match(ID) || match(NUMBER) || match(STRING) || match('('))

{

expr\_list();

if (match(';'))

{

advance();

return 1;

}

else

{

printf("ERROR:statement()16\n");

return 0;

}

}

else

{

printf("ERROR:statement()17\n");

return 0;

}

}

else if (match(SCAN))

{

advance();

if (match(ID))

{

id\_list();

if (match(';'))

{

advance();

return 1;

}

else

{

printf("ERROR:statement()18\n");

return 0;

}

}

else

{

printf("ERROR:statement()19\n");

return 0;

}

}

else

{

printf("ERROR:statement()20\n");

return 0;

}

}

int statement\_list()

{

if (match(SCAN) || match(PRINT)|| match(WHILE)|| match(RETURN) || match(IF) || match('{') || match(VOID) || match(STR) || match(INT) || match(';') || match('(') || match(ID) || match(STRING) || match(NUMBER) || match('-'))

{

statement();

while (match(SCAN) || match(PRINT) || match(RETURN) ||match(WHILE)|| match(IF) || match('{') || match(VOID) || match(STR) || match(INT) || match(';') || match('(') || match(ID) || match(STRING) || match(NUMBER) || match('-'))

{

statement();

}

return 1;

}

else

{

printf("ERROR statement\_list()\n");

return 0;

}

}

int expression\_statement()

{

if (match(';'))

{

advance();

return 1;

}

else if (match('-') || match(ID) || match(NUMBER) || match(STRING) || match('('))

{

expr();

if (match(';'))

{ advance();

return 1;

}

else

{

printf("ERROR:expression\_statement()1\n");

return 0;

}

}

else

{

printf("ERROR:expression\_statement()2\n");

return 0;

}

}

int expr()

{

if (match('-') || match(ID) || match(NUMBER) || match(STRING) || match('('))

{

cmp\_expr();

}

else

{

printf("ERROR:expr()\n");

return 0;

}

}

int cmp\_expr()

{

if (match('-') || match(ID) || match(NUMBER) || match(STRING) || match('('))

{

add\_expr();

while(match(CMP))

{

advance();

if (match('-') || match(ID) || match(NUMBER) || match(STRING) || match('('))

{

cmp\_expr();

}

}

return 1;

}

else

{

printf("ERROR: cmp\_expr()\n");

return 0;

}

}

int add\_expr()

{

if (match('-') || match(ID) || match(NUMBER) || match(STRING) || match('('))

{

mul\_expr();

while(match('+')||match('-'))

{

advance();

if (match('-') || match(ID) || match(NUMBER) || match(STRING) || match('('))

{

add\_expr();

}

}

}

else

{

printf("ERROR: add\_expr()\n");

return 0;

}

}

int mul\_expr()

{

if (match(ID) || match(NUMBER) || match(STRING) || match('('))

{

primary\_expr();

while(match('\*') || match('/') || match('%'))

{

advance();

if ( match(ID) || match(NUMBER) || match(STRING) || match('('))

{

mul\_expr();

}

}

return 1;

}

else if(match('-'))

{

advance();

if (match(ID) || match(NUMBER) || match(STRING) || match('('))

{

return primary\_expr();

}

else

{

printf("ERROR: mul\_expr()\n");

return 0;

}

}

else

{

printf("ERROR: mul\_expr()\n");

return 0;

}

}

int primary\_expr()

{

if (match(ID))

{

advance();

if (match('('))

{

advance();

if (match(')'))

{

advance();

return 1;

}

else if (match('-') || match(ID) || match(NUMBER) || match(STRING) || match('('))

{

if (expr\_list())

{

if (match(')'))

{

advance();

return 1;

}

else

{

printf("ERROR:primary\_expr\n");

return 0;

}

}

else

{

printf("ERROR:primary\_expr\n");

return 0;

}

}

else

{

printf("ERROR:primary\_expr\n");

return 0;

}

}

else if (match('['))

{

advance();

if (match('-') || match(ID) || match(NUMBER) || match(STRING) || match('('))

{

if (expr())

{

if (match(']'))

{

advance();

if (match('='))

{

advance();

if (match('-') || match(ID) || match(NUMBER) || match(STRING) || match('('))

{

if (expr())

{

return 1;

}

else

{

printf("ERROR:primary\_expr\n");

return 0;

}

}

else

{

printf("ERROR:primary\_expr\n");

return 0;

}

}

else

{

return 1;

}

}

else

{

printf("ERROR:primary\_expr\n");

return 0;

}

}

else

{

printf("ERROR:primary\_expr\n");

return 0;

}

}

else

{

printf("ERROR:primary\_expr\n");

return 0;

}

}

else if (match('='))

{

advance();

if (match('-') || match(ID) || match(NUMBER) || match(STRING) || match('('))

{

if (expr())

{

return 1;

}

else

{

printf("ERROR:primary\_expr\n");

return 0;

}

}

else

{

printf("ERROR:primary\_expr\n");

return 0;

}

}

else if (match(ASSIGN))

{

advance();

if (match('-') || match(ID) || match(NUMBER) || match(STRING) || match('('))

{

expr();

return 1;

}

else

{

printf("ERROR:primary\_expr\n");

return 0;

}

}

else

{

return 1;

}

}

else if (match('('))

{

advance();

if (match('-') || match(ID) || match(NUMBER) || match(STRING) || match('('))

{

if (expr())

{

if (match(')'))

{

advance();

return 1;

}

else

{

return 0;

}

}

else

{

return 0;

}

}

else

{

return 0;

}

}

else if (match(NUMBER))

{

advance();

return 1;

}

else if (match(STRING))

{

advance();

return 1;

}

else

{

printf("ERROR:primary\_expr\n");

return 0;

}

}

int expr\_list()

{

if (match('-') || match(ID) || match(NUMBER) || match(STRING) || match('('))

{

expr();

while (match(','))

{

advance();

if (match('-') || match(ID) || match(NUMBER) || match(STRING) || match('('))

{

expr\_list();

}

else

{

printf("ERROR:Fail Match expr\_list\n");

return 0;

}

}

return 1;

}

else

{

printf("ERROR:Fail Match expr\_list\n");

return 0;

}

}

int id\_list() //ok

{

if (!match(ID))

{

printf("ERROR:Fail Match id\_list\n");

return 0;

}

else

{

advance();

while (match(','))

{

advance();

id\_list();

}

return 1;

}

}

int main(int argc, char \*\*argv)

{

setbuf(stdout, NULL);

if(argc < 2 )

{

printf("input file is needed.\n");

exit(0);

}

yyin = fopen(argv[1],"r");

advance();

program();

printf("rdcheck succeed!\n");

return 0;

}

**rdparser.c的代码：**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

enum yytokentype {

NUMBER = 258, STRING, ID,

INT, STR, VOID, IF, ELSE, WHILE, RETURN, PRINT, SCAN,

CMP, //< > = >= <= !=

ASSIGN, //+= -= \*= /=

FEND //end of file

};

typedef struct \_ast ast;

typedef struct \_ast\* past;

struct \_ast {

int ivalue; //保存整数信息

char\* svalue; //保存字符串信息

char\* nodeType; //结点类型

past left; //指向左子树

past right; //指向右子树

};

extern int yylex();

extern int yylval;

extern char\* yytext;

extern FILE\* yyin;

int tok;

void advance();

bool match(int);

past program();

past external\_declaration();

past decl\_or\_stmt();

past declarator\_list();

past intstr\_list();

past initializer();

past declarator();

past parameter\_list();

past parameter();

past type();

past statement();

past statement\_list();

past expression\_statement();

past expr();

past cmp\_expr();

past add\_expr();

past mul\_expr();

past primary\_expr();

past expr\_list();

past id\_list();

void advance()

{

tok = yylex();

printf("tokval: %s\n", yytext);

}

bool match(int terminal\_symbol)

{

if (tok == terminal\_symbol)

return true;

else

return false;

}

past newAstNode();

void showAst(past node, int nest);

past newList\_nofinal(past);

past newList\_final(past);

past setnull(past);

//创建一个空树结点

past newAstNode()

{

past node = (past)malloc(sizeof(ast));

if (node == NULL)

{

printf("run out of memory.\n");

exit(0);

}

memset(node, 0, sizeof(ast));

//memset()函数原型是extern void \*memset(void \*buffer, int c, int count)

//buffer：为指针或是数组

//c：是赋给buffer的值

//count：是buffer的长度.

//这个函数在socket中多用于清空数组.如:原型是memset(buffer, 0, sizeof(buffer))

return node;

}

//输出树

void showAst(past l,int j)

{

int a=1;

if(l->nodeType)

{

int i = 0;

a=0;

for(i; i<j; i++)

{

printf(" ");

}

if(strcmp(l->nodeType, "intValue") == 0)

{

printf("%s %d\n", l->nodeType, l->ivalue);

}

else if(strcmp(l->nodeType, "strValue") == 0)

{

printf("%s %s\n", l->nodeType, l->svalue);

}

else if(strcmp(l->nodeType, "idValue") == 0)

{

printf("%s %s\n", l->nodeType, l->svalue);

}

else if(strcmp(l->nodeType, "finalValue") == 0)

{

printf("%s %d\n", l->nodeType, l->ivalue);

}

else

{

printf("%s\n",l->nodeType);

}

}

if(l->left)

{

showAst(l->left,j+1-a);

}

if(l->right)

{

showAst(l->right,j+1-a);

}

return ;

}

past setnull(past buf)//将叶子结点的后续指针赋空值

{

buf->left = NULL;

buf->right = NULL;

return buf;

}

past newList\_nofinal(past l)

{

l->right = newAstNode();

l = l->right;

return l;

}

past newList\_final(past l)

{

l->left = newAstNode();

l->left->nodeType = (char\*)malloc(20);

strcpy(l->left->nodeType, yytext);

if (match(NUMBER))

{

strcpy(l->left->nodeType, "intValue");

l->left->ivalue = yylval;

}

if (match(STRING))

{

strcpy(l->left->nodeType, "strValue");

l->left->svalue = (char\*)malloc(50);

strcpy(l->left->svalue, yytext);

}

if(match(ID))

{

strcpy(l->left->nodeType, "idValue");

l->left->svalue = (char\*)malloc(50);

strcpy(l->left->svalue, yytext);

}

if(match(CMP)||match(ASSIGN)||match('+')||match('-')||match('\*')||match('/')||match('%')||match(';')||match(',')||match('{')||match('}')||match('(')||match(')')||match('[')||match(']')||match('='))

{

strcpy(l->left->nodeType, "finalValue");

l->left->ivalue = tok;

}

return newList\_nofinal(l);

}

past program()

{

past l = newAstNode();

past buf = l;

//buf指向l所指向的，l始终指向第一个结点，操作则对buf进行

l->nodeType = "program";

while (!match(FEND))

{

buf->left = external\_declaration();

buf=newList\_nofinal(buf);

}

buf = setnull(buf);

return l;

}

past external\_declaration()

{

past l = newAstNode();

past buf = l;

buf->nodeType = "external\_declaration";

buf->left= type();

buf = newList\_nofinal(buf);

buf->left = declarator();

buf = newList\_nofinal(buf);

buf->left = decl\_or\_stmt();

buf = newList\_nofinal(buf);

buf = setnull(buf);

return l;

}

past decl\_or\_stmt()

{

past l = newAstNode();

past buf = l;

buf->nodeType = "decl\_or\_stmt";

if (match('{'))

{

buf = newList\_final(buf);

advance();

if (match('}'))

{

buf = newList\_final(buf);

advance();

}

else

{

buf->left = statement\_list();

buf = newList\_nofinal(buf);

if (match('}'))

{

buf = newList\_final(buf);

advance();

}

else

{

printf(" ERROR:decl\_or\_stmt()");

}

}

}

else if (match(','))

{

buf = newList\_final(buf);

advance();

buf->left =declarator\_list();

buf = newList\_nofinal(buf);

if (match(';'))

{

buf = newList\_final(buf);

advance();

}

else

{

printf(" ERROR:decl\_or\_stmt()");

}

}

else if (match(';'))

{

buf = newList\_final(buf);

advance();

}

else

{

printf(" ERROR:decl\_or\_stmt()");

}

buf = setnull(buf);

return l;

}

past declarator\_list()

{

past l = newAstNode();

past buf = l;

buf->nodeType = "declarator\_list";

buf->left = declarator();

buf = newList\_nofinal(buf);

while (match(','))

{

buf = newList\_final(buf);

advance();

buf->left = declarator\_list();

buf = newList\_nofinal(buf);

}

buf = setnull(buf);

return l;

}

past intstr\_list()

{

past l = newAstNode();

past buf = l;

buf->nodeType = "intstr\_list";

buf->left = initializer();

buf = newList\_nofinal(buf);

while (match(','))

{

buf = newList\_final(buf);

advance();

buf->left = intstr\_list();

buf = newList\_nofinal(buf);

}

buf = setnull(buf);

return l;

}

past initializer()

{

past l = newAstNode();

past buf = l;

buf->nodeType = "initializer";

if (!match(NUMBER)&&!match(STRING))

{

printf("Error:initializer():expect NUMBER or STRING\n");

}

buf = newList\_final(buf);

buf = setnull(buf);

advance();

return l;

}

past declarator()

{

past l = newAstNode();

past buf = l;

buf->nodeType = "declarator";

if (!match(ID))

{

printf("Error:declarator():expect ID\n");

}

buf = newList\_final(buf);

advance();

if (match('='))

{

buf = newList\_final(buf);

advance();

buf->left = expr();

buf = newList\_nofinal(buf);

}

else if (match('('))

{

buf = newList\_final(buf);

advance();

if (match(')'))

{

buf = newList\_final(buf);

advance();

}

else

{

buf->left = parameter\_list();

buf = newList\_nofinal(buf);

if (!match(')'))

{

printf("Error:declarator():expect )\n");

}

buf = newList\_final(buf);

advance();

}

}

else if (match('['))

{

buf = newList\_final(buf);

advance();

if (match(']'))

{

buf = newList\_final(buf);

advance();

}

else

{

buf->left = expr();

buf = newList\_nofinal(buf);

if (!match(']'))

{

printf("Error:declarator():expect ]\n");

}

buf = newList\_final(buf);

advance();

}

if (match('='))

{

buf = newList\_final(buf);

advance();

if (!match('{'))

{

printf("Error:declarator():expect {\n");

}

buf = newList\_final(buf);

advance();

buf->left = intstr\_list();

buf = newList\_nofinal(buf);

if (!match('}'))

{

printf("Error:declarator():expect }\n");

}

buf = newList\_final(buf);

advance();

}

}

buf = setnull(buf);

return l;

}

past parameter\_list()

{

past l = newAstNode();

past buf = l;

buf->nodeType = "parameter\_list";

buf->left = parameter();

buf = newList\_nofinal(buf);

while (match(','))

{

buf = newList\_final(buf);

advance();

buf->left = parameter\_list();

buf = newList\_nofinal(buf);

}

buf = setnull(buf);

return l;

}

past parameter()

{

past l = newAstNode();

past buf = l;

buf->nodeType = "parameter";

buf->left = type();

buf = newList\_nofinal(buf);

if (!match(ID))

{

printf("Error:parameter():expect ID\n");

}

buf = newList\_final(buf);

advance();

buf = setnull(buf);

return l;

}

past type()

{

past l = newAstNode();

past buf = l;

l->nodeType = "type";

if (!match(INT)&& !match(STR)&&!match(VOID))

{

printf("Error:type():expect int or str or past\n");

}

buf = newList\_final(buf);

advance();

buf = setnull(buf);

return l;

}

past statement()

{

past l = newAstNode();

past buf = l;

buf->nodeType = "statement";

if (match(SCAN))

{

buf = newList\_final(buf);

advance();

buf->left = id\_list();

buf = newList\_nofinal(buf);

if (!match(';'))

{

printf("Error,expect ;\n");

}

buf = newList\_final(buf);

advance();

}

else if (match(PRINT))

{

buf = newList\_final(buf);

advance();

if (match(';'))

{

buf = newList\_final(buf);

advance();

}

else

{

buf->left = expr\_list();

buf = newList\_nofinal(buf);

if (!match(';'))

{

printf("Error,expect ;\n");

}

buf = newList\_final(buf);

advance();

}

}

else if (match(RETURN))

{

buf = newList\_final(buf);

advance();

if (match(';'))

{

buf = newList\_final(buf);

advance();

}

else

{

buf->left = expr();

buf = newList\_nofinal(buf);

if (!match(';'))

{

printf("Error,expect ;\n");

}

buf = newList\_final(buf);

advance();

}

}

else if (match(WHILE))

{

buf = newList\_final(buf);

advance();

if (!match('('))

{

printf("Error,expect (\n");

}

buf = newList\_final(buf);

advance();

buf->left = expr();

buf = newList\_nofinal(buf);

if (!match(')'))

{

printf("Error,expect )\n");

}

buf = newList\_final(buf);

advance();

buf->left = statement();

buf = newList\_nofinal(buf);

}

else if (match(IF))

{

buf = newList\_final(buf);

advance();

if (!match('('))

{

printf("Error,expect (\n");

}

buf = newList\_final(buf);

advance();

buf->left = expr();

buf = newList\_nofinal(buf);

if (!match(')'))

{

printf("Error,expect )");

}

buf = newList\_final(buf);

advance();

buf->left = statement();

buf = newList\_nofinal(buf);

if (match(ELSE))

{

buf = newList\_final(buf);

advance();

buf->left = statement();

buf = newList\_nofinal(buf);

}

}

else if (match('{'))

{

buf = newList\_final(buf);

advance();

buf->left = statement\_list();

buf = newList\_nofinal(buf);

if (!match('}'))

{

printf("Error,expect }\n");

}

buf = newList\_final(buf);

advance();

}

else if (match(VOID)||match(STR)||match(INT))

{

buf = newList\_final(buf);

advance();

buf->left = declarator\_list();

buf = newList\_nofinal(buf);

if (!match(';'))

{

printf("Error,expect ;\n");

}

buf = newList\_final(buf);

advance();

}

else if(match(';') || match('-') || match(ID) || match(NUMBER) || match(STRING))

{

buf->left = expression\_statement();

buf = newList\_nofinal(buf);

}

buf = setnull(buf);

return l;

}

past statement\_list()

{

past l = newAstNode();

past buf = l;

buf->nodeType = "statement\_list";

while(match(SCAN) || match(PRINT) || match(WHILE) || match(RETURN) || match(IF) || match('{') || match(VOID) || match(STR) || match(INT) || match(';') || match('(') || match(ID) || match(STRING) || match(NUMBER) || match('-'))

{

buf->left = statement();

buf = newList\_nofinal(buf);

}

buf = setnull(buf);

return l;

}

past expression\_statement()

{

past l = newAstNode();

past buf = l;

buf->nodeType = "expression\_statement";

if (match(';'))

{

buf = newList\_final(buf);

advance();

}

else

{

buf->left = expr();

buf = newList\_nofinal(buf);

if (!match(';'))

{

printf("Error,expect ;\n");

}

buf = newList\_final(buf);

advance();

}

buf = setnull(buf);

return l;

}

past expr()

{

past l = newAstNode();

past buf = l;

buf->nodeType = "expr";

buf->left = cmp\_expr();

buf = newList\_nofinal(buf);

buf = setnull(buf);

return l;

}

past cmp\_expr()

{

past l = newAstNode();

past buf = l;

buf->nodeType = "cmp\_expr";

buf->left = add\_expr();

buf = newList\_nofinal(buf);

while (match(CMP))

{

buf = newList\_final(buf);

advance();

buf->left = cmp\_expr();

buf = newList\_nofinal(buf);

}

buf = setnull(buf);

return l;

}

past add\_expr()

{

past l = newAstNode();

past buf = l;

buf->nodeType = "add\_expr";

buf->left = mul\_expr();

buf = newList\_nofinal(buf);

while (match('+')||match('-'))

{

buf = newList\_final(buf);

advance();

buf->left = add\_expr();

buf = newList\_nofinal(buf);

}

buf = setnull(buf);

return l;

}

past mul\_expr()

{

past l = newAstNode();

past buf = l;

buf->nodeType = "mul\_expr";

if (match('-'))

{

buf = newList\_final(buf);

advance();

buf->left = mul\_expr();

buf = newList\_nofinal(buf);

}

else

{

buf->left = primary\_expr();

buf = newList\_nofinal(buf);

while (match('\*')|| match('/') || match('%'))

{

buf = newList\_final(buf);

advance();

buf->left = mul\_expr();

buf = newList\_nofinal(buf);

}

}

buf = setnull(buf);

return l;

}

past primary\_expr()

{

past l = newAstNode();

past buf = l;

buf->nodeType = "primary\_expr";

if (match(ID))

{

buf = newList\_final(buf);

advance();

if (match('['))

{

buf = newList\_final(buf);

advance();

buf->left = expr();

buf = newList\_nofinal(buf);

if (!match(']'))

{

printf("Error,expect ]\n");

}

buf = newList\_final(buf);

advance();

if (match('='))

{

buf = newList\_final(buf);

advance();

buf->left = expr();

buf = newList\_nofinal(buf);

}

}

else if (match('='))

{

buf = newList\_final(buf);

advance();

buf->left = expr();

buf = newList\_nofinal(buf);

}

else if (match('('))

{

buf = newList\_final(buf);

advance();

if (match(')'))

{

buf = newList\_final(buf);

advance();

}

else

{

buf->left = expr\_list();

buf = newList\_nofinal(buf);

if (!match(')'))

{

printf("Error,expect )\n");

}

buf = newList\_final(buf);

advance();

}

}

else if (match(ASSIGN))

{

buf = newList\_final(buf);

advance();

buf->left = expr();

buf = newList\_nofinal(buf);

}

}

else if (match('('))

{

buf = newList\_final(buf);

advance();

buf->left = expr();

buf = newList\_nofinal(buf);

if (!match(')'))

{

printf("Error,expect )\n");

}

buf = newList\_final(buf);

advance();

}

else

{

if (!match(NUMBER)&&!match(STRING))

{

printf("Error,expect NUMBER or STRING \n");

}

buf = newList\_final(buf);

advance();

}

buf = setnull(buf);

return l;

}

past expr\_list()

{

past l = newAstNode();

past buf = l;

buf->nodeType = "expr\_list";

buf->left = expr();

buf = newList\_nofinal(buf);

while (match(','))

{

buf = newList\_final(buf);

advance();

buf->left = expr\_list();

buf = newList\_nofinal(buf);

}

buf = setnull(buf);

return l;

}

past id\_list()

{

past l = newAstNode();

past buf = l;

buf->nodeType = "id\_list";

if (!match(ID))

{

printf("Error,expect ID\n");

}

buf = newList\_final(buf);

advance();

while(match(','))

{

buf = newList\_final(buf);

advance();

buf->left = id\_list();

buf = newList\_nofinal(buf);

}

buf = setnull(buf);

return l;

}

int main(int argc, char\*argv[])

{

setbuf(stdout, NULL);

if(argc < 2 )

{

printf("input file is needed.\n");

exit(0);

}

yyin = fopen(argv[1],"r");

advance();

past program\_tree=program();

showAst(program\_tree, 0);

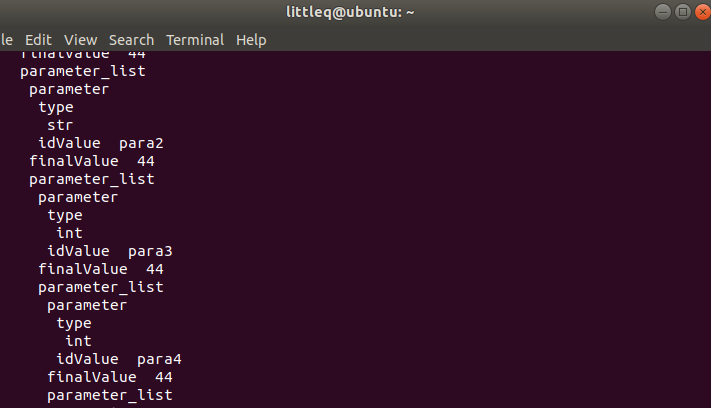
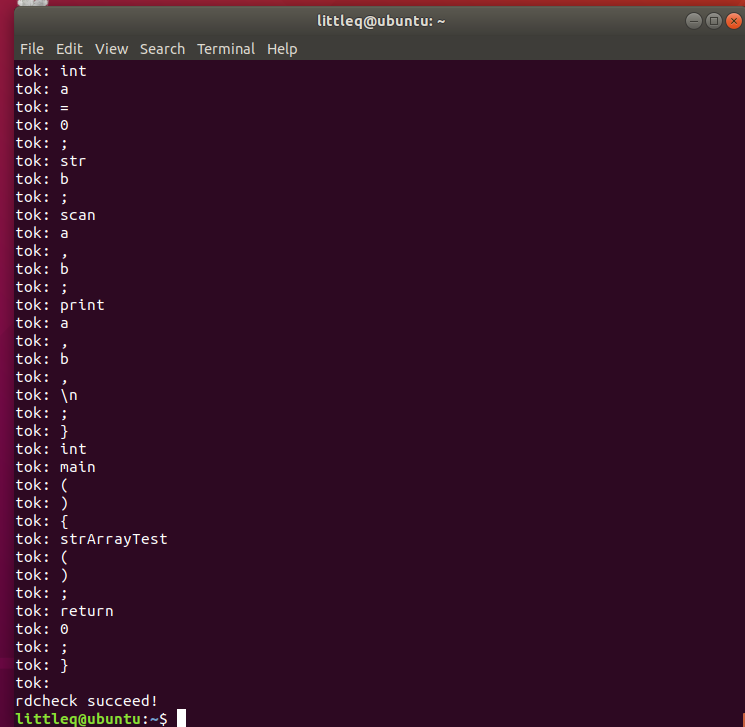
printf("rdparser succeed!\n");

fclose(yyin);

return 0;

}

**五、实验运行结果：**



**六、实验结论与总结：**

本次实验是基于第一次实验的原理，分析所给文法，编写flex.l文件得到词法分析结果，在由此进行递归向下的语法分析。实验复习了实验一的内容，使我对实验一有了更加深入的问题，同时有些实验一没有解决的问题在实验二中经过了一定时间的学习之后我也成功解决，可以说是更加的完善了我的知识。同时本次实验也有不少的新内容，一开始觉得十分的困难，但是经过不断反复的看书、看PPT、看示例，终于有了头绪，成功写出了rdcheck1.0版本，但是由于一些疏忽和理解不够，出现了致命的错误，于是我几乎重写了代码，耗时许久，我更加认识到写代码前的理解准备工作是多么的重要，准备完善可以起到事半功倍的效果。写完了不生成树的递归向下程序，又有了新的难题，即生成树的版本，这个版本书上资料很少，示例也少，则更加考验我自己的分析和解决问题能力，我在网上查询资料之后再结合所给示例，经过自己的理解完成了本次代码的rdparser.c文件，整个实验耗费了无数的时间和精力，无数次想要就此放弃，但是所幸坚持下来了，本次实验也锻炼了我的代码能力和增强了我的心理承受能力，同时我对词法分析和语法分析有了更加深入的理解，对它们的工作原理也有了更加独特深入的思考理解，虽然过程艰辛，但是收获也是颇丰。

**报告评分：**

**指导教师签字：**