

**课 程 实验 报 告**

**题目： Vector语言 编译器**

**课程名称： 编译原理实验**

**专业班级： CS1807**

**学 号： X2020I1015**

**姓 名： 李延波**

**指导教师： 杨茂林**

**报告日期： 2021年6月28日**

**计算机科学与技术学院**

**目录**

[1 概述 3](#_Toc44344119)

[2 系统描述 4](#_Toc44344120)

[2.1自定义语言概述 4](#_Toc44344121)

[2.2单词文法与语言文法 4](#_Toc44344122)

[2.3 符号表结构定义 9](#_Toc44344123)

[2.4 错误类型码定义 10](#_Toc44344124)

[2.5中间代码结构定义 11](#_Toc44344125)

[2.6目标代码指令集选择 12](#_Toc44344126)

[3 系统设计与实现 13](#_Toc44344127)

[3.1词法分析器 13](#_Toc44344128)

[3.2语法分析器 14](#_Toc44344129)

[3.3符号表管理 14](#_Toc44344130)

[3.4语义检查 15](#_Toc44344131)

[3.5报错功能 17](#_Toc44344132)

[3.6中间代码生成 17](#_Toc44344133)

[3.7汇编代码生成 23](#_Toc44344134)

[4 系统测试与评价 26](#_Toc44344135)

[4.1测试用例 26](#_Toc44344136)

[4.2正确性测试 29](#_Toc44344137)

[4.3报错功能测试 33](#_Toc44344138)

[4.4系统的优点 33](#_Toc44344139)

[4.5系统的缺点 34](#_Toc44344140)

[5实验小结 35](#_Toc44344141)

**[参考文献](#_Toc44344142)** [36](#_Toc44344142)

**[附件：源代码](#_Toc44344143)** [37](#_Toc44344143)

# 概述

本次实验是构造一个高级语言的子集编译器，目标代码是汇编语言。按照任务书，实现的方案有很多种选择。

本次实验将选择基于miniC与C的语言，将其命名为Vector, 实验的任务主要是通过对简单编译器的完整实现，加深课程中关键算法的理解，提高学生系统软件研发技术。以下是已实现的功能。

**实验一：对Vector语言的定义**

1. 数据类型包括char类型、int类型和float类型，字符串作为可选项；
2. 基本运算包括了算数运算、比较运算、自增自减运算和复合赋值运算；
3. 控制语句包括if语句，while语句和break、continue语句。
4. 多维数组。
5. 语言支持行注释和块注释。

**实验二: 词法分析器设计与实现**

1. 单词的分类和形式化描述；
2. 定义了保留字和操作符、界符的内部码；
3. 实现了一个完整的词法分析器。

**实验三: 自定义语言语法分析设计与实现**

1. 调研语法生成工具；
2. 根据实验一定义的语法规则，编写源程序；
3. 生成语法分析程序；
4. 测试语法分析器的功能。

**实验四：符号表管理**

1. 选择数组结构实现符号表；
2. 实现了创建符号表、插入表项、查询表项、修改表项、释放符号表空间等。
3. 计算语义
4. 输出抽象语法树。

**实验五：静态语义分析**

1.使用未定义的变量;

2.调用未定义或未声明的函数;

3.在同一作用域，名称的重复定义(如变量名、函数名、结构类型名以及结构体成员名等)。为更清楚说明语义错误，这里也可以拆分成几种类型的错误，如变量重复定义、函数重复定义、结构体成员名重复等;

4.对非函数名采用函数调用形式;

5.对函数名采用非函数调用形式访问;

6.函数调用时参数个数不匹配，如实参表达式个数太多、或实参表达式个数太少;

7.函数调用时实参和形参类型不匹配;

8.对非数组变量采用下标变量的形式访问;

9.数组变量的下标不是整型表达式;

10.对非结构变量采用成员选择运算符“.”;

11.结构成员不存在;

12.赋值号左边不是左值表达式;

13.对非左值表达式进行自增、自减运算;

14.对结构体变量进行自增、自减运算;

15.类型不匹配。如数组名与结构变量名间的运算，需要指出类型不匹配错误;

16.函数返回值类型与函数定义的返回值类型不匹配;

17.函数没有返回语句(当函数返回值类型不是 void 时);

18.break 语句不在循环语句或 switch 语句中;

19.continue 语句不在循环语句中;

**实验六:中间代码生成**

1. 完成可执行语句的中间代码生成。

由于时间原因，只进行到了实验六。

# 2系统描述

### 2.1自定义语言概述

采用简化的C语言的文法。其文法规则如下：

G[program]:

program → ExtDefList

ExtDefList→ExtDef ExtDefList | ε

ExtDef→Specifier ExtDecList ; |Specifier FunDec CompSt

Specifier→int | float | char

ExtDecList→VarDec | VarDec , ExtDecList

VarDec→ID | ID ArrayDec

ArrayDec→LB Exp RB | LB Exp RB ArrayDec

FucDec→ID ( VarList ) | ID ( )

VarList→ParamDec , VarList | ParamDec

ParamDec→Specifier VarDec

CompSt→{ DefList StmList }

StmList→Stmt StmList | ε

Stmt→Exp ; | CompSt | return Exp ;| if ( Exp ) Stmt | if ( Exp ) Stmt else Stmt |

while(Exp)Stmt|FOR(Exp;Exp;Exp)Stmt|BREAK;|CONTIUNE;|;

DefList→Def DefList | ε

Def→Specifier DecList ;

DecList→Dec | Dec , DecList

Dec→VarDec | VarDec = Exp

Exp →Exp =Exp | Exp && Exp | Exp || Exp | Exp < Exp | Exp <= Exp

| Exp == Exp | Exp != Exp | Exp > Exp | Exp >= Exp

| Exp + Exp | Exp - Exp | Exp \* Exp | Exp / Exp | Exp % Exp| ID | INT | FLOAT| ( Exp )| - Exp | ! Exp | ID ( Args ) | ++Exp| Exp++

| --( Exp ) | Exp--| ID ( Args ) | ID ( ) | ID ArrayDec

Args→Exp , Args | Exp

### 2.2单词文法与语言文法

**2.2.1 单词文法描述**

Vector 中的单词可以为6类：标识符、关键字、运算符、界符、常量以及注释，具体每种单词的文法定义如表2-1所示。

表2-1 单词的文法定义

|  |  |  |
| --- | --- | --- |
| 单词符号说明 | 单词种类码 | 正则表达式 |
| id | id | [A-Za-z][A-Za-z0-9]\* |
| {int} | INT | ([0-9]+)|(0[xX][0-9a-fA-F]+)|(0[0-7]+) |
| {float} | FLOAT | ([0-9]\*\.[0-9]+)|([0-9]+\.) |
| string | string | (\".\*\") |
| {char} | CHAR | \'[0-9a-zA-Z]\' |
| “int” | TYPE |  |
| “float” | TYPE |  |
| “char” | TYPE |  |
| “string” | TYPE |  |
| “struct” | STRUCT |  |
| “return” | RETURN |  |
| “if” | IF |  |
| “else” | ELSE |  |
| “while” | WHILE |  |
| “for” | FOR |  |
| “break” | BREAK |  |
| “continue” | CONTINUE |  |
| “;” | SEMI |  |
| “,” | COMMA |  |
| ">"|"<"|">="|"<="|"=="|"!=" | RELOP |  |
| “=” | ASSIGNOP |  |
| “+” | PLUS |  |
| “-“ | MINUS |  |
| “\*” | STAR |  |
| “/” | DIV |  |
| “%” | MOD |  |
| “++” | SELFADD |  |
| “--” | SELFDEC |  |
| “+=” | ADD\_ASSIGNOP |  |
| “-=” | MINUS\_ASSIGNOP |  |
| “\*=” | STAR\_ASSIGNOP |  |
| “/=” | DIV\_ASSIGNOP |  |
| “&&” | AND |  |
| “||” | OR |  |
| “!” | NOT |  |
| “(” | LP |  |
| “)” | RP |  |
| "[" | LB |  |
| "]" | RB |  |
| “{” | LC |  |
| “}” | RC |  |
| [\n] | 换行符无种类码 |  |
| [\r\t] | 制表符无种类码 |  |
| "//"[^\n]\* | 注释，无种类码 |  |
| "/\*" {BEGIN COMMENT;}  <COMMENT>"\*/" {BEGIN INITIAL;}  <COMMENT>([^\*]|\n)+|.  <COMMENT><<EOF>>. | 多行注释，无种类码 |  |

**2.2.2语言文法描述**

首先定义非终结符的类型，结合bison的语法规则，形式为%type<union的成员名> 非终结符。

定义的非终结符如下：

%type <ptr> program ExtDefList ExtDef Specifier StructSpecifier ExtDecList FuncDec CompSt VarList VarDec ParamDec Stmt StmList DefList Def DecList Dec Exp Args

其次，用%token定义终结符的语义值类型。

%token <type\_int> INT /\*指定INT的语义值是type\_int，有词法分析得到的数值\*/

%token <type\_id> ID RELOP TYPE ARRAY /\*指定ID,RELOP 的语义值是type\_id，有词法分析得到的标识符字符串\*/

%token <type\_float> FLOAT /\*指定ID的语义值是type\_id，有词法分析得到的标识符字符串\*/

%token <type\_char> CHAR

%token LP RP LB RB LC RC SEMI COMMA FOR\_1

%token bool STRING ASSIGNOP PLUS MINUS STAR DIV AND OR DOT NOT STRUCT RETURN BREAK CONTINUE IF ELSE WHILE FOR SELFADD SELFDEC ADD\_ASSIGNOP MINUS\_ASSIGNOP STAR\_ASSIGNOP DIV\_ASSIGNOP

/\*以下为接在上述token后依次编码的枚举常量，作为AST结点类型标记\*/

%token EXT\_DEF\_LIST EXT\_VAR\_DEF FUNC\_DEF FUNC\_DEC EXT\_DEC\_LIST PARAM\_LIST PARAM\_DEC VAR\_DEF DEC\_LIST DEF\_LIST COMP\_STM STM\_LIST EXP\_STMT IF\_THEN IF\_THEN\_ELSE

%token FUNC\_CALL ARGS FUNCTION PARAM ARG CALL LABEL GOTO JLT JLE JGT JGE EQ NEQ STRUCT\_ACCESS

然后，定义运算符的优先级和综合性。

%right ASSIGNOP ADD\_ASSIGNOP MINUS\_ASSIGNOP STAR\_ASSIGNOP DIV\_ASSIGNOP

%left OR

%left AND

%left RELOP

%left PLUS MINUS

%left STAR DIV

%right UMINUS NOT SELFADD SELFDEC

%nonassoc LOWER\_THEN\_ELSE

%left DOT LP RP LB RB

%nonassoc ELSE

%nonassoc ARRAY\_FIRST

接着，是语法定义的核心部分：语法规则部分。语法规则由2.1定义的语法规则编写，具体代码实现如下：

program: ExtDefList { display($1,0);semantic\_Analysis0($1); } //显示语法树,语义分析；

/\*定义整个语法树\*/

ExtDefList: {$$=NULL;}

|ExtDef ExtDefList

{$$=mknode(2,EXT\_DEF\_LIST,yylineno,$1,$2);} //每一个EXTDEFLIST的结点，其第1棵子树对应一个外部变量声明或函数

;

/\*外部声明，外部变量或函数\*/

ExtDef:SpecifierExtDecListSEMI{$$=mknode(2,EXT\_VAR\_DEF,yylineno,$1,$2);} //该结点对应一个外部变量声明

|SpecifierFuncDecCompSt {$$=mknode(3,FUNC\_DEF,yylineno,$1,$2,$3);} //该结点对应一个函数定义

| error SEMI {$$=NULL;}

|Specifier SEMI {$$=mknode(1,EXT\_VAR\_DEF,yylineno,$1);}

;

Specifier: TYPE {

$$=mknode(0,TYPE,yylineno);strcpy($$->type\_id,$1);

if(!strcmp($1,"int")) $$->type = INT;

else if(!strcmp($1,"float")) $$->type = FLOAT;

else if(!strcmp($1,"char")) $$->type = CHAR;

else if(!strcmp($1,"string")) $$->type = STRING;

}

StructSpecifier:STRUCTIDLCDefListRC{$$=mknode(1,STRUCT,yylineno,$4);strcpy($$->type\_id,$2);}

| STRUCT ID{$$=mknode(1,STRUCT,yylineno,$2);strcpy($$->type\_id,$2);}

| STRUCT LC DefList RC {$$=mknode(1,STRUCT,yylineno,$3);}

;

/\*变量名称列表\*/

ExtDecList: VarDec {$$=$1;} /\*每一个EXT\_DECLIST的结点，其第一棵子树对应一个变量名(ID类型的结点),第二棵子树对应剩下的外部变量名\*/

|VarDecCOMMAExtDecList {$$=mknode(2,EXT\_DEC\_LIST,yylineno,$1,$3);}

;

/\*变量名称，由一个ID组成\*/

VarDec: ID {$$=mknode(0,ID,yylineno);strcpy($$->type\_id,$1);} //ID结点，标识符符号串存放结点的type\_id

| VarDec LB Exp RB {$$=mknode(2,ARRAY,yylineno,$1,$3);}

;

/\*函数定义\*/

FuncDec:IDLPVarListRP {$$=mknode(1,FUNC\_DEC,yylineno,$3);strcpy($$->type\_id,$1);}//函数名存放在$$->type\_id

|IDLPRP {$$=mknode(0,FUNC\_DEC,yylineno);strcpy($$->type\_id,$1);$$->ptr[0]=NULL;}//函数名存放在$$->type\_id

;

//参数列表

VarList: ParamDec {$$=mknode(1,PARAM\_LIST,yylineno,$1);}

|ParamDecCOMMAVarList {$$=mknode(2,PARAM\_LIST,yylineno,$1,$3);}

;

//参数定义

ParamDec:SpecifierVarDec {$$=mknode(2,PARAM\_DEC,yylineno,$1,$2);}

;

//复合语句

CompSt:LCDefListStmListRC {$$=mknode(2,COMP\_STM,yylineno,$2,$3);}

;

/\*语句列表，由0个或多个语句stmt组成\*/

StmList: {$$=NULL; }

| Stmt StmList {$$=mknode(2,STM\_LIST,yylineno,$1,$2);}

;

//语句

Stmt: Exp SEMI {$$=mknode(1,EXP\_STMT,yylineno,$1);}

| CompSt {$$=$1;} //复合语句结点直接最为语句结点，不再生成新的结点

| RETURN Exp SEMI {$$=mknode(1,RETURN,yylineno,$2);}

| IF LP Exp RP Stmt %prec LOWER\_THEN\_ELSE {$$=mknode(2,IF\_THEN,yylineno,$3,$5);}

| IF LP Exp RP Stmt ELSE Stmt {$$=mknode(3,IF\_THEN\_ELSE,yylineno,$3,$5,$7);}

| FOR LP Exp SEMI Exp SEMI Exp RP Stmt {$$=mknode(4,FOR,yylineno,$3,$5,$7,$9);}

| FOR LP Exp COMMA Exp SEMI SEMI Exp RP Stmt {$$=mknode(4,FOR\_1,yylineno,$3,$5,$8,$10);}

| WHILE LP Exp RP Stmt {$$=mknode(2,WHILE,yylineno,$3,$5);}

;

/\*定义列表\*/

DefList: {$$=NULL; }

| Def DefList {$$=mknode(2,DEF\_LIST,yylineno,$1,$2);}

| error SEMI {$$=NULL;}

;

/\*定义\*/

Def: Specifier DecList SEMI {$$=mknode(2,VAR\_DEF,yylineno,$1,$2);}

;

DecList: Dec {$$=mknode(1,DEC\_LIST,yylineno,$1);}

| Dec COMMA DecList {$$=mknode(2,DEC\_LIST,yylineno,$1,$3);}

;

Dec: VarDec {$$=$1;}

| VarDec ASSIGNOP Exp {$$=mknode(2,ASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id,"ASSIGNOP");}

;

//表达式

Exp: Exp ASSIGNOP Exp {$$=mknode(2,ASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id,"ASSIGNOP");}//$$结点type\_id空置未用，正好存放运算符

| Exp AND Exp {$$=mknode(2,AND,yylineno,$1,$3);strcpy($$->type\_id,"AND");}

| Exp OR Exp {$$=mknode(2,OR,yylineno,$1,$3);strcpy($$->type\_id,"OR");}

| Exp RELOP Exp {$$=mknode(2,RELOP,yylineno,$1,$3);strcpy($$->type\_id,$2);} //词法分析关系运算符号自身值保存在$2中

| Exp PLUS Exp {$$=mknode(2,PLUS,yylineno,$1,$3);strcpy($$->type\_id,"PLUS");}

| Exp MINUS Exp {$$=mknode(2,MINUS,yylineno,$1,$3);strcpy($$->type\_id,"MINUS");}

| Exp STAR Exp {$$=mknode(2,STAR,yylineno,$1,$3);strcpy($$->type\_id,"STAR");}

| Exp DIV Exp {$$=mknode(2,DIV,yylineno,$1,$3);strcpy($$->type\_id,"DIV");}

| LP Exp RP {$$=$2;}

| MINUS Exp %prec UMINUS {$$=mknode(1,UMINUS,yylineno,$2);strcpy($$->type\_id,"UMINUS");}

| NOT Exp {$$=mknode(1,NOT,yylineno,$2);strcpy($$->type\_id,"NOT");}

//| DPLUS Exp {$$=mknode(1,DPLUS,yylineno,$2);strcpy($$->type\_id,"DPLUS");}

//| Exp DPLUS {$$=mknode(1,DPLUS,yylineno,$1);strcpy($$->type\_id,"DPLUS");}

| Exp SELFADD {$$=mknode(1,SELFADD,yylineno,$1);strcpy($$->type\_id,"RIGHTADD");}

| SELFADD Exp {$$=mknode(1,SELFADD,yylineno,$2);strcpy($$->type\_id,"LEFTADD");}

| Exp SELFDEC {$$=mknode(1,SELFDEC,yylineno,$1);strcpy($$->type\_id,"RIGHTDEC");}

| SELFDEC Exp {$$=mknode(1,SELFDEC,yylineno,$2);strcpy($$->type\_id,"LEFTDEC");}

| Exp ADD\_ASSIGNOP Exp {$$=mknode(2,ADD\_ASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id,"ADD\_ASSIGNOP");}

| Exp MINUS\_ASSIGNOP Exp {$$=mknode(2,MINUS\_ASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id,"MINUS\_ASSIGNOP");}

| Exp STAR\_ASSIGNOP Exp {$$=mknode(2,STAR\_ASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id,"STAR\_ASSIGNOP");}

| Exp DIV\_ASSIGNOP Exp {$$=mknode(2,DIV\_ASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id,"DIV\_ASSIGNOP");}

| VarDec {$$ = $1;}

| Exp DOT Exp {$$=mknode(2,STRUCT\_ACCESS,yylineno,$1,$3);}

| ID LP Args RP {$$=mknode(1,FUNC\_CALL,yylineno,$3);strcpy($$->type\_id,$1);}

| ID LP RP {$$=mknode(0,FUNC\_CALL,yylineno);strcpy($$->type\_id,$1);}

| ID {$$=mknode(0,ID,yylineno);strcpy($$->type\_id,$1);}

| INT {$$=mknode(0,INT,yylineno);$$->type\_int=$1;$$->type=INT;}

| FLOAT {$$=mknode(0,FLOAT,yylineno);$$->type\_float=$1;$$->type=FLOAT;}

| CHAR {$$=mknode(0,CHAR,yylineno);$$->type\_char = $1;}

| BREAK {$$=mknode(0,BREAK,yylineno);strcpy($$->type\_id,"BREAK");}

| CONTINUE {$$=mknode(0,CONTINUE,yylineno);strcpy($$->type\_id,"CONTINUE");}

;

Args: Exp COMMA Args {$$=mknode(2,ARGS,yylineno,$1,$3);}

| Exp {$$=mknode(1,ARGS,yylineno,$1);}

;

### 2.3 符号表结构定义

在编译过程中，编译器使用符号表来记录源程序中各种名字的特性信息。这本次实验中采用的是单表组织形式来实现符号表，用一个符号栈来表示当前在作用域内的符号，每当有一个新的符号出现，则将其属性压栈到符号栈中，当符号出作用域则立即将这个符号弹出符号栈。

符号表的属性列包括：变量名、别名、层号、类型、标记以及偏移量。别名在后续的实验步骤中将会用到。层号用来记录符号所在的作用域的，每当程序进入一个复合语句时，层号就加一，退出时层号就减一。类型记录变量的数据类型或者是函数的返回值类型。偏移量记录变量的偏移地址。

符号表采用顺序表进行管理，用单表实现，用一个符号栈老表示在当前作用域内的付哈，每当有一个新的符号出现，则将新的符号以及对应的属性压入符号栈中。当作用域结束之后就将退栈。

//符号表，这里设置为顺序栈，index初值为0

struct symboltable{

struct symbol symbols[MAXLENGTH];

int index;

} symbolTable;

其中symbol的结构如下所示：

struct symbol { //这里只列出了一个符号表项的部分属性，没考虑属性间的互斥

char name[33]; //变量或函数名

int level; //层号

int type; //变量类型或函数返回值类型

int paramnum; //对函数适用，记录形式参数个数

char alias[10]; //别名，为解决嵌套层次使用

char flag; //符号标记，函数：'F' 变量：'V' 参数：'P' 临时变量：'T'

char offset; //外部变量和局部变量在其静态数据区或活动记录中的偏移量。

### 2.4 错误类型码定义

词法分析由工具flex实现，该阶段的错误由flex根据正则表达式语法进行处理。语法分析阶段，bison对单词流进行文法规则匹配，如果遇到不能符合任何语法结构时会自动报错。

语义分析阶段负责检查各种语义错误，主要包括：

（1）使用未定义的变量；

（2）调用未定义或未声明的函数；

（3）在同一作用域，名称的重复定义。如变量重复定义、函数重复定义等；

（4）对非函数名采用函数调用形式；

（5）对函数名采用非函数调用形式访问；

（6）函数调用时参数个数不匹配，如实参表达式个数太多、或实参表达式个数太少；

（7）函数调用时实参和形参类型不匹配；

（8）对非数组变量采用下标变量的形式访问；

（9）数组变量的下标不是整型表达式；

（10）赋值号左边不是左值表达式；

（11）对非左值表达式进行自增、自减运算；

（12）对结构体变量进行自增、自减运算；

（13）类型不匹配。如整形与字符型参与运算，字符型与浮点型参与运算等类型不匹配情况；

（14）函数返回值类型与函数定义的返回值类型不匹配；

（15）break语句不在循环语句或switch语句中；

（16）continue语句不在循环语句中；

### 2.5中间代码结构定义

选用四元式作为中间代码的形式，各种定义如表2-3所示：

表2-3 中间代码定义

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **语法** | **描述** | **Op** | **Opn1** | **Opn2** | **Result** |
| x[i] := y | 数组赋值 | ASSIGNARRAY | Y | I | X |
| x := y | 赋值操作 | ASSIGNOP | X |  | X |
| x := y + z | 加法操作 | PLUS | Y | Z | X |
| x := y - z | 减法操作 | MINUS | Y | Z | X |
| x := y \* z | 乘法操作 | STAR | Y | Z | X |
| x := y / z | 除法操作 | DIV | Y | Z | X |
| x := x + 1 | 自增操作 | AUTOADD | X | 1 | X |
| x := x - 1 | 自减操作 | AUTOSUB | X | 1 | X |
| FUNCTION f: | 定义函数f | FUNCTION |  |  | F |
| PARAM x | 函数形参 | PARAM |  |  | X |
| LABEL x | 定义标号x | LABEL |  |  | X |
| GOTO x | 无条件转移 | GOTO |  |  | X |
| IF x [relop] y GOTO z | 条件转移 | [relop] | X | Y | Z |
| ARG x | 传实参x | ARG |  |  | X |
| CALL f | 调用函数 | CALL | F |  |  |
| PARAM x | 函数形参 | PARAM |  |  | X |
| LABEL x | 定义标号x | LABEL |  |  | X |
| BLOCK x | 定义基本块x | BLOCK |  |  | X |

# 3系统设计与实现

1. **词法分析器**

词法分析器的构造技术线路，首选一个就是设计能准确表示各类单词的正则表达式。用正则表达式表示的词法规则等价转化为相应的有穷自动机FA，确定化、最小化，最后依据这个FA编写对应的词法分析程序。

而在miniC中单词可分为6类：关键字、标识符、运算符、常量、界符、注释。因此可以考虑通过利用词法生成器自动化生成工具Flex编写lex文件，以正则表达式（正规式）的形式给出词法规则，遵循上述技术线路，Flex自动生成给定的词法规则的词法分析程序，对指定的高级语言程序进行词法分析。

### 3.1.1 定义部分

定义部分其中可以有一个%{ 到%}的区间部分，主要包含c语言的一些宏定义，如文件包含、宏名定义，以及一些变量和类型的定义和声明，会直接被复制到词法分析器源程序lex.yy.c中。%{ 到%}之外的部分是一些正则式宏名的定义，例如：id [A-Za-z][A-Za-z0-9]\*，定义了一个表示标识符的宏名id，这些宏名在后面的规则部分会用到。

在vector.l文件中定义部分代码如下所示:

%{

#include "parser.tab.h"

#include "string.h"

#include "def.h"

int yycolumn=1;

#define YY\_USER\_ACTION yylloc.first\_line=yylloc.last\_line=yylineno; \

yylloc.first\_column=yycolumn;\

yylloc.last\_column=yycolumn+yyleng-1;\

yycolumn+=yyleng;

typedef union {

int type\_int;

int type\_float;

char type\_char;

char type\_id[32];

struct node \*ptr;

} YYLVAL;

#define YYSTYPE YYLVAL

%}

%option yylineno

id [A-Za-z][A-Za-z0-9]\*

int ([1-9][0-9]\*)|0

float ([0-9]\*\.[0-9]+)|([0-9]+\.)

char \'[0-9a-zA-Z]\'

string (\".\*\")

%%

{int} {printf("(%s , INT)\n",yytext);yylval.type\_int=atoi(yytext); return INT;}

{float} {printf("(%s , FLOAT)\n",yytext);yylval.type\_float=atof(yytext); return FLOAT;}

{char} {printf("(%s ,CHAR)\n",yytext);yylval.type\_char=yytext[1];return CHAR;}

### 3.1.2 规则部分

规则部分的一条规则的组成为：

正则表达式 动作

这部分以正则表达式的形式，罗列出所有种类的单词，表示词法分析器一旦识别出该正则表达式所对应的单词，就执行动作所对应的操作.

每当词法分析器识别出一个单词后，将该单词对应的字符串保存在yytext中，其长度为yyleng，供后续使用。

规则部分代码如下：

{int} {printf("(%s , INT)\n",yytext);yylval.type\_int=atoi(yytext); return INT;}

{float} {printf("(%s , FLOAT)\n",yytext);yylval.type\_float=atof(yytext); return FLOAT;}

{char} {printf("(%s ,CHAR)\n",yytext);yylval.type\_char=yytext[1];return CHAR;}

"int" {printf("(int , TYPE)\n");strcpy(yylval.type\_id, yytext);return TYPE;}

"float" {printf("(float , TYPE)\n");strcpy(yylval.type\_id, yytext);return TYPE;}

"char" {printf("(char , TYPE)\n");strcpy(yylval.type\_id, yytext);return TYPE;}

"string" {printf("(string , TYPE)\n");strcpy(yylval.type\_id, yytext);return TYPE;}

"struct" {printf("(struct , STRUCT)\n");return STRUCT;}

"return" {printf("(return , RETURN)\n");return RETURN;}

"if" {printf("(if , IF)\n");return IF;}

"else" {printf("(else , ELSE)\n");return ELSE;}

"while" {printf("(while , WHILE)\n");return WHILE;}

"break" {printf("(break , BREAK)\n");return BREAK;}

"continue" {printf("(continue , CONTINUE)\n");return CONTINUE;}

"for" {printf("(for , FOR)\n");return FOR;}

{id} {printf("(%s , ID)\n",yytext);strcpy(yylval.type\_id, yytext); return ID;/\*由于关键字的形式也符合标识符的规则，所以把关键字的处理全部放在标识符的前面，优先识别\*/}

";" {printf("(; , SEMI)\n");return SEMI;}

"," {printf("(, , COMMA)\n");return COMMA;}

">"|"<"|">="|"<="|"=="|"!=" {printf("(%s , RELOP)\n",yytext);strcpy(yylval.type\_id, yytext);return RELOP;}

"=" {printf("(= , ASSIGNOP)\n");return ASSIGNOP;}

"+" {printf("(+ , PLUS)\n");return PLUS;}

"-" {printf("(- , MINUS)\n");return MINUS;}

"\*" {printf("(\* , STAR)\n");return STAR;}

"/" {printf("(/ , DIV)\n");return DIV;}

"++" {printf("(++ , SELFADD)\n");return SELFADD;}

"--" {printf("(-- , SELFDEC)\n");return SELFDEC;}

"+=" {printf("(+= , ADD\_ASSIGNOP)\n");return ADD\_ASSIGNOP;}

"-=" {printf("(-= , MINUS\_ASSIGNOP)\n");return MINUS\_ASSIGNOP;}

"\*=" {printf("(\*= , STAR\_ASSIGNOP)\n");return STAR\_ASSIGNOP;}

"/=" {printf("(/= , DIV\_ASSIGNOP)\n");return DIV\_ASSIGNOP;}

"&&" {printf("(&& , AND)\n");return AND;}

"||" {printf("(|| , OR)\n");return OR;}

"!" {printf("(! , NOT)\n");return NOT;}

"(" {printf("(( , LP)\n");return LP;}

")" {printf("() , RP)\n");return RP;}

"{" {printf("({ , LC)\n");return LC;}

"}" {printf("(} , RC)\n");return RC;}

"." {printf("(. , DOT)\n");return DOT;}

"[" {printf("([ , LB)\n");return LB;}

"]" {printf("(] , RB)\n");return RB;}

[\n] {yycolumn=1;}

[ \r\t] {}

"//" {char c=input();while(c!='\n') c=input();}

"/\*" {

char c = input();

loop:

while(c!='\*') c = input();

c = input();

if(c != '/') goto loop;

}

. {printf("Error type A :Mysterious character \"%s\"\n\t at Line %d\n",yytext,yylineno);}

### 3.1.3 用户子程序部分

这部分代码会被原封不动的拷贝到lex.yy.c中，以方便用户自定义所需要执行的函数（包括之前的main函数）。如果用户想要对这部分用到的变量、函数或者头文件进行声明，可以前面的定义部分（即Flex源代码文件的第一部分）之前使用“%{“和”%}“符号将要声明的内容添加进去。被”%{“和”%}“所包围的内容也会被一并拷贝到lex.yy.c的最前面。

以下是用户子程序部分代码：

/\* 和bison联用时，不需要这部分

void main()

{

yylex();

return 0;

}

\*/

int yywrap()

{

return 1;

}

1. **语法分析器**

语法分析采用生成器自动化生成工具GNU Bison（前身是YACC），该工具采用了LALR（1）的自底向上的分析技术，完成语法分析。在语法分析阶段，当语法正确时，生成抽象语法树，作为后续语义分析的输入。Bison程序文件的扩展名为.y。Bison源程序代码也分为三个部分，分别为声明部分、辅助定义部分、规则部分以及用户函数部分。

### 3.2.1 声明部分

%{到%}间的声明部分内容包含语法分析中需要的头文件包含，宏定义和全局变量的定义等，这部分会直接被复制到语法分析的C语言源程序中。

声明部分代码如下所示：

#include "stdio.h"

#include "math.h"

#include "string.h"

#include "def.h"

extern int yylineno;

extern char \*yytext;

extern FILE \*yyin;

void yyerror(const char\* fmt, ...);

void display(struct ASTNode \*,int);

### 3.2.2 辅助声明部分

在辅助声明部分，主要处理实验中要用到的几个主要内容：

1. 终结符定义，在Flex和Bison联合使用时，parser.y 通过%token后面罗列出所有终结符(单词)的种类码标识符。

终结符定义代码如下：

%token LP RP LB RB LC RC SEMI COMMA FOR\_1 /\*用bison对该文件编译时，带参数-d，生成的.tab.h中给这些单词进行编码，可在lex.l中包含parser.tab.h使用这些单词种类码\*/

%token bool STRING ASSIGNOP PLUS MINUS STAR DIV AND OR DOT NOT STRUCT RETURN BREAK CONTINUE IF ELSE WHILE FOR SELFADD SELFDEC ADD\_ASSIGNOP MINUS\_ASSIGNOP STAR\_ASSIGNOP DIV\_ASSIGNOP

1. 语义值的类型定义，mini-c的文法中，每个符号（终结符和非终结符）都会有一个属性值，这个值的类型默认为整型。实际运用中，属性值的类型会有些差异，如ID的属性值类型是一个字符串，INT的属性值类型是整型。这样各种符号就会对应不同类型，这时可以用联合将这多种类型统一起来，联合定义代码如下：

%union {

int type\_int;

float type\_float;

char type\_char;

char type\_id[32];

struct ASTNode \*ptr;

};

1. 终结符属性值的类型说明，需要在%token定义终结符符号时指定其属性对应联合中的哪个成员。这样在parser.y文件中使用文法规则时，直接通过INT或ID，就可以使用整型常数自身值整数或标识符自身值字符串。

对应部分代码如下：

%token <type\_int> INT /\*指定INT的语义值是type\_int，有词法分析得到的数值\*/

%token <type\_id> ID RELOP TYPE ARRAY /\*指定ID,RELOP 的语义值是type\_id，有词法分析得到的标识符字符串\*/

%token <type\_float> FLOAT /\*指定ID的语义值是type\_id，有词法分析得到的标识符字符串\*/

%token <type\_char> CHAR

1. 非终结符属性值的类型说明，对于非终结符，如果需要完成语义计算时，会涉及到非终结符的属性值类型，这个类型对应联合的某个成员，可使用格式：%type <union的成员名> 非终结符。

%type <ptr> program ExtDefList ExtDef Specifier StructSpecifier ExtDecList FuncDec CompSt VarList VarDec ParamDec Stmt StmList DefList Def DecList Dec Exp Args

1. 优先级与结合性定义。left表示左结合，right表示右结合，前面符号的优先级低，后面的优先级高。

相应的优先级与结合性定义代码如下所示：

%right ASSIGNOP ADD\_ASSIGNOP MINUS\_ASSIGNOP STAR\_ASSIGNOP DIV\_ASSIGNOP

%left OR

%left AND

%left RELOP

%left PLUS MINUS

%left STAR DIV

%right UMINUS NOT SELFADD SELFDEC

%nonassoc LOWER\_THEN\_ELSE

%left DOT LP RP LB RB

%nonassoc ELSE

%nonassoc ARRAY\_FIRST

### 3.2.3 规则部分

Bison采用的是LR分析法，需要在每条规则后给出相应的语义动作。具体来讲是在书写产生式。第一个产生式左边非终结符默认为初始符号。产生式里的箭头在这里用冒号”:”表示，一组产生式与另一组之间以分号“；”隔开。产生式里无论是终结符还是非终结符都各自对应一个属性值，产生式右边的符号值按从左到右的顺序依次对应$1、$2、$3。

规则部分代码如下所示：program: ExtDefList { display($1,0); semantic\_Analysis0($1);} //显示语法树,语义分析

;

/\*定义整个语法树\*/

ExtDefList: {$$=NULL;} //表示整个语法树为空

| ExtDef ExtDefList {$$=mknode(2,EXT\_DEF\_LIST,yylineno,$1,$2);} //每一个EXTDEFLIST的结点，其第1棵子树对应一个外部变量声明或函数

;

/\*外部声明，外部变量或函数\*/

ExtDef: Specifier ExtDecList SEMI {$$=mknode(2,EXT\_VAR\_DEF,yylineno,$1,$2);} //该结点对应一个外部变量声明

|Specifier FuncDec CompSt {$$=mknode(3,FUNC\_DEF,yylineno,$1,$2,$3);} //该结点对应一个函数定义

// | Specifier ID ArrayDec {$$=mknode(2,ARRAY\_DF,yylineno,$1,$2);} //外部数组定义

| error SEMI {$$=NULL;printf("\t missing SEMI \t\n");}

;

/\*表示一个类型,补充对char类型的定义\*/

Specifier: TYPE {$$=mknode(0,TYPE,yylineno);strcpy($$->type\_id,$1);$$->type=!strcmp($1,"int")?INT:(!strcmp($1,"float")?FLOAT:CHAR);}

;

/\*变量名称列表\*/

ExtDecList: VarDec {$$=$1;} /\*每一个EXT\_DECLIST的结点，其第一棵子树对应一个变量名(ID类型的结点),第二棵子树对应剩下的外部变量名\*/

| VarDec COMMA ExtDecList {$$=mknode(2,EXT\_DEC\_LIST,yylineno,$1,$3);}

;

/\*变量名称，由一个ID组成\*/

VarDec: ID {$$=mknode(0,ID,yylineno);strcpy($$->type\_id,$1);} //ID结点，标识符符号串存放结点的type\_id

| ID ArrayDec {$$=mknode(1,ARRAY\_DF,yylineno,$2);strcpy($$->type\_id,$1);} //数组

;

//补充数组声明

ArrayDec: LB Exp RB {$$ = $2;/\*$$=mknode(1,ARRAY\_DEC,yylineno,$2);strcpy($$->type\_id,"ARRAY\_DEC");\*/}

| LB Exp RB ArrayDec {$$=mknode(2,ARRAY\_DEC,yylineno,$2,$4);strcpy($$->type\_id,"ARRAY\_DEC");}

| error RB {$$=NULL;printf("\t define array error \t\n");}

;

/\*函数定义\*/

FuncDec: ID LP VarList RP {$$=mknode(1,FUNC\_DEC,yylineno,$3);strcpy($$->type\_id,$1);}//函数名存放在$$->type\_id

|ID LP RP {$$=mknode(0,FUNC\_DEC,yylineno);strcpy($$->type\_id,$1);$$->ptr[0]=NULL;}//函数名存放在$$->type\_id

//| error RP {$$=NULL; printf("\t define function error\n");}

;

//这里是参数列表

VarList: ParamDec {$$=mknode(1,PARAM\_LIST,yylineno,$1);}

| ParamDec COMMA VarList {$$=mknode(2,PARAM\_LIST,yylineno,$1,$3);}

;

//参数定义

ParamDec: Specifier VarDec {$$=mknode(2,PARAM\_DEC,yylineno,$1,$2);}

;

//复合语句

CompSt: LC DefList StmList RC {$$=mknode(2,COMP\_STM,yylineno,$2,$3);}

//| error RC {$$=NULL; printf("\t compst error\n");}

;

/\*语句列表，由0个或多个语句stmt组成\*/

StmList: {$$=NULL; }

| Stmt StmList {$$=mknode(2,STM\_LIST,yylineno,$1,$2);}

;

//语句

Stmt: Exp SEMI {$$=mknode(1,EXP\_STMT,yylineno,$1);}

| CompSt {$$=$1;} //复合语句结点直接最为语句结点，不再生成新的结点

| RETURN Exp SEMI {$$=mknode(1,RETURN,yylineno,$2);}

| IF LP Exp RP Stmt %prec LOWER\_THEN\_ELSE {$$=mknode(2,IF\_THEN,yylineno,$3,$5);}

| IF LP Exp RP Stmt ELSE Stmt {$$=mknode(3,IF\_THEN\_ELSE,yylineno,$3,$5,$7);}

| WHILE LP Exp RP Stmt {$$=mknode(2,WHILE,yylineno,$3,$5);}

| FOR LP Exp SEMI Exp SEMI Exp RP Stmt {$$=mknode(4,FOR\_NODE,yylineno,$3,$5,$7,$9);}

| BREAK SEMI {$$=mknode(0,BREAK\_NODE,yylineno);strcpy($$->type\_id,"BREAK");}

| CONTINUE SEMI {$$=mknode(0,CONTINUE\_NODE,yylineno);strcpy($$->type\_id,"CONTINUE");}

| SEMI {$$ = mknode(0,BLANK,yylineno);strcpy($$->type\_id,"BLANK");} //空语句

;

/\*定义列表\*/

DefList: {$$=NULL; }

| Def DefList {$$=mknode(2,DEF\_LIST,yylineno,$1,$2);}

| error SEMI {$$=NULL;}

;

/\*定义\*/

Def: Specifier DecList SEMI {$$=mknode(2,VAR\_DEF,yylineno,$1,$2);}

//| Specifier ID ArrayDec SEMI {$$=mknode(2,ARRAY\_DF,yylineno,$1,$2);}

;

DecList: Dec {$$=mknode(1,DEC\_LIST,yylineno,$1);}

| Dec COMMA DecList {$$=mknode(2,DEC\_LIST,yylineno,$1,$3);}

;

Dec: VarDec {$$=$1;}

| VarDec ASSIGNOP Exp {$$=mknode(2,ASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id,"ASSIGNOP");}

;

//表达式

Exp: Exp ASSIGNOP Exp {$$=mknode(2,ASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id,"ASSIGNOP");}//$$结点type\_id空置未用，正好存放运算符

| Exp AND Exp {$$=mknode(2,AND,yylineno,$1,$3);strcpy($$->type\_id,"AND");}

| Exp OR Exp {$$=mknode(2,OR,yylineno,$1,$3);strcpy($$->type\_id,"OR");}

| Exp RELOP Exp {$$=mknode(2,RELOP,yylineno,$1,$3);strcpy($$->type\_id,$2);} //词法分析关系运算符号自身值保存在$2中

| Exp PLUS Exp {$$=mknode(2,PLUS,yylineno,$1,$3);strcpy($$->type\_id,"PLUS");}

| Exp MINUS Exp {$$=mknode(2,MINUS,yylineno,$1,$3);strcpy($$->type\_id,"MINUS");}

| Exp STAR Exp {$$=mknode(2,STAR,yylineno,$1,$3);strcpy($$->type\_id,"STAR");}

| Exp DIV Exp {$$=mknode(2,DIV,yylineno,$1,$3);strcpy($$->type\_id,"DIV");}

| Exp MOD Exp {$$=mknode(2,MOD,yylineno,$1,$3);strcpy($$->type\_id,"MOD");}

| Exp COMADD Exp {$$=mknode(2,COMADD,yylineno,$1,$3);strcpy($$->type\_id,"COMADD");}

| Exp COMSUB Exp {$$=mknode(2,COMSUB,yylineno,$1,$3);strcpy($$->type\_id,"COMSUB");}

| Exp COMSTAR Exp {$$=mknode(2,COMSTAR,yylineno,$1,$3);strcpy($$->type\_id,"COMSTAR");}

| Exp COMDIV Exp {$$=mknode(2,COMDIV,yylineno,$1,$3);strcpy($$->type\_id,"COMDIV");}

| Exp COMMOD Exp {$$=mknode(2,COMMOD,yylineno,$1,$3);strcpy($$->type\_id,"COMMOD");}

| LP Exp RP {$$=$2;}

| MINUS Exp %prec UMINUS {$$=mknode(1,UMINUS,yylineno,$2);strcpy($$->type\_id,"UMINUS");}

| NOT Exp {$$=mknode(1,NOT,yylineno,$2);strcpy($$->type\_id,"NOT");}

| AUTOADD Exp {$$=mknode(1,AUTOADD,yylineno,$2);strcpy($$->type\_id,"AUTOADD");}

| Exp AUTOADD {$$=mknode(1,AUTOADD,yylineno,$1);strcpy($$->type\_id,"AUTOADD");}

| AUTOSUB Exp {$$=mknode(1,AUTOSUB,yylineno,$2);strcpy($$->type\_id,"AUTOSUB");}

| Exp AUTOSUB {$$=mknode(1,AUTOSUB,yylineno,$1);strcpy($$->type\_id,"AUTOSUB");}

// | LB Args RB {$$=$2;}

//| ID ArrayDec {$$=mknode(1,ID,yylineno,$3);strcpy($$->type\_id,$1);}

| ID LP Args RP {$$=mknode(1,FUNC\_CALL,yylineno,$3);strcpy($$->type\_id,$1);} //带括号应该是函数括号内的参数内容

| ID LP RP {$$=mknode(0,FUNC\_CALL,yylineno);strcpy($$->type\_id,$1);}

| ID ArrayDec {$$=mknode(1,ARRAY\_CALL,yylineno,$2);strcpy($$->type\_id,$1);}

| ID {$$=mknode(0,ID,yylineno);strcpy($$->type\_id,$1);}

| INT {$$=mknode(0,INT,yylineno);$$->type\_int=$1;$$->type=INT;}

| FLOAT {$$=mknode(0,FLOAT,yylineno);$$->type\_float=$1;$$->type=FLOAT;}

| CHAR {$$=mknode(0,CHAR,yylineno); strcpy($$->type\_char,$1);$$->type=CHAR;}

// | SEMI {$$ = mknode(0,BLANK,yylineno);strcpy($$->type\_id,"BLANK");} //空语句

;

Args: Exp COMMA Args {$$=mknode(2,ARGS,yylineno,$1,$3);}

| Exp {$$=mknode(1,ARGS,yylineno,$1);}

;

### 3.2.4 用户函数部分

这部分的代码会被原封不动的拷贝到parser.tab.c中，以方便用户自定义所需要的函数。在这次实验中，提供了指错函数yyerror, 一旦有词法、语法错误，即可准确、及时地进行报错，并给出错误位置以及错误性质。

用户函数部分代码如下所示：

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

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

if (!yyin) return -1;

yylineno=1;

yyparse();

return 0;

}

#include<stdarg.h>

void yyerror(const char\* fmt, ...)

{

va\_list ap;

va\_start(ap, fmt);

fprintf(stderr, "Grammar Error at Line %d Column %d: ", yylloc.first\_line,yylloc.first\_column);

vfprintf(stderr, fmt, ap);

fprintf(stderr, ".\n");

}

### 3.2.5 建立抽象语法树

在语法分析阶段，一个很重要任务就是生成待编译程序的抽象语法树AST，AST不同于推导树，它去掉了一些修饰性的单词部分，简明扼要地把程序的语法结构表示出来，后续的语义分析、中间代码生成都可以通过遍历抽象语法树来完成。

AST抽象语法树节点的定义如下：

struct ASTNode {

//以下对结点属性定义没有考虑存储效率，只是简单地列出要用到的一些属性

//int kind;

enum node\_kind kind; //节点类型

union {

char type\_id[33]; //由标识符生成的叶结点

int type\_int; //由整常数生成的叶结点

float type\_float; //由浮点常数生成的叶结点

char type\_char[3]; //由字符类型生成的叶节点

};

struct ASTNode\* ptr[4]; //由kind确定有多少棵子树

int place; //存放（临时）变量在符号表的位置序号

char Etrue[15], Efalse[15]; //对布尔表达式的翻译时，真假转移目标的标号

char Snext[15]; //结点对应语句S执行后的下一条语句位置标号

struct codenode\* code; //该结点中间代码链表头指针

int type; //用以标识表达式结点的类型

int pos; //语法单位所在位置行号

int offset; //偏移量

int width; //占数据字节数

int num; //计数器，可以用来统计形参个数

};

建立抽象语法树节点的函数mknode代码如下所示：

struct ASTNode \* mknode(int num,int kind,int pos,...){

struct ASTNode \*T=(struct ASTNode \*)calloc(sizeof(struct ASTNode),1);

int i=0;

T->kind=kind;

T->pos=pos;

// va\_list pArgs = {0};

va\_list pArgs;

va\_start(pArgs, pos);

//printf("%d\n",kind);

for(i=0;i<num;i++)

T->ptr[i]= va\_arg(pArgs, struct ASTNode \*);

while (i<4) T->ptr[i++]=NULL;

//printf("%s",T->ptr[0]);

// if(kind == 262)

// printf("%s,%s\n",T->ptr[0]->type\_id,T->ptr[1]->type\_id);

va\_end(pArgs);

return T;

}

建立完抽象语法树AST后，可以通过先根遍历，逐一输出抽象语法树AST里的节点内容。遍历函数名display，代码设计如下：

void display(struct ASTNode \*T,int indent)

{//对抽象语法树的先根遍历

// printf("oooooooo\n");

// if(T)

// printf("%d\n",T->kind);

// else

// printf("error\n");

int i=1;

struct ASTNode \*T0;

if (T)

{

switch (T->kind) {

case EXT\_DEF\_LIST: display(T->ptr[0],indent); //显示该外部定义（外部变量和函数）列表中的第一个

display(T->ptr[1],indent); //显示该外部定义列表中的其它外部定义

break;

case EXT\_VAR\_DEF: printf("%\*c外部变量定义：(%d)\n",indent,' ',T->pos);

display(T->ptr[0],indent+3); //显示外部变量类型

printf("%\*c变量名：\n",indent+3,' ');

display(T->ptr[1],indent+6); //显示变量列表

break;

case TYPE: printf("%\*c类型： %s\n",indent,' ',T->type\_id);

break;

case EXT\_DEC\_LIST: display(T->ptr[0],indent); //依次显示外部变量名，

display(T->ptr[1],indent); //后续还有相同的，仅显示语法树此处理代码可以和类似代码合并

break;

case FUNC\_DEF: printf("%\*c函数定义：(%d)\n",indent,' ',T->pos);

display(T->ptr[0],indent+3); //显示函数返回类型

display(T->ptr[1],indent+3); //显示函数名和参数

display(T->ptr[2],indent+3); //显示函数体

break;

case FUNC\_DEC: printf("%\*c函数名：%s\n",indent,' ',T->type\_id);

if (T->ptr[0]) {

printf("%\*c函数形参：\n",indent,' ');

display(T->ptr[0],indent+3); //显示函数参数列表

}

else printf("%\*c无参函数\n",indent+3,' ');

break;

case PARAM\_LIST: display(T->ptr[0],indent); //依次显示全部参数类型和名称，

display(T->ptr[1],indent);

break;

case PARAM\_DEC:

// char tp[8] = "error";

// int id = T->ptr[0]->type;

// if(id == INT)

// strcpy(tp,"int");

// else if(id == FLOAT)

// strcpy(tp,"float");

// else if(id == CHAR)

// strcpy(tp,"char");

// else if(id == STRING)

// strcpy(tp,"string");

printf("%\*c类型：%s, 参数名：%s\n",indent,' ',T->ptr[0]->type\_id,T->ptr[1]->type\_id);

break;

case EXP\_STMT: printf("%\*c表达式语句：(%d)\n",indent,' ',T->pos);

display(T->ptr[0],indent+3);

break;

case RETURN: printf("%\*c返回语句：(%d)\n",indent,' ',T->pos);

display(T->ptr[0],indent+3);

break;

case COMP\_STM: printf("%\*c复合语句：(%d)\n",indent,' ',T->pos);

printf("%\*c复合语句的变量定义部分：\n",indent+3,' ');

display(T->ptr[0],indent+6); //显示定义部分

printf("%\*c复合语句的语句部分：\n",indent+3,' ');

display(T->ptr[1],indent+6); //显示语句部分

break;

case STM\_LIST: display(T->ptr[0],indent); //显示第一条语句

display(T->ptr[1],indent); //显示剩下语句

break;

case WHILE: printf("%\*c循环语句：(%d)\n",indent,' ',T->pos);

printf("%\*c循环条件：\n",indent+3,' ');

display(T->ptr[0],indent+6); //显示循环条件

printf("%\*c循环体：(%d)\n",indent+3,' ',T->pos);

display(T->ptr[1],indent+6); //显示循环体

break;

case FOR: printf("%\*cFOR循环：(%d)\n",indent,' ',T->pos);

printf("%\*c表达式1：\n",indent,' ');

display(T->ptr[0],indent + 3);

printf("%\*c表达式2：\n",indent,' ');

display(T->ptr[1],indent + 3);

printf("%\*c表达式3：\n",indent,' ');

display(T->ptr[2],indent + 3);

printf("%\*c循环体：\n",indent,' ');

display(T->ptr[3],indent + 6);

break;

case FOR\_1: printf("%\*cFOR循环：(%d)\n",indent,' ',T->pos);

printf("%\*c表达式1：\n",indent,' ');

display(T->ptr[0],indent + 3);

display(T->ptr[1],indent + 3);

printf("%\*c表达式2：\n",indent,' ');

printf("%\*c表达式3：\n",indent,' ');

display(T->ptr[2],indent + 3);

printf("%\*c循环体：\n",indent,' ');

display(T->ptr[3],indent + 6);

break;

case IF\_THEN: printf("%\*c条件语句(IF\_THEN)：(%d)\n",indent,' ',T->pos);

printf("%\*c条件：\n",indent+3,' ');

display(T->ptr[0],indent+6); //显示条件

printf("%\*cIF子句：(%d)\n",indent+3,' ',T->pos);

display(T->ptr[1],indent+6); //显示if子句

break;

case IF\_THEN\_ELSE: printf("%\*c条件语句(IF\_THEN\_ELSE)：(%d)\n",indent,' ',T->pos);

printf("%\*c条件：\n",indent+3,' ');

display(T->ptr[0],indent+6); //显示条件

printf("%\*cIF子句：(%d)\n",indent+3,' ',T->pos);

display(T->ptr[1],indent+6); //显示if子句

printf("%\*cELSE子句：(%d)\n",indent+3,' ',T->pos);

display(T->ptr[2],indent+6); //显示else子句

break;

case DEF\_LIST: display(T->ptr[0],indent); //显示该局部变量定义列表中的第一个

display(T->ptr[1],indent); //显示其它局部变量定义

break;

case VAR\_DEF: printf("%\*c局部变量定义：(%d)\n",indent,' ',T->pos);

display(T->ptr[0],indent+3); //显示变量类型

display(T->ptr[1],indent+3); //显示该定义的全部变量名

break;

case DEC\_LIST: printf("%\*c变量名：\n",indent,' ');

T0=T;

while (T0) {

if (T0->ptr[0]->kind==ID)

printf("%\*c %s\n",indent+6,' ',T0->ptr[0]->type\_id);

else if (T0->ptr[0]->kind==ASSIGNOP)

{

printf("%\*c %s ASSIGNOP\n ",indent+6,' ',T0->ptr[0]->ptr[0]->type\_id);

display(T0->ptr[0]->ptr[1],indent+strlen(T0->ptr[0]->ptr[0]->type\_id)+7); //显示初始化表达式

}

else

{

display(T0->ptr[0],indent + 3);

}

T0=T0->ptr[1];

}

break;

case CONTINUE:

case BREAK: printf("%\*c%s\n",indent,' ',T->type\_id);

break;

case ID: printf("%\*cID： %s\n",indent,' ',T->type\_id);

break;

case CHAR: printf("%\*cCHAR: %c\n",indent,' ',T->type\_char);

break;

case INT: printf("%\*cINT：%d\n",indent,' ',T->type\_int);

break;

case FLOAT: printf("%\*cFLAOT：%f\n",indent,' ',T->type\_float);

break;

case ARRAY: for(int i = 0;i<indent;i++) putchar(' ');

printf("ARRAY：\n");

//printf("%\*c%s[] 数组大小：%d\n",indent,' ',T->ptr[0]->type\_id,T->ptr[1]->type\_int);

display(T->ptr[0],indent + 3);

display(T->ptr[1],indent + 3);

break;

case STRUCT: for(int i = 0;i<indent;i++) putchar(' ');

printf("STRUCT: ID:%s\n",T->type\_id);

display(T->ptr[0],indent + 3);

break;

case STRUCT\_ACCESS:

printf("%\*c结构成员：",indent,' ');

printf("%s . %s\n",T->ptr[0]->type\_id,T->ptr[1]->type\_id);

break;

case SELFADD:

if(!strcmp(T->type\_id,"RIGHTADD"))

{

printf("%\*c++(right)\n",indent,' ');

}

else

printf("%\*c++(left)\n",indent,' ');

display(T->ptr[0],indent + 3);

break;

case SELFDEC:

if(!strcmp(T->type\_id,"RIGHTDEC"))

{

printf("%\*c--(right)",indent,' ');

}

else

printf("%\*c--(left)",indent,' ');

display(T->ptr[0],indent + 3);

break;

case ADD\_ASSIGNOP:

case MINUS\_ASSIGNOP:

case STAR\_ASSIGNOP:

case DIV\_ASSIGNOP:

case ASSIGNOP:

case AND:

case OR:

case RELOP:

case PLUS:

case MINUS:

case STAR:

case DIV:

printf("%\*c%s\n",indent,' ',T->type\_id);

display(T->ptr[0],indent+3);

display(T->ptr[1],indent+3);

break;

case NOT:

case UMINUS: printf("%\*c%s\n",indent,' ',T->type\_id);

display(T->ptr[0],indent+3);

break;

case FUNC\_CALL: printf("%\*c函数调用：(%d)\n",indent,' ',T->pos);

printf("%\*c函数名：%s\n",indent+3,' ',T->type\_id);

display(T->ptr[0],indent+3);

break;

case ARGS: i=1;

while (T) { //ARGS表示实际参数表达式序列结点，其第一棵子树为其一个实际参数表达式，第二棵子树为剩下的

struct ASTNode \*T0=T->ptr[0];

printf("%\*c第%d个实际参数表达式：\n",indent,' ',i++);

display(T0,indent+3);

T=T->ptr[1];

}

// printf("%\*c第%d个实际参数表达式：\n",indent,' ',i);

// display(T,indent+3);

printf("\n");

break;

}

}

}

1. **符号表管理**

符号表结构定义如2.3节所述，在本节不在赘述。

在语义分析过程中，各个变量名有其对应的作用域，一个作用域内不允许名字重复，为此，通过一个全局变量LEV来管理，LEV的初始值为0。这样在处理外部变量名，以及函数名时，对应符号的层号值都是0；处理函数形式参数时，固定形参名在填写符号表时，层号为1。由于vector中允许有复合语句，复合语句中可定义局部变量，函数体本身也是一个复合语句，这样在AST的遍历中，通过LEV的修改来管理不同的作用域。

1. 每次遇到一个复合语句的结点COM\_STM，首先对LEV加1，表示准备进入一个新的作用域，为了管理这个作用域中的变量，使用栈symbol\_scope\_TX，记录该作用域变量在符号表中的起点位置，即将符号表symbolTable的栈顶位置symbolTable.index保存在栈symbol\_scope\_TX中。
2. 每次要登记一个新的符号到符号表中时，首先在symbolTable中，从栈顶向栈底方向查层号为LEV的符号，是否有和当前待登记的符号重名，是则报重复定义错误，否则使用LEV作为层号将新的符号登记到符号表中。
3. 每次遍历完一个复合语句结点COM\_STM的所有子树，准备回到其父结点时，这时该复合语句语义分析完成，需要从符号表中删除该复合语句的变量，方法是首先symbol\_scope\_TX退栈，取出该复合语句作用域的起点，再根据这个值修改symbolTable.index，同时LEV减一，很简单地完成了符号表的符号删除操作。
4. 符号表的查找操作，在AST的遍历过程中，分析各种表达式，遇到变量的访问时，在symbolTable中，从栈顶向栈底方向查询是否有相同的符号定义，如果全部查询完后没有找到，就是该符号没有定义；如果相同符号在符号表中有多处定义，按查找的方向可知，符合就近优先的原则。如果查找到符号后，就进一步进行语义分析，如：（1）函数调用时，根据函数名在符号表找到的是一个变量，不是函数，需要报错；（2）函数调用时，根据函数名找到这个函数，需要判断参数个数、类型是否匹配；（3）根据变量名查找的是一个函数。等等，需要做出各种检查。
5. **语义检查**

语义检查这部分完成的是静态语义分析，具体实现的检错功能如2.4节所述，在本节不在赘述。语义检查部分主要包括：

1. 控制流检查。控制流语句必须使得程序跳转到合法的地方。例如一个跳转语句会使控制转移到一个由标号指明的后续语句。如果标号没有对应到语句，那么就出现一个语义错误。再者，break、continue语句必须出现在循环语句当中。

在vector没有定义各种转移语句，实验时可以考虑增加上去；教材中 有break语句的介绍，可供参考。

1. 唯一性检查。对于某些不能重复定义的对象或者元素，如同一作用域的标识符不能同名，需要在语义分析阶段检测出来。
2. 名字的上下文相关性检查。名字的出现在遵循作用域与可见性的前提下应该满足一定的上下文的相关性。如变量在使用前必须经过声明，如果是面向对象的语言，在外部不能访问私有变量等等。
3. 类型检查包括检查函数参数传递过程中形参与实参类型是否匹配、是否进行自动类型转换等等。

由于时间原因，仅仅完成了静态语义分析之前的部分。

# 4系统测试与评价

**4.1测试用例**

int main()

{

int a,b,c;

a=10;b=20;c=30;

c=++a + ++a+ b++ +b++;

c=a==b || a>10 && b<20;

return 1;

}

int main()

{

int a,b,c,m;

while (a<b)

{

if (a<b) continue;

for(c=1;c<10;c++)

if (c<5) continue;

else break;

}

return 1;

}

int main()

{

int a,b,s;

a=0;

s=0;

while(a<6)

{

a=a+1;

if ((b=a/2)\*2==a) continue;

s+=a;

if (a==10) break;

}

write(s);

return 1;

}

实验二测试代码如下所示：

int array\_read(struct ASTNode \*T) //返回数据类型

{

// printf("offset:%d",T->offset);

struct opn opn1, opn2, result;

struct ASTNode \*T1 = T; //记录数组的开始结点

int i = 0,

j = 0, k;

int rtn, tmp, tmp1;

int record[MAXLENGTH]; //记录访问的每一维数的大小

struct ASTNode \*Node\_Array[MAXLENGTH];

int total = 0;

while (T->kind == ARRAY\_READ)

{

// semantic\_Analysis(T->ptr[1]);

// printf("IR:");

// prnIR(T->ptr[1]->code);

switch (T->ptr[1]->type)

{

case FLOAT:

case CHAR:

semantic\_error(T->pos, "", "数组下标表达式非法");

break;

}

Node\_Array[i] = T->ptr[1];

T = T->ptr[0];

i++;

}

rtn = searchSymbolTable(T->type\_id);

if (rtn == -1)

{

semantic\_error(T->pos, T->type\_id, "数组未定义");

return -1;

}

else if (symbolTable.symbols[rtn].flag == 'F')

{

semantic\_error(T->pos, T->type\_id, "函数名不能用下标访问");

return -1;

}

else if (symbolTable.symbols[rtn].flag == 'V')

{

semantic\_error(T->pos, T->type\_id, "非数组变量，不能用下标访问");

return -1;

}

else if (symbolTable.symbols[rtn].flag == 'P')

{

semantic\_error(T->pos, T->type\_id, "非数组变量，不能用下标访问");

return -1;

}

else if (symbolTable.symbols[rtn].array\_info->dimension != i)

{

semantic\_error(T->pos, "", "数组维数错误");

return -1;

}

for (j = 0; j < i; j++)

{

if (record[j] >= symbolTable.symbols[rtn].array\_info->diminfo[j])

{

semantic\_error(T->pos, "", "数组越界");

return -1;

}

// int temp = 1;

// for (k = j; k > 0; k--)

// {

// temp \*= symbolTable.symbols[rtn].array\_info->diminfo[k - 1];

// }

// total += temp \* record[j];

// temp = 0;

//record 3 2 1 k

//diminfo 4 3 2 j

//real 2 3 4

// total += record[]

}

// prn\_symbol();

// printf("%d\n", Node\_Array[i - 1]->kind);

semantic\_Analysis(Node\_Array[i - 1]);

for (j = i - 2; j >= 0; j--)

{

//先计算乘法,乘上前一位的元素上限

T->place = fill\_Temp(newTemp(), LEV, INT, 'T', Node\_Array[j]->offset);

strcpy(result.id, symbolTable.symbols[T->place].alias);

result.kind = ID;

opn2.kind = INT;

opn2.const\_int = symbolTable.symbols[rtn].array\_info->diminfo[j];

opn1.kind = ID;

strcpy(opn1.id, symbolTable.symbols[Node\_Array[j + 1]->place].alias);

T1->code = merge(2, T1->code, genIR(STAR, opn1, opn2, result));

//分析树的右半部分生成临时变量

semantic\_Analysis(Node\_Array[j]);

//再计算加法

tmp = fill\_Temp(newTemp(), LEV, INT, 'T', Node\_Array[j]->offset);

strcpy(result.id, symbolTable.symbols[tmp].alias);

result.kind = ID;

opn2.kind = ID;

strcpy(opn2.id, symbolTable.symbols[Node\_Array[j]->place].alias);

opn1.kind = ID;

strcpy(opn1.id, symbolTable.symbols[T->place].alias);

// printf("tmp name : %s = %s + %s \n ", symbolTable.symbols[tmp].alias, symbolTable.symbols[T->place].alias, symbolTable.symbols[Node\_Array[j]->place].alias);

T1->code = merge(3, T1->code, Node\_Array[j]->code, genIR(PLUS, opn1, opn2, result));

}

//计算总的偏移数，以字节为基本单位

tmp1 = fill\_Temp(newTemp(), LEV, INT, 'T', T->offset);

strcpy(result.id, symbolTable.symbols[tmp1].alias);

result.kind = ID;

opn2.kind = INT;

opn2.const\_int = (symbolTable.symbols[rtn].array\_info->type == CHAR ? 1 : 4);

opn1.kind = ID;

if(i==1)

strcpy(opn1.id, symbolTable.symbols[Node\_Array[i - 1]->place].alias);

else

strcpy(opn1.id, symbolTable.symbols[tmp].alias);

T1->code = merge(2, T1->code, genIR(STAR, opn1, opn2, result));

// prnIR(T1->code);

T1->place = searchSymbolTable(T->type\_id);

symbolTable.symbols[T1->place].array\_info->tartget = tmp1;

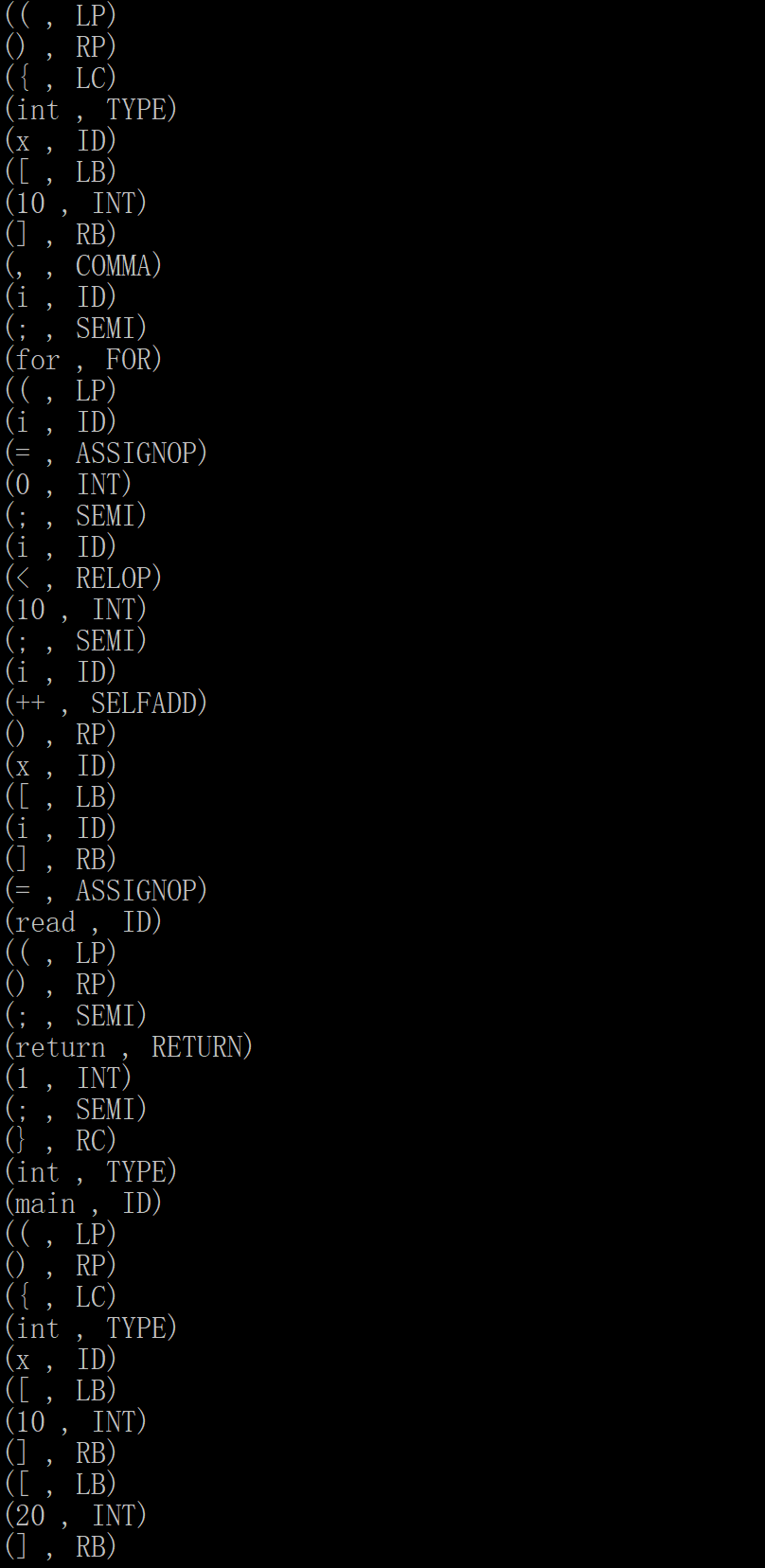
// prn\_symbol();

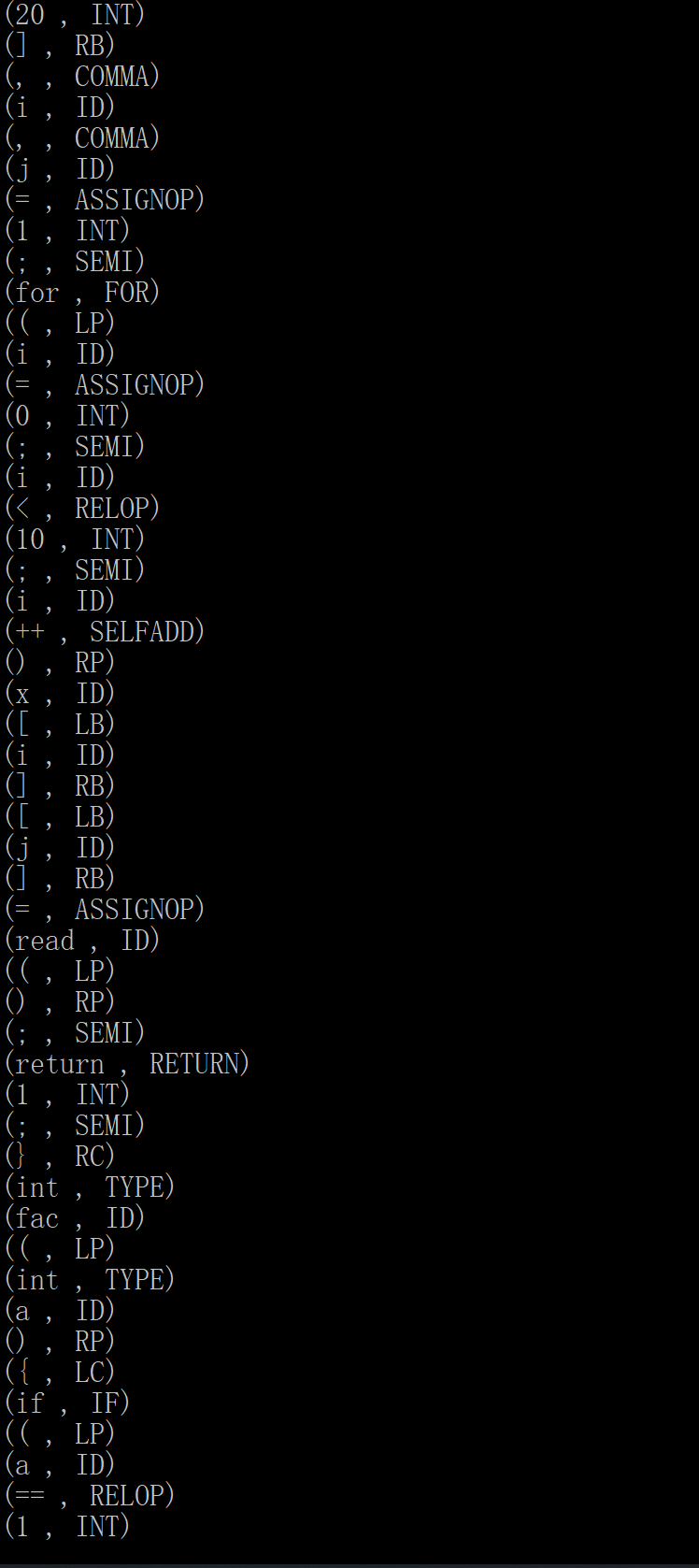
return symbolTable.symbols[rtn].array\_info->type;

}

**4.2正确性测试**

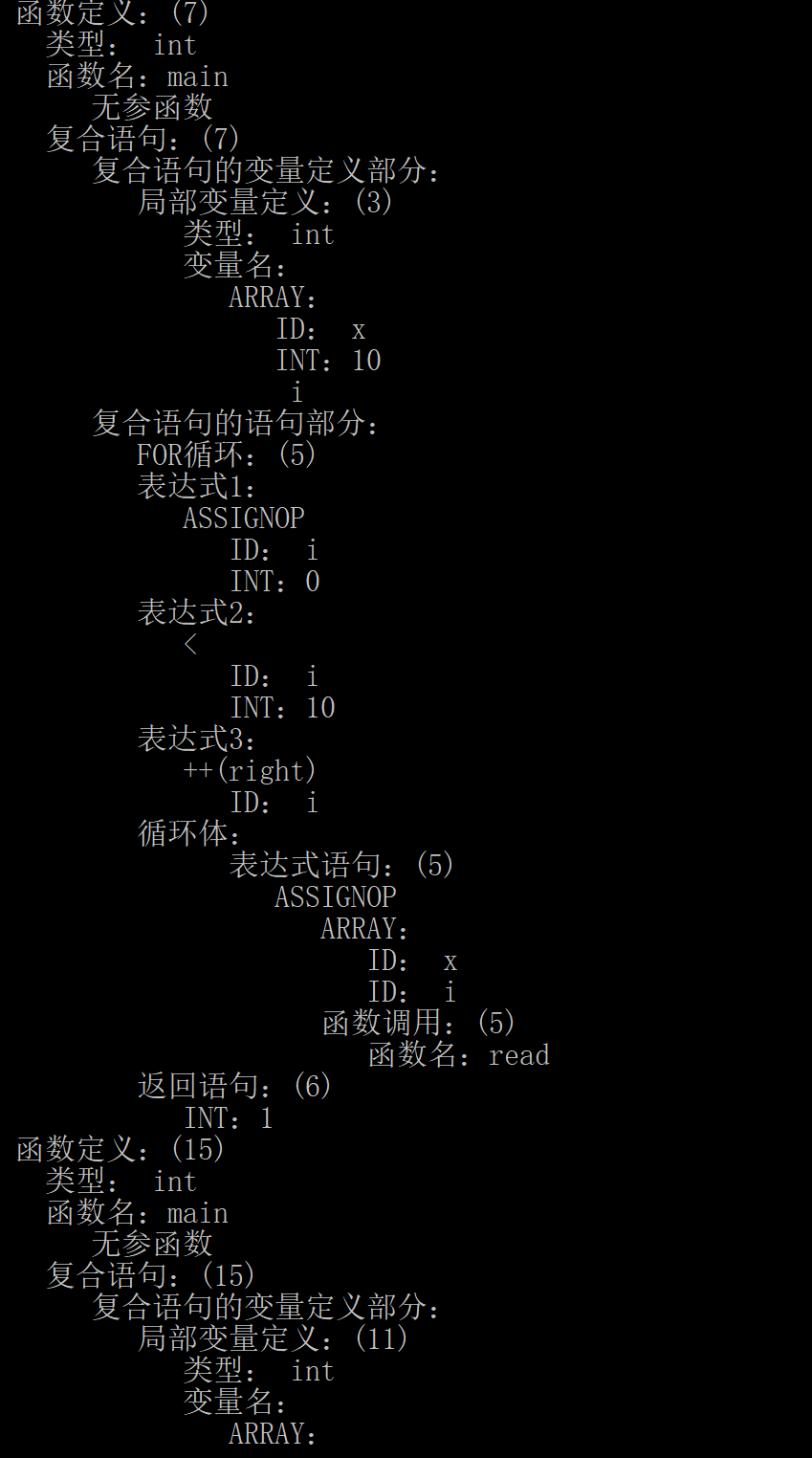
实验一的测试结果如图4-2-1所示：

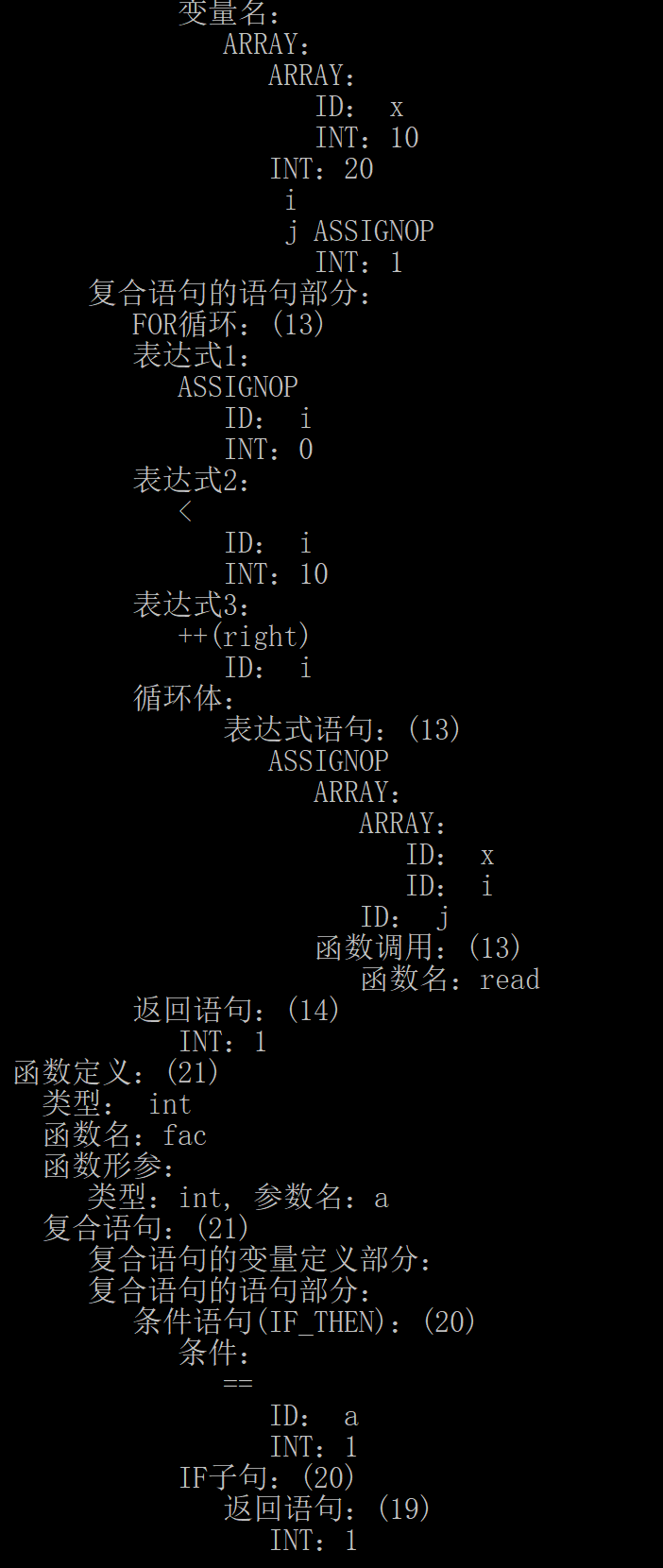
****

****

**图4-2-1 实验二的测试结果图**

实验三测试结果如图4-2-2所示：





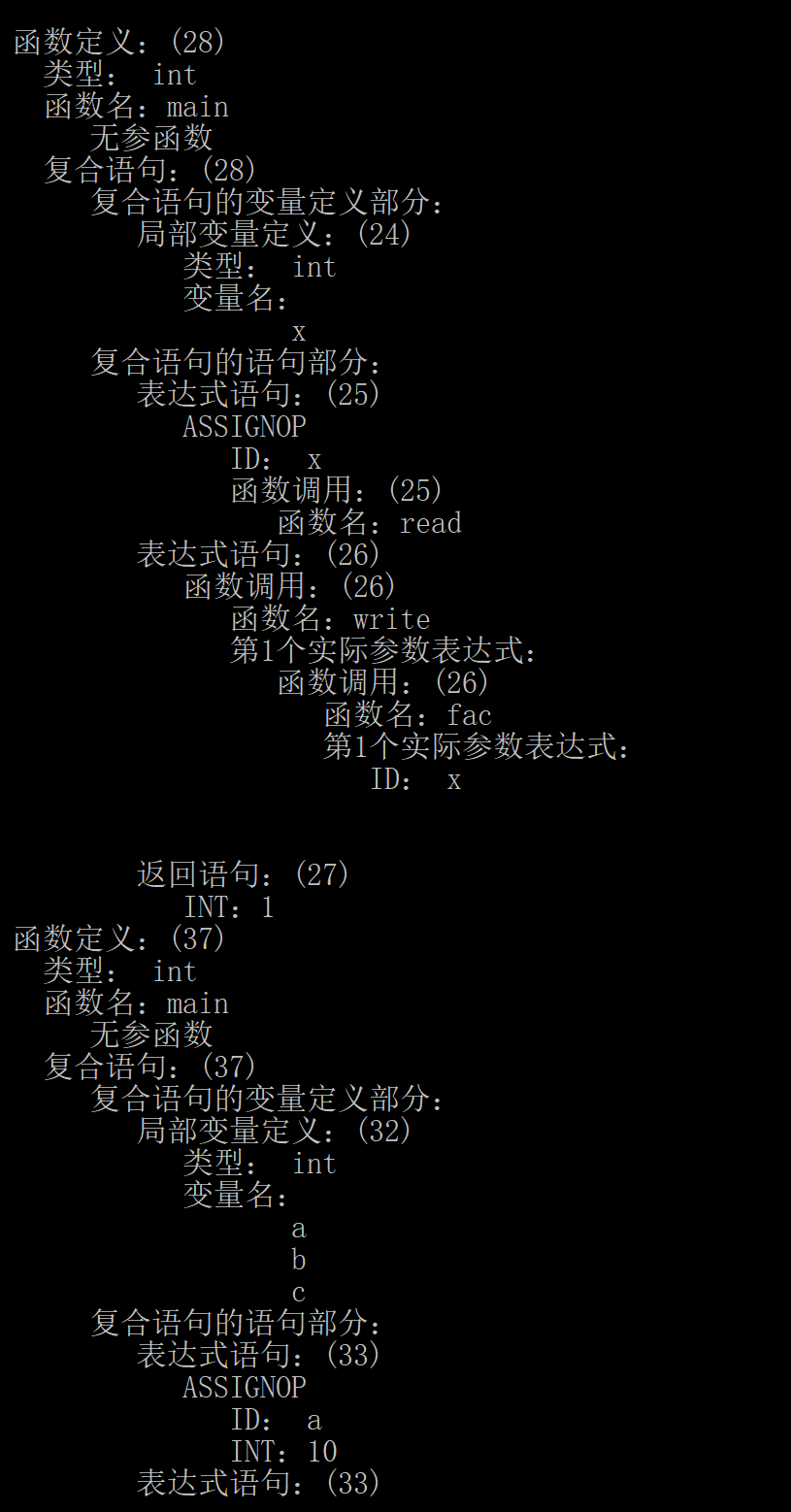


图4-2-2 实验三测试结果图

实验四、五测试结果如图4-2-3所示：



**4.3 系统的优点**

系统在miniC的基础上，实现了单行注释、多行注释、char类型、数组支持、continue、break语句等多种功能，并且也实现了空语句、16进制、8进制赋值等功能，能够检测16种静态语义错误并给出详细的出错信息，能够详细详细地展示编译过程中每一步的结果。

**4.4 系统的缺点**

实验未完成，仅仅完成了五个实验。没有将源程序最后完美的生成目标程序。

# 

# 5.实验小结或体会

编译原理实验可以说是计算机专业中比较难的实验之一。我在这门实验上耗费了很多的时间，但是很可惜这门实验没有最终很好的完成。仅仅把静态语义分析完成，后来我又尝试生成中间代码，很可惜检查的时间已经截止了。

通过这8个实验，我仅仅做完了Vector编译器的一部分。虽然还有很多功能不支持，还有很多的缺点和不足，但是在这次实验中我对编译器的工作流程有了更深层次的理解，加深了我对词法分析，语法分析，语义分析的掌握。

在实验中我完成了如下内容：

1.运用flex工具编写词法分析器，通过编写单词对应的正则表达式规则，实现了程序的词法分析。

2.运用bison编写语法分析器，并在老师给定的框架上加入了新的语法规则，如for循环、break、continue语句、数组识别等，实现了源代码的语法分析。

3.完善实验指导教程里的display函数，实现了抽象语法树的遍历。

4.按照实验指导教程设计符号表，并完成符号表里符号的管理。

5.在前面完成内容的基础上，实现了静态语义检查，能够检查出代码里诸如类型不匹配，break、continue语句出现位置错误，表达式需要左值等合计16种静态语义错误。

通过这次实验，我感受到了编译原理实验是一个非常具有挑战性的实验，编译原理实验不仅巩固了课上所讲述的课程内容，也让我进一步了解到真正设计编程语言时应该注意到的问题以及诸如寄存器分配算法等许多课外知识。从一开始的毫无头绪去下手，到后来有了头绪，以及对环境的配置，还有对flex和bison还有gcc的熟练应用，都花费了很多的时间。还有抽象语法树和符号表的创建也是最有挑战的实验。

最后，感谢指导老师杨茂林老师，他真是一位很好的老师，在这次实验中我碰到的问题老师都会一一解答，帮我调试。真的很感谢他。通过这次实验我看到了自己的缺陷和不足，今后我会更加努力的去学习，提高自己的知识技术水平。

**参考文献**

[1] 王生元 等. 编译原理(第三版). 北京：清华大学出版社，20016

[2] 胡伦俊等. 编译原理(第二版). 北京：电子工业出版社，2005

[3] 王元珍等. 80X86汇编语言程序设计. 武汉：华中科技大学出版社,2005

[4] 王雷等. 编译原理课程设计. 北京：机械工业出版社，2005

[5] 曹计昌等. C语言程序设计. 北京：科学出版社，2008