

**实验报告**

**题目： 简单的C语言编译器**

**课程名称： 编译原理**

**专业班级： CS1601**

**学 号： U201614532**

**姓 名： 吕鹏泽**

**指导教师： 祝建华**

**报告日期： 2019年6月16日**

**计算机科学与技术学院**

**目录**

[1选题背景 1](#_Toc11612254)

[2系统关键定义 2](#_Toc11612255)

[2.1单词文法描述 2](#_Toc11612256)

[2.2语句文法描述 2](#_Toc11612257)

[2.3 符号表结构描述 5](#_Toc11612258)

[2.4 错误类型码描述 6](#_Toc11612259)

[2.5中间代码描述 7](#_Toc11612260)

[3系统设计与实现 9](#_Toc11612261)

[3.1编译程序符号表结构 9](#_Toc11612262)

[3.2编译程序报错功能 9](#_Toc11612263)

[3.3词法语法分析器（实验一） 9](#_Toc11612264)

[3.4语义分析（实验二） 12](#_Toc11612265)

[3.5中间代码生成功能（实验三） 13](#_Toc11612266)

[4系统测试与评价 15](#_Toc11612267)

[4.1 测试用例 15](#_Toc11612268)

[4.2 正确性测试 16](#_Toc11612269)

[4.3 报错功能测试 24](#_Toc11612270)

[4.4 系统的优点 25](#_Toc11612271)

[4.5 系统的缺点 25](#_Toc11612272)

[5实验小结或体会 26](#_Toc11612273)

[**参考文献** 27](#_Toc11612274)

[**附件：源代码** 28](#_Toc11612275)

# 1选题背景

本次课程设计是构造一个高级语言的子集的编译器，目标代码可以是汇编语言也可以是其他形式的机器语言。按照任务书，实现的方案可以有很多种选择。

可以根据自己对编程语言的定义选择实现语言的特定功能。建议大家选用decaf语言。

编译器的语法和词法分析采用课程的课堂实验的结果，重点在语义分析、符号表结构设计、中间代码、目标代码存储结构设计、代码优化等阶段的实现。

课设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，提高自己对系统软件编写的兴趣。

# 2系统关键定义

## 2.1单词文法描述

单词是本语言中具有独立意义的最小单位，可分为 5 大类： 关键字（保留字）、 运算符、 界符、 常量和标识符。

**关键字**

类型关键字：int、float、char；

分支循环关键字：if、else、while；

返回语句：return；

**运算符**

算术运算：+、-、\*、/、++、--；

复杂算术运算：+=，-=，\*=，/=；

逻辑运算：&&，||，！；

比较运算符：<、>、=>、<=、==、!=；

赋值运算符：=；

**界符**

分号，逗号，小括号，中括号，大括号，空白字符（包括空格、回车）；

**注释**

单行注释：**//**

多行注释：**/\***xxx**\*/**

**常量**

整型常量：如：12

浮点常量：如：1.2

字符常量：如：‘a’）

## 2.2语句文法描述

本实验实现的简单C语言文法描述如下：

$accept: program $end

program: ExtDefList

ExtDefList: /\* empty \*/

| ExtDef ExtDefList

ExtDef: Specifier ExtDecList SEMI

| Specifier FuncDec CompSt

| error SEMI

Specifier: TYPE

ExtDecList: VarDec

| VarDec COMMA ExtDecList

VarDec: ID

| VarDec LB INT RB

FuncDec: ID LP VarList RP

| ID LP RP

VarList: ParamDec

| ParamDec COMMA VarList

ParamDec: Specifier VarDec

CompSt: LC DefList StmList RC

StmList: /\* empty \*/

| Stmt StmList

Stmt: Exp SEMI

| CompSt

| RETURN Exp SEMI

| IF LP Exp RP Stmt

| IF LP Exp RP Stmt ELSE Stmt

| WHILE LP Exp RP Stmt

DefList: /\* empty \*/

| Def DefList

Def: Specifier DecList SEMI

DecList: Dec

| Dec COMMA DecList

Dec: VarDec

| VarDec ASSIGNOP Exp

Exp: Exp ASSIGNOP Exp

| Exp AND Exp

| Exp OR Exp

| Exp RELOP Exp

| Exp PLUS Exp

| Exp MINUS Exp

| Exp STAR Exp

| Exp DIV Exp

| LP Exp RP

| MINUS Exp

| Exp COMP\_PLUS Exp

| Exp COMP\_MINUS Exp

| Exp INC

| Exp DEC

| INC Exp

| DEC Exp

| NOT Exp

| ID LP Args RP

| ID LP RP

| VarDec LB INT RB

| ID

| INT

| FLOAT

| CHAR

Args: Exp COMMA Args

| Exp

**作用域：**

支持多种层次的作用域。最高层是全局作用域，其中只由函数定义，每个函数由自己的函数作用域，包括一个用于声明参数表的参数作用域和存放函数体的局部作用域。

局部作用域中的变量必须先声明后使用，同一个作用域中的标识符是唯一的，在嵌套的作用域中重新声明的标识符屏蔽外层的同名标识符，不可访问在一个已经关闭的作用域中声明的标识符。

**类型：**

预定义好的基本类型有int，float和char。

**变量：**

变量的类型可以是int，float和char之一。 在一个类定义内部、 但是不在任何函数内声明的变量具有类作用域。

在函数参数表中声明的变量具有参数作用域，而在函数体中声明的变量具有局部作用域。一旦被声明，则该变量保持可见直到该作用域关闭。需要注意的是：局部变量只能在作用域的开始部分声明。

**函数：**

函数的定义用于建立函数名字以及与这个名字相关联的类型签名，类型签名包括函数返回值类型、 形参表的大小以及各形参的类型。 函数的定义提供类型签名以及组成函数体的语句。

## 2.3 符号表结构描述

符号表设计如下所示，由symbol组成的结构体数组，存储符号表的信息。其本质是一个顺序栈，index初值为0。

struct symboltable{

struct symbol symbols[MAXLENGTH];

int index;//栈顶指针

}symbolTable;

符号表的节点设计如下：

struct symbol{

char name[33];

int level;

int type;

int paramnum;

char alias[10];

char flag;

char offset;

};

name表示变量或函数名；

level表示层号，外部变量名或函数名层号为0，形参名为1，每到1个复合语句层号加1，退出减1；

type表示变量类型或函数返回值类型；

paramnum表示形式参数个数；

alias[10]表示别名，为解决嵌套层次使用，使得每一个数据名称唯一；

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

offset表示偏移量；

局部变量堆栈结构，存储各作用域的起始位置序号，top存储当前作用域符号在符号表的位置。结构实现如下：

struct symbol\_scope\_begin{

int TX[30];

int top;

} symbol\_scope\_TX;

## 2.4 错误类型码描述

ERROR 1：变量重复定义

ERROR 2：函数调用参数太少

ERROR 3：参数类型不匹配

ERROR 4：函数调用参数太多

ERROR 5：变量未定义

ERROR 6：是函数名，类型不匹配

ERROR 7：操作数类型不一致

ERROR 8：赋值语句必须为左值

ERROR 9：除数不能为0

ERROR 10：自增自减的操作数必须是整形变量

ERROR 11：函数未定义

ERROR 12：不是一个函数

ERROR 13：函数重复定义

ERROR 14：参数名重复定义

ERROR 15：返回值类型不匹配

## 2.5中间代码描述

采用三地址代码TAC作为中间语言，中间语言代码的定义如表4-1所示。

**表4-1 中间代码定义**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **语法** | **描述** | **Op** | **Opn1** | **Opn2** | **Result** |
| LABEL x | 定义标号x | LABEL |  |  | X |
| FUNCTION f: | 定义函数f | FUNCTION |  |  | F |
| x := y | 赋值操作 | ASSIGN | 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 |
| GOTO x | 无条件转移 | GOTO |  |  | X |
| IF x [relop] y GOTO z | 条件转移 | [relop] | X | Y | Z |
| RETURN x | 返回语句 | RETURN |  |  | X |
| ARG x | 传实参x | ARG |  |  | X |
| x:=CALL f | 调用函数 | CALL | F |  | X |
| PARAM x | 函数形参 | PARAM |  |  | X |
| READ x | 读入 | READ |  |  | X |
| WRITE x | 打印 | WRITE |  |  | X |

结构体定义如下：

struct opn{

int kind;//标识操作的类型

int type;//标识操作数的类型

union{

int const\_int;//整常数值，立即数

float const\_float;//浮点常数值，立即数

char const\_char;//字符常数值，立即数

double const\_double;

char id[33];//变量或临时变量的别名或标号字符串

};

int level;//变量的层号

int offset;//偏移量

};

struct codenode{//三地址TAC代码结点

int op;//TAC代码的运算符种类

struct opn opn1,opn2,result;//2个操作数和运算结果

struct codenode \*next,\*prior;

};

# 3系统设计与实现

## 3.1编译程序符号表结构

实验中采用双向链表这种方式管理符号表。此时的符号表symbolTable是一个顺序栈，栈顶指针index初始值为0，每次填写符号时，将新的符号填写到栈顶位置，再栈顶指针加1。

## 3.2编译程序报错功能

为了处理未定义就使用的情形，所要做的就是在一个表达式使用了变量或者调用了函数时，查找符号表中是否存在对应的符号，如果没有则说明发生了这种错误，进行报错，然后继续分析。

至于重复定义的情况，就是在定义函数或者变量时，在符号表中查找有无同名函数或同名且同作用域的变量，如果有则报错，且不添加重复符号，否则就添加新符号。

为了解决类型不匹配的问题，需要一个计算值表达式类型的函数，因为大多数赋值表达式的右部都不是一个单一的数字，往往是一个复杂的表达式，至于它计算之后的值是什么类型，则需要该函数分析。

对于除数为0的错误，目前只能识别较为简单的情况，毕竟判断一个复杂表达式是否为0并不是静态语义分析的工作，现在能做的，就是在遇到除法时，检验除数是否为数字0，如果是则报错。

## 3.3词法语法分析器（实验一）

词法分析采用flex工具，代码文件为lex.l，其格式为：

**定义部分**

**%%**

**规则部分**

**%%**

**用户子程序部分**

其中规则部分代码如下：

{int} {yylval.type\_int=atoi(yytext);return INT;}

{float} {yylval.type\_float=atof(yytext);return FLOAT;}

{char} {yylval.type\_char=yytext[1];return CHAR;}

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

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

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

"return" {return RETURN;}

"if" {return IF;}

"else" {return ELSE;}

"while" {return WHILE;}

{id} {strcpy(yylval.type\_id, yytext);return ID;}//由于关键词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

";" {return SEMI;}

"," {return COMMA;}

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

"=" {return ASSIGNOP;}

"++" {return INC;}

"+=" {return COMP\_PLUS;}

"+" {return PLUS;}

"--" {return DEC;}

"-=" {return COMP\_MINUS;}

"-" {return MINUS;}

"\*" {return STAR;}

"/" {return DIV;}

"&&" {return AND;}

"||" {return OR;}

"!" {return NOT;}

"(" {return LP;}

")" {return RP;}

"[" {return LB;}

"]" {return RB;}

"{" {return LC;}

"}" {return RC;}

[\n] {yycolumn=1;}//每到新的一行列数置1

[ \r\t] {}

(\/\/).\* {}//匹配注释//

. {printf("ERROR:未定义的字符 %s at line %d colemn %d\n",yytext,yylineno,yycolumn);}

语法分析的任务是生成待编译程序的抽象语法树AST，然后采用先根遍历的方式将其输出。其实现采用的是自动化生成工具Bison，Bison可以根据给 定的语法规则，自动化生成对应的语法分析程序。但是，语法分析的目的不仅仅是判断源程序的语句是否符合语法规则，还应该（如果符合语法规则）构造源程序对应的语法分析树，用于编译的后续阶段。Bison 和 Flex 可以无缝对接，即将 Flex进行词法分析后得到的单词序列作为Bison的输入，从而用来进行语法分析。

经过分析，将语法树的节点类型分为以下几种：

外部变量定义列表：EXT\_DEF\_LIST

外部变量定义：EXT\_VAR\_DEF

函数定义FUNC\_DEF

函数声明：FUNC\_DEC

表达式定义列表：EXT\_DEC\_LIST

数组定义：ARRAY\_DEF

参数列表：PARAM\_LIST

参数定义：PARAM\_DEC

一个局部变量定义：VAR\_DEF

局部变量列表：DEC\_LIST

声明列表：DEF\_LIST

复合语句：COMP\_STM

复合语句列表：STM\_LIST

表达式语句：EXP\_STMT

if语句：IF\_THEN,

if else语句：IF\_THEN\_ELSE

函数调用：FUNC\_CALL

多个实参的列表：ARGS

函数：FUNCTION

形参：PARAM

一个实参说明：ARG

抽象语法树的树节点的定义如下：

struct node{

enum node\_kind kind;//结点类型

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

double type\_double;//由双精度浮点数生成的叶节点

char type\_char;//由字符生成的叶节点

int array\_size[2];//数组大小

};

struct node \*ptr[3];//子树指针，由kind确定有多少棵子树

int level;//层号

int place;//表示结点对应的变量或运算结果符号表的位置序号

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];//该结点对饮语句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

char op[10];

int type;//结点对应值的类型

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;//参数个数

};

## 3.4语义分析（实验二）

构建符号表的主要工作是正确地添加符号、删除符号和明确符号的作用域。当遇到函数定义、形参定义和变量定义时需要添加符号，这些标识符在使用时不必处理，只需在定义时添加一次即可。

假设全局作用域为0，那么函数的作用域自然为0，因为它定义在全局范围内。函数的形参有一个单独的作用域，在函数作用域的基础上加一，那么形参作用域的数值就为1。函数体内的代码与形参应该不属于同一个域，故作用域需要再加一。每次进入一个if-else块或者while块中作用域也需要再当前基础上加一。

例如在处理一个函数结点时，会添加一个符号，类型为函数，然后继续处理它的形参和代码部分，作用域level也分别增加。以赋值语句的左值检测为例，代码实现如下：

case ASSIGNOP:

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

{

semantic\_error(T->pos, "", "ERROR 8：赋值语句必须为左值");

}

else

{

Exp(T->ptr[0]); //处理左值，例中仅为变量

T->ptr[1]->offset = T->offset;

Exp(T->ptr[1]);

T->type = T->ptr[0]->type;

T->width = T->ptr[1]->width;

T->code = merge(2, T->ptr[0]->code, T->ptr[1]->code);

opn1.kind = ID;

strcpy(opn1.id, symbolTable.symbols[T->ptr[1]->place].alias); //右值一定是个变量或临时变量

opn1.offset = symbolTable.symbols[T->ptr[1]->place].offset;

result.kind = ID;

strcpy(result.id, symbolTable.symbols[T->ptr[0]->place].alias);

result.offset = symbolTable.symbols[T->ptr[0]->place].offset;

T->code = merge(2, T->code, genIR(ASSIGNOP, opn1, opn2, result));

}

break;

## 3.5中间代码生成功能（实验三）

三地址中间代码TAC是一个4元组，逻辑上包含（op、opn1、opn2、result），其中op表示操作类型说明，opn1和opn2表示2个操作数，result表示运算结果。后续还需要根据个TAC序列生成目标代码，所以设计其存储结构时，每一部分要考虑目标代码生成是所需要的信息。

（1）运算符：表示这条指令需要完成的运算，可以用枚举常量表示，如PLUS表示双目加，JLE表示小于等于，PARAM表示形参，ARG表示实参等。

（2）操作数与运算结果：这些部分包含的数据类型有多种，整常量，实常量，还有使用标识符的情况，如变量的别名、变量在其数据区的偏移量和层号、转移语句中的标号等。类型不同，所以考虑使用联合。为了明确联合中的有效成员，将操作数与运算结果设计成结构类型，包含kind，联合等几个成员，kind说明联合中的有效，联合成员是整常量，实常量或标识符表示的别名或标号或函数名等。

（3）为了配合后续的TAC代码序列的生成，将TAC代码作为数据元素，用双向循环链表表示TAC代码序列。

# 4系统测试与评价

## 4.1 测试用例

test\_error.c

int fun(int a, int b)

{

return 0;

}

int test(int input)

{

int a;

int a;

float b;

fun(b,1);

fun(1, 2);

fun(1, 2, 3);

fun = 3;

if(1.0 < a)

b = 3;

b++;

a = t;

b = a;

a = 1/0;

a + b = 3;

fun1();

a = b();

a = fun(1,2);

}

int fun()

{

return 'a';

}

int g(int a, char a)

{

return 0;

}

test.c

int main(int input){

int a=0;

float b=1.2;

char c='?';

a=5+c-2\*3;

a=++a - a--;

a=-c\*8 + 3;

if(a==0) {

float b;

b=3.66;

}

else

{

b = 2.5;

}

while (a>0)

{

a = 3;

}

a = 4;

}

## 4.2 正确性测试

函数定义：

类型： int

函数名：main

函数形参：

类型：int, 参数名：input

复合语句：

复合语句的变量定义：

局部变量定义：

类型： int

变量定义：

a ASSIGNOP

INT：0

局部变量定义：

类型： float

变量定义：

b ASSIGNOP

FLAOT：1.200000

局部变量定义：

类型： char

变量定义：

c ASSIGNOP

CHAR：?

复合语句的语句部分：

表达式语句：

ASSIGNOP

标识符： a

MINUS

PLUS

INT：5

标识符： c

STAR

INT：2

INT：3

表达式语句：

ASSIGNOP

标识符： a

MINUS

PRE\_INC

标识符： a

POST\_DEC

标识符： a

表达式语句：

ASSIGNOP

标识符： a

PLUS

STAR

UMINUS

标识符： c

INT：8

INT：3

条件语句(IF\_THEN\_ELSE)：

条件：

==

标识符： a

INT：0

IF子句：

复合语句：

复合语句的变量定义：

局部变量定义：

类型： float

变量定义：

b

复合语句的语句部分：

表达式语句：

ASSIGNOP

标识符： b

FLAOT：3.660000

ELSE子句：

复合语句：

复合语句的变量定义：

复合语句的语句部分：

表达式语句：

ASSIGNOP

标识符： b

FLAOT：2.500000

循环语句：

循环条件：

>

标识符： a

INT：0

循环体：

复合语句：

复合语句的变量定义：

复合语句的语句部分：

表达式语句：

ASSIGNOP

标识符： a

INT：3

表达式语句：

ASSIGNOP

标识符： a

INT：4

变量名 别 名 层 号 类 型 标记 偏移量

read 0 int F 4

write 0 int F 4

x 1 int P 12

main v1 0 int F 0

input v2 1 int P 12

a v3 1 int V 16

temp1 1 int T 20

b v4 1 float V 24

temp2 1 float T 32

c v5 1 char V 36

temp3 1 char T 37

temp4 1 int T 38

temp5 1 int T 42

temp6 1 int T 46

temp7 1 int T 50

temp8 1 int T 54

temp9 1 int T 58

temp10 1 int T 38

temp11 1 int T 38

temp12 1 int T 42

temp13 1 int T 42

temp14 1 int T 46

temp15 1 char T 38

temp16 1 char T 38

temp17 1 int T 39

temp18 1 int T 43

temp19 1 int T 47

temp20 1 int T 51

temp21 1 int T 38

b v6 2 float V 38

temp22 2 float T 46

变量名 别 名 层 号 类 型 标记 偏移量

read 0 int F 4

write 0 int F 4

x 1 int P 12

main v1 0 int F 0

input v2 1 int P 12

a v3 1 int V 16

temp1 1 int T 20

b v4 1 float V 24

temp2 1 float T 32

c v5 1 char V 36

temp3 1 char T 37

temp4 1 int T 38

temp5 1 int T 42

temp6 1 int T 46

temp7 1 int T 50

temp8 1 int T 54

temp9 1 int T 58

temp10 1 int T 38

temp11 1 int T 38

temp12 1 int T 42

temp13 1 int T 42

temp14 1 int T 46

temp15 1 char T 38

temp16 1 char T 38

temp17 1 int T 39

temp18 1 int T 43

temp19 1 int T 47

temp20 1 int T 51

temp21 1 int T 38

temp23 2 float T 38

变量名 别 名 层 号 类 型 标记 偏移量

read 0 int F 4

write 0 int F 4

x 1 int P 12

main v1 0 int F 0

input v2 1 int P 12

a v3 1 int V 16

temp1 1 int T 20

b v4 1 float V 24

temp2 1 float T 32

c v5 1 char V 36

temp3 1 char T 37

temp4 1 int T 38

temp5 1 int T 42

temp6 1 int T 46

temp7 1 int T 50

temp8 1 int T 54

temp9 1 int T 58

temp10 1 int T 38

temp11 1 int T 38

temp12 1 int T 42

temp13 1 int T 42

temp14 1 int T 46

temp15 1 char T 38

temp16 1 char T 38

temp17 1 int T 39

temp18 1 int T 43

temp19 1 int T 47

temp20 1 int T 51

temp21 1 int T 38

temp24 1 int T 38

temp25 2 int T 38

变量名 别 名 层 号 类 型 标记 偏移量

read 0 int F 4

write 0 int F 4

x 1 int P 12

main v1 0 int F 0

input v2 1 int P 12

a v3 1 int V 16

temp1 1 int T 20

b v4 1 float V 24

temp2 1 float T 32

c v5 1 char V 36

temp3 1 char T 37

temp4 1 int T 38

temp5 1 int T 42

temp6 1 int T 46

temp7 1 int T 50

temp8 1 int T 54

temp9 1 int T 58

temp10 1 int T 38

temp11 1 int T 38

temp12 1 int T 42

temp13 1 int T 42

temp14 1 int T 46

temp15 1 char T 38

temp16 1 char T 38

temp17 1 int T 39

temp18 1 int T 43

temp19 1 int T 47

temp20 1 int T 51

temp21 1 int T 38

temp24 1 int T 38

temp26 1 int T 38

FUNCTION main :

PARAM v2

temp1 := #0

v3 := temp1

temp2 := #1.200000

v4 := temp2

temp3 := #63

v5 := temp3

temp4 := #5

temp5 := temp4 + v5

temp6 := #2

temp7 := #3

temp8 := temp6 \* temp7

temp9 := temp5 - temp8

v3 := temp9

temp10 := v3

temp11 := v3 + #1

v3 := temp11

temp12 := v3

temp13 := v3 - #1

v3 := temp13

temp14 := temp11 - temp12

v3 := temp14

temp15 := v5

temp16 := v5 - #1

v5 := temp16

temp17 := #8

temp18 := temp16 \* temp17

temp19 := #3

temp20 := temp18 + temp19

v3 := temp20

temp21 := #0

IF v3 == temp21 GOTO label6

GOTO label7

LABEL label6 :

temp22 := #3.660000

v6 := temp22

GOTO label5

LABEL label7 :

temp23 := #2.500000

v4 := temp23

LABEL label5 :

LABEL label10 :

temp24 := #0

IF v3 > temp24 GOTO label9

GOTO label8

LABEL label9 :

temp25 := #3

v3 := temp25

GOTO label10

LABEL label8 :

temp26 := #4

v3 := temp26

LABEL label1 :

## 4.3 报错功能测试

在9行,a ERROR 1：变量重复定义

在11行, ERROR 3：参数类型不匹配

在13行, ERROR 4：函数调用参数太多

在15行,fun ERROR 6：是函数名，类型不匹配

在16行, ERROR 7：操作数类型不一致

在18行,b ERROR 10：自增自减的操作数必须是整形变量

在19行,t ERROR 5：变量未定义

在21行, ERROR 9：除数不能为0

在22行, ERROR 8：赋值语句必须为左值

在23行,fun1 ERROR 11：函数未定义

在24行,b ERROR 12：不是一个函数

在29行,fun ERROR 13：函数重复定义

在31行,ERROR 15：返回值类型不匹配

在34行,a ERROR 14：参数名重复定义

图4.1报错功能截图

## 4.4 系统的优点

实现了简单的C语言编译器，支持int、char、float三种类型的变量，支持函数，能处理15种语义错误。

## 4.5 系统的缺点

没有实现较为复杂的数组、结构体功能。

# 5实验小结或体会

总体来说，非常满意自己这学期在编译原理实验中的一些成果和心得。

实验耗时很长，实验过程中知道了如何把理论课程上的终结符与非终结符概念运用到语言设计上，另外，学会了如何利用正则表达式表达某一类符号。我学会了如何编写词法分析器的正规式（正规表达式）和使用自动化生成工具Flex 得到词法分析程序；学会了如何使用自动化生成工具 Bison得到语法分析程序，以及如何在Bison环境中使用属性文法、语义动作增强文法的表达能力；学会了如何利用源程序的语法分析树，对其进行语义分析；学会了如何对源程序进行分析而能得到其符号表，以及如何维护和检测源程序的作用域栈；学会了中间代码的常用形式三地址码，以及如何将源代码转化为中间代码。

**参考文献**

[1] 吕映芝等. 编译原理(第二版). 北京：清华大学出版社，2005

[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

**附件：源代码**

**lex.l**

%{

#include "parser.tab.h"

#include "string.h"

#include "def.h"

#include "stdio.h"

int yycolumn=1;

#define YY\_USER\_ACTION yylloc.first\_line=yylloc.last\_line=yylineno; yylloc.last\_column=yycolumn+yyleng-1; yycolumn+=yyleng;

typedef union{

int type\_int;

int type\_float;

int type\_char;

char type\_id[32];

struct node \*pr;

}YYLVAL;

#define YYSTYPE YYLVAL

%}

%option yylineno

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

int [0-9]+

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

char \'.\'

%%

{int} {yylval.type\_int=atoi(yytext);return INT;}

{float} {yylval.type\_float=atof(yytext);return FLOAT;}

{char} {yylval.type\_char=yytext[1];return CHAR;}

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

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

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

"return" {return RETURN;}

"if" {return IF;}

"else" {return ELSE;}

"while" {return WHILE;}

{id} {strcpy(yylval.type\_id, yytext);return ID;}//由于关键词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

";" {return SEMI;}

"," {return COMMA;}

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

"=" {return ASSIGNOP;}

"++" {return INC;}

"+=" {return COMP\_PLUS;}

"+" {return PLUS;}

"--" {return DEC;}

"-=" {return COMP\_MINUS;}

"-" {return MINUS;}

"\*" {return STAR;}

"/" {return DIV;}

"&&" {return AND;}

"||" {return OR;}

"!" {return NOT;}

"(" {return LP;}

")" {return RP;}

"[" {return LB;}

"]" {return RB;}

"{" {return LC;}

"}" {return RC;}

[\n] {yycolumn=1;}//每到新的一行列数置1

[ \r\t] {}

(\/\/).\* {}//匹配注释//

. {printf("ERROR:未定义的字符 %s at line %d colemn %d\n",yytext,yylineno,yycolumn);}

%%

/\* 和bison联用时，不需要这部分

void main()

{

yylex();

return 0;

}

\*/

int yywrap()

{

return 1;

}

**parser.y**

%error-verbose

%locations

%{

#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, ...);//可变长参数

void display(struct node \*,int);

%}

%union{

int type\_int;

float type\_float;

char type\_char;

char type\_id[32];

struct node \*ptr;

};

//%type定义非终结符的语义值类型(对应union中成员ptr的类型，本实验中为一个树节点的指针)

/\*非终结符类型说明\*/

%type <ptr> program //初始语法单元

%type <ptr> ExtDefList //0个或多个ExtDef

%type <ptr> ExtDef //一个全局变量、结构体或函数的定义

%type <ptr> Specifier //类型描述符

%type <ptr> ExtDecList //0个或多个变量，如a,b,c (int a,b,c;)

%type <ptr> FuncDec //函数头

%type <ptr> CompSt //函数体

%type <ptr> VarList //形参列表

%type <ptr> VarDec //一个变量的定义

%type <ptr> ParamDec //一个形参的定义

%type <ptr> Stmt //一条语句

%type <ptr> StmList //多条语句

%type <ptr> DefList //多个变量定义

%type <ptr> Def //一条变量定义

%type <ptr> DecList //变量列表

%type <ptr> Dec //一个变量

%type <ptr> Exp //一个表达式

%type <ptr> Args //实参列表

/\*终结符类型说明\*/

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

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

%token <type\_float> FLOAT//指定FLOAT的语义是type\_id,由词法分析得到的标识符字符串

%token <type\_char> CHAR

//括号

%token LP RP LB RB LC RC

//分隔符

%token SEMI COMMA

//运算符

%token PLUS MINUS STAR DIV ASSIGNOP AND OR NOT

//关键字

%token IF ELSE WHILE RETURN

//特殊运算符

%token INC DEC COMP\_PLUS COMP\_MINUS

/\*指定优先级\*/

%left ASSIGNOP

%left OR

%left AND

%left RELOP

%left COMP\_PLUS COMP\_MINUS

%left PLUS MINUS

%left STAR DIV

%right UMINUS NOT INC DEC

%nonassoc LOWER\_THEN\_ELSE

%nonassoc ELSE

%%

//程序

program: ExtDefList {display($1,0);semantic\_Analysis0($1);prnIR($1->code);}

;

//外部变量定义列表

ExtDefList: {$$=NULL;}

|ExtDef ExtDefList {$$=mknode(EXT\_DEF\_LIST,$1,$2,NULL,yylineno);} //形如int a;int b;int c;

;

//一个全局变量定义|一个函数定义|一个结构体定义

ExtDef: Specifier ExtDecList SEMI {$$=mknode(EXT\_VAR\_DEF,$1,$2,NULL,yylineno);} //形如int a,b;

|Specifier FuncDec CompSt {$$=mknode(FUNC\_DEF,$1,$2,$3,yylineno);} //形如int fun() {}

|error SEMI {$$=NULL;}

;

//类型描述符

Specifier: TYPE {$$=mknode(TYPE,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);

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

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

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

;

//0个或多个变量，如a,b,c (int a,b,c;)

//每一个EXT\_DEFLIST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

ExtDecList: VarDec {$$=$1;}

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

;

//标识符，标识符符号串存放节点的type\_id

VarDec: ID {$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}

| VarDec LB INT RB {$$=mknode(ARRAY\_DEF,$1,NULL,NULL,yylineno);$$->array\_size[0]=$3;}//一维数组

;

//函数头

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

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

;

//形参列表

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

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

;

//一个形参的定义

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

;

//函数体

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

;

//语句列表

StmList: {$$=NULL;}

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

;

//一条语句

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

| 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(RETURN,$2,NULL,NULL,yylineno);}

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

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

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

;

//局部变量定义列表

DefList: {$$=NULL;}

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

;

//一个局部变量定义

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

;

//0个或多个局部变量，形如a,b,c

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

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

;

//一个局部变量，形如a

Dec: VarDec {$$=$1;}

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

;

//一个表达式

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

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

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

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

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

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

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

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

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

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

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

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

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

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

| INC Exp {$$=mknode(INC,$2,NULL,NULL,yylineno);strcpy($$->type\_id,"PRE\_INC");}

| DEC Exp {$$=mknode(DEC,$2,NULL,NULL,yylineno);strcpy($$->type\_id,"PRE\_DEC");}

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

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

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

| VarDec LB INT RB {$$=mknode(ARRAY\_DEF,$1,NULL,NULL,yylineno);strcpy($$->type\_id,"ARRAY\_DEF");$$->array\_size[0]=$3;}

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

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

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

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

;

//实参列表

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

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

;

%%

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

vfprintf(stderr, fmt, ap);

fprintf(stderr, ".\n");

}

**def.h**

#include "stdio.h"

#include "stdlib.h"

#include "string.h"

#include "stdarg.h"

#include "parser.tab.h"

enum node\_kind{EXT\_DEF\_LIST,EXT\_VAR\_DEF,FUNC\_DEF,FUNC\_DEC,EXT\_DEC\_LIST,ARRAY\_DEF,TWO\_ARRAY\_DEF,PARAM\_LIST,PARAM\_DEC,VAR\_DEF,DEC\_LIST,DEF\_LIST,COMP\_STM,STM\_LIST,EXP\_STMT,IF\_THEN,IF\_THEN\_ELSE,FUNC\_CALL,ARGS,FUNCTION,PARAM,ARG,CALL,LABEL,GOTO,JLT,JLE,JGT,JGE,EQ,NEQ,TAG};

#define MAXLENGTH 1000//定义符号表的大小

#define DX 3\*sizeof(int)//活动记录控制信息需要的单元数

struct opn{

int kind;//标识操作的类型

int type;//标识操作数的类型

union{

int const\_int;//整常数值，立即数

float const\_float;//浮点常数值，立即数

char const\_char;//字符常数值，立即数

double const\_double;//?????????????????????????????????????????????????????????????????????????????????

char id[33];//变量或临时变量的别名或标号字符串

};

int level;//变量的层号，0表示是全局变量，数据保存在静态数据区

int offset;//变量单元偏移量，或函数在符号表的定义位置序号，目标代码生成时用

};

struct codenode{//三地址TAC代码结点,采用双向循环链表存放中间语言代码

int op;//TAC代码的运算符种类

struct opn opn1,opn2,result;//2个操作数和运算结果

struct codenode \*next,\*prior;

};

struct node{//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

enum node\_kind kind;//结点类型

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

double type\_double;//由双精度浮点数生成的叶节点

char type\_char;//由字符生成的叶节点

int array\_size[2];//数组大小

};

struct node \*ptr[3];//子树指针，由kind确定有多少棵子树

int level;//层号

int place;//表示结点对应的变量或运算结果符号表的位置序号

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];//该结点对饮语句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

char op[10];

int type;//结点对应值的类型

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;//参数个数

};

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，外部变量名或函数名层号为0，形参名为1，每到1个复合语句层号加1，退出减1

int type;//变量类型或函数返回值类型

int paramnum;//形式参数个数

char alias[10];//别名，为解决嵌套层次使用，使得每一个数据名称唯一

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

//或函数活动记录大小，目标代码生成时使用

};

//符号表，是一个顺序栈，index初值为0

struct symboltable{

struct symbol symbols[MAXLENGTH];

int index;//栈顶指针

}symbolTable;

/\*当前作用域的符号在符号表的起始位置序号,这是一个栈结构。

每到达一个复合语句，将符号表的index值进栈，

离开复合语句时，取其退栈值修改符号表的index值，

完成删除该复合语句中的所有变量和临时变量\*/

struct symbol\_scope\_begin{

int TX[30];

int top;

} symbol\_scope\_TX;

struct node \*mknode(int kind,struct node \*first,struct node \*second,struct node \*third,int pos);

void semantic\_Analysis0(struct node \*T);

void semantic\_Analysis(struct node \*T);

void boolExp(struct node \*T);

void Exp(struct node \*T);

void objectCode(struct codenode \*head);发

**ast.c**

#include "def.h"

// #define DISPLAY

struct node \* mknode(int kind,struct node \*first,struct node \*second, struct node \*third,int pos) {

struct node \*T=(struct node \*)malloc(sizeof(struct node));

T->kind=kind;

T->ptr[0]=first;

T->ptr[1]=second;

T->ptr[2]=third;

T->pos=pos;

return T;

}

void display(struct node \*T,int indent){//对抽象语法树的先根遍历,indent为标识符

int i=1;

struct node \*T0;

if (T){

switch (T->kind){

case EXT\_DEF\_LIST:

display(T->ptr[0],indent);//显示该外部定义列表中的第一个

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：\n",indent,' ');

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：\n",indent,' ');

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:

if(T->ptr[0]->type==INT)

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

if(T->ptr[0]->type==FLOAT)

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

if(T->ptr[0]->type==CHAR)

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

break;

case EXP\_STMT:

printf("%\*c表达式语句：\n",indent,' ');

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

break;

case RETURN:

printf("%\*c返回语句：\n",indent,' ');

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

break;

case COMP\_STM:

printf("%\*c复合语句：\n",indent,' ');

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

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：\n",indent+3,' ');

display(T->ptr[1],indent+6);//显示循环体

break;

case IF\_THEN:

printf("%\*c条件语句(IF\_THEN)：\n",indent,' ');

printf("%\*c条件：\n",indent+3,' ');

display(T->ptr[0],indent+6);//显示条件

printf("%\*cIF子句：\n",indent+3,' ');

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)：\n",indent,' ');

printf("%\*c条件：\n",indent+3,' ');

display(T->ptr[0],indent+6);//显示条件

printf("%\*cIF子句：\n",indent+3,' ');

display(T->ptr[1],indent+6);//显示if子句

printf("%\*cELSE子句：\n",indent+3,' ');

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：\n",indent,' ');

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变量定义：\n",indent,' ');

T0=T;

while (T0) {

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

printf("%\*c %s\n",indent+3,' ',T0->ptr[0]->type\_id);

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

printf("%\*c %s ASSIGNOP\n ",indent+3,' ',T0->ptr[0]->ptr[0]->type\_id);

//显示初始化表达式

display(T0->ptr[0]->ptr[1],indent+strlen(T0->ptr[0]->ptr[0]->type\_id)+4);

}

T0=T0->ptr[1];

}

break;

case ID:

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

break;

case TAG:

printf("%\*c结构体变量名： %s\n",indent,' ',T->type\_id);

break;

case ARRAY\_DEF:

printf("%\*c一维数组定义： \n",indent,' ');

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

printf("%\*cSIZE: %d\n",indent,' ',T->array\_size[0]);

break;

case INT:

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

break;

case FLOAT:

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

break;

case CHAR:

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

break;

case ASSIGNOP:

case AND:

case OR:

case RELOP:

case COMP\_PLUS:

case COMP\_MINUS:

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 INC:

case DEC:

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：\n",indent,' ');

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 node \*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("\n");

break;

}

}

}

char \*strcat0(char \*s1,char \*s2) {//拼接s1和s2

static char result[10];

strcpy(result,s1);

strcat(result,s2);

return result;

}

char \*newAlias() {

static int no=1;

char s[10];

itoa(no++,s,10);

return strcat0("v",s);

}

char \*newLabel() {

static int no=1;

char s[10];

itoa(no++,s,10);

return strcat0("label",s);

}

char \*newTemp(){

static int no=1;

char s[10];

itoa(no++,s,10);

return strcat0("temp",s);

}

//codenode结构体在def.h中定义，代表三地址TAC代码结点

//生成一条TAC代码的结点组成的双向循环链表，返回头指针

struct codenode \*genIR(int op,struct opn opn1,struct opn opn2,struct opn result){

struct codenode \*h=(struct codenode \*)malloc(sizeof(struct codenode));

h->op=op;

h->opn1=opn1;

h->opn2=opn2;

h->result=result;

h->next=h->prior=h;

return h;

}

//生成一条标号语句，返回头指针

struct codenode \*genLabel(char \*label){

struct codenode \*h=(struct codenode \*)malloc(sizeof(struct codenode));

h->op=LABEL;

strcpy(h->result.id,label);

h->next=h->prior=h;

return h;

}

//生成GOTO语句，返回头指针

struct codenode \*genGoto(char \*label){

struct codenode \*h=(struct codenode \*)malloc(sizeof(struct codenode));

h->op=GOTO;

strcpy(h->result.id,label);

h->next=h->prior=h;

return h;

}

//合并多个中间代码的双向循环链表，首尾相连

struct codenode \*merge(int num,...){

struct codenode \*h1,\*h2,\*p,\*t1,\*t2;

va\_list ap;

va\_start(ap,num);

h1=va\_arg(ap,struct codenode \*);

while (--num>0) {

h2=va\_arg(ap,struct codenode \*);

if (h1==NULL) h1=h2;

else if (h2){

t1=h1->prior;

t2=h2->prior;

t1->next=h2;

t2->next=h1;

h1->prior=t2;

h2->prior=t1;

}

}

va\_end(ap);

return h1;

}

//输出中间代码

void prnIR(struct codenode \*head){

char opnstr1[32],opnstr2[32],resultstr[32];

struct codenode \*h=head;

do{

if (h->opn1.kind==INT)

sprintf(opnstr1,"#%d",h->opn1.const\_int);

if (h->opn1.kind==FLOAT)

sprintf(opnstr1,"#%f",h->opn1.const\_float);

if (h->opn1.kind==CHAR)

sprintf(opnstr1,"#%d",h->opn1.const\_char);

if (h->opn1.kind==ID)

sprintf(opnstr1,"%s",h->opn1.id);

if (h->opn2.kind==INT)

sprintf(opnstr2,"#%d",h->opn2.const\_int);

if (h->opn2.kind==FLOAT)

sprintf(opnstr2,"#%f",h->opn2.const\_float);

if (h->opn2.kind==CHAR)

sprintf(opnstr2,"#%d",h->opn2.const\_char);

if (h->opn2.kind==ID)

sprintf(opnstr2,"%s",h->opn2.id);

sprintf(resultstr,"%s",h->result.id);

switch (h->op) {

case ASSIGNOP:

printf(" %s := %s\n",resultstr,opnstr1);

break;

case INC:

case DEC:

printf(" %s := %s %c %c\n",resultstr,opnstr1, \

h->op==INC ? '+' : h->op==DEC ? '-' : '\\','1');

break;

case PLUS:

case MINUS:

case STAR:

case DIV:

if(h->op!=COMP\_PLUS&&h->op!=COMP\_MINUS)

printf(" %s := %s %c %s\n",resultstr,opnstr1, \

h->op==PLUS ? '+' : h->op==MINUS ? '-' : h->op==STAR ? '\*' : '\\',opnstr2);

else printf(" %s := %s %s %s\n",resultstr,opnstr1, \

h->op==COMP\_PLUS ? "+=" : h->op==COMP\_MINUS ? "-=" : "\\",opnstr2);

break;

case FUNCTION:

printf("\nFUNCTION %s :\n",h->result.id);

break;

case PARAM:

printf(" PARAM %s\n",h->result.id);

break;

case LABEL:

printf("LABEL %s :\n",h->result.id);

break;

case GOTO:

printf(" GOTO %s\n",h->result.id);

break;

case JLE:

printf(" IF %s <= %s GOTO %s\n",opnstr1,opnstr2,resultstr);

break;

case JLT:

printf(" IF %s < %s GOTO %s\n",opnstr1,opnstr2,resultstr);

break;

case JGE:

printf(" IF %s >= %s GOTO %s\n",opnstr1,opnstr2,resultstr);

break;

case JGT:

printf(" IF %s > %s GOTO %s\n",opnstr1,opnstr2,resultstr);

break;

case EQ:

printf(" IF %s == %s GOTO %s\n",opnstr1,opnstr2,resultstr);

break;

case NEQ:

printf(" IF %s != %s GOTO %s\n",opnstr1,opnstr2,resultstr);

break;

case ARG://表示实参

printf(" ARG %s\n",h->result.id);

break;

case CALL://调用函数

printf(" %s := CALL %s\n",resultstr, opnstr1);

break;

case RETURN:

if (h->result.kind)

printf(" RETURN %s\n",resultstr);

else

printf(" RETURN\n");

break;

}

h=h->next;

} while (h!=head);

}

void semantic\_error(int line,char \*msg1,char \*msg2){//这里可以只收集错误信息，最后在一次显示

printf("在%d行,%s %s\n",line,msg1,msg2);

}

//symbolTable在def.h中定义，代表符号表

void prn\_symbol(){ //显示符号表

int i=0;

printf("%6s %6s %6s %6s %4s %6s\n","变量名","别 名","层 号","类 型","标记","偏移量");

for(i=0;i<symbolTable.index;i++){

if(symbolTable.symbols[i].type==INT)

printf("%6s %6s %6d %6s %4c %6d\n",symbolTable.symbols[i].name,\

symbolTable.symbols[i].alias,symbolTable.symbols[i].level,\

"int",\

symbolTable.symbols[i].flag,symbolTable.symbols[i].offset);

else if(symbolTable.symbols[i].type==FLOAT)

printf("%6s %6s %6d %6s %4c %6d\n",symbolTable.symbols[i].name,\

symbolTable.symbols[i].alias,symbolTable.symbols[i].level,\

"float",\

symbolTable.symbols[i].flag,symbolTable.symbols[i].offset);

else if(symbolTable.symbols[i].type==CHAR)

printf("%6s %6s %6d %6s %4c %6d\n",symbolTable.symbols[i].name,\

symbolTable.symbols[i].alias,symbolTable.symbols[i].level,\

"char",\

symbolTable.symbols[i].flag,symbolTable.symbols[i].offset);

}

}

int searchSymbolTable(char \*name) {

int i;

for(i=symbolTable.index-1;i>=0;i--)

if (!strcmp(symbolTable.symbols[i].name, name)) return i;

return -1;

}

//填符号表

int fillSymbolTable(char \*name,char \*alias,int level,int type,char flag,int offset) {

//首先根据name查符号表，不能重复定义 重复定义返回-1

int i;

/\*符号查重，考虑外部变量声明前有函数定义，

其形参名还在符号表中，这时的外部变量与前函数的形参重名是允许的\*/

for(i=symbolTable.index-1;symbolTable.symbols[i].level==level||(level==0 && i>=0);i--) {

if (level==0 && symbolTable.symbols[i].level==1) continue; //外部变量和形参不必比较重名

if (!strcmp(symbolTable.symbols[i].name, name)) return -1;

}

//填写符号表内容

strcpy(symbolTable.symbols[symbolTable.index].name,name);

strcpy(symbolTable.symbols[symbolTable.index].alias,alias);

symbolTable.symbols[symbolTable.index].level=level;

symbolTable.symbols[symbolTable.index].type=type;

symbolTable.symbols[symbolTable.index].flag=flag;

symbolTable.symbols[symbolTable.index].offset=offset;

return symbolTable.index++; //返回的是符号在符号表中的位置序号，中间代码生成时可用序号取到符号别名

}

//填写临时变量到符号表，返回临时变量在符号表中的位置

int fill\_Temp(char \*name,int level,int type,char flag,int offset) {

strcpy(symbolTable.symbols[symbolTable.index].name,"");

strcpy(symbolTable.symbols[symbolTable.index].alias,name);

symbolTable.symbols[symbolTable.index].level=level;

symbolTable.symbols[symbolTable.index].type=type;

symbolTable.symbols[symbolTable.index].flag=flag;

symbolTable.symbols[symbolTable.index].offset=offset;

return symbolTable.index++; //返回的是临时变量在符号表中的位置序号

}

int LEV=0; //层号

int func\_size; //1个函数的活动记录大小

void ext\_var\_list(struct node \*T){//处理变量列表

int rtn,num=1;

switch (T->kind){

case EXT\_DEC\_LIST:

T->ptr[0]->type=T->type; //将类型属性向下传递变量结点

T->ptr[0]->offset=T->offset; //外部变量的偏移量向下传递

T->ptr[1]->type=T->type; //将类型属性向下传递变量结点

T->ptr[1]->offset=T->offset+T->width; //外部变量的偏移量向下传递

T->ptr[1]->width=T->width;

ext\_var\_list(T->ptr[0]);

ext\_var\_list(T->ptr[1]);

T->num=T->ptr[1]->num+1;

break;

case ID:

rtn=fillSymbolTable(T->type\_id,newAlias(),LEV,T->type,'V',T->offset); //最后一个变量名

if (rtn==-1)

semantic\_error(T->pos,T->type\_id, "ERROR 1：变量重复定义");

else T->place=rtn;

T->num=1;

break;

}

}

int match\_param(int i,struct node \*T,int lastpos){

int j,num=symbolTable.symbols[i].paramnum;

int type1,type2;

if (num==0 && T==NULL) return 1;

for (j=1;j<num;j++) {

if (!T){

semantic\_error(lastpos,"", "ERROR 2：函数调用参数太少");

return 0;

}

type1=symbolTable.symbols[i+j].type; //形参类型

type2=T->ptr[0]->type;

if (type1!=type2){

semantic\_error(T->pos,"", "ERROR 3：参数类型不匹配");

return 0;

}

lastpos=T->pos;

T=T->ptr[1];

}

if (T->ptr[1]){ //num个参数已经匹配完，还有实参表达式

semantic\_error(T->pos,"", "ERROR 4：函数调用参数太多");

return 0;

}

return 1;

}

void boolExp(struct node \*T){//布尔表达式，参考文献[2]p84的思想

struct opn opn1,opn2,result;

int op;

int rtn;

if (T){

switch (T->kind) {

case INT:

if (T->type\_int!=0)

T->code=genGoto(T->Etrue);

else T->code=genGoto(T->Efalse);

T->width=0;

break;

case FLOAT:

if (T->type\_float!=0.0)

T->code=genGoto(T->Etrue);

else T->code=genGoto(T->Efalse);

T->width=0;

break;

case CHAR:

if (T->type\_char!=0)

T->code=genGoto(T->Etrue);

else T->code=genGoto(T->Efalse);

T->width=0;

break;

case ID:

//查符号表，获得符号表中的位置，类型送type

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

if (rtn==-1)

semantic\_error(T->pos,T->type\_id, "ERROR 5：变量未定义");

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

semantic\_error(T->pos,T->type\_id, "ERROR 6：是函数名，类型不匹配");

else {

opn1.kind=ID; strcpy(opn1.id,symbolTable.symbols[rtn].alias);

opn1.offset=symbolTable.symbols[rtn].offset;

opn2.kind=INT; opn2.const\_int=0;

result.kind=ID; strcpy(result.id,T->Etrue);

T->code=genIR(NEQ,opn1,opn2,result);

T->code=merge(2,T->code,genGoto(T->Efalse));

}

T->width=0;

break;

case RELOP: //处理关系运算表达式,2个操作数都按基本表达式处理

T->ptr[0]->offset=T->ptr[1]->offset=T->offset;

Exp(T->ptr[0]);

T->width=T->ptr[0]->width;

Exp(T->ptr[1]);

if(T->ptr[0]->type!=T->ptr[1]->type)

semantic\_error(T->pos,"", "ERROR 7：操作数类型不一致");

if (T->width<T->ptr[1]->width) T->width=T->ptr[1]->width;

opn1.kind=ID; strcpy(opn1.id,symbolTable.symbols[T->ptr[0]->place].alias);

opn1.offset=symbolTable.symbols[T->ptr[0]->place].offset;

opn2.kind=ID; strcpy(opn2.id,symbolTable.symbols[T->ptr[1]->place].alias);

opn2.offset=symbolTable.symbols[T->ptr[1]->place].offset;

result.kind=ID; strcpy(result.id,T->Etrue);

if (strcmp(T->type\_id,"<")==0)

op=JLT;

else if (strcmp(T->type\_id,"<=")==0)

op=JLE;

else if (strcmp(T->type\_id,">")==0)

op=JGT;

else if (strcmp(T->type\_id,">=")==0)

op=JGE;

else if (strcmp(T->type\_id,"==")==0)

op=EQ;

else if (strcmp(T->type\_id,"!=")==0)

op=NEQ;

T->code=genIR(op,opn1,opn2,result);

T->code=merge(4,T->ptr[0]->code,T->ptr[1]->code,T->code,genGoto(T->Efalse));

break;

case AND:

case OR:

if (T->kind==AND) {

strcpy(T->ptr[0]->Etrue,newLabel());

strcpy(T->ptr[0]->Efalse,T->Efalse);

}

else {

strcpy(T->ptr[0]->Etrue,T->Etrue);

strcpy(T->ptr[0]->Efalse,newLabel());

}

strcpy(T->ptr[1]->Etrue,T->Etrue);

strcpy(T->ptr[1]->Efalse,T->Efalse);

T->ptr[0]->offset=T->ptr[1]->offset=T->offset;

boolExp(T->ptr[0]);

T->width=T->ptr[0]->width;

boolExp(T->ptr[1]);

if (T->width<T->ptr[1]->width) T->width=T->ptr[1]->width;

if (T->kind==AND)

T->code=merge(3,T->ptr[0]->code,genLabel(T->ptr[0]->Etrue),T->ptr[1]->code);

else

T->code=merge(3,T->ptr[0]->code,genLabel(T->ptr[0]->Efalse),T->ptr[1]->code);

break;

case NOT:

strcpy(T->ptr[0]->Etrue,T->Efalse);

strcpy(T->ptr[0]->Efalse,T->Etrue);

boolExp(T->ptr[0]);

T->code=T->ptr[0]->code;

break;

}

}

}

void Exp(struct node \*T)

{//处理基本表达式，参考文献[2]p82的思想

int rtn,num,width;

struct node \*T0;

struct opn opn1,opn2,result;

if (T){

switch (T->kind) {

case ID:

//查符号表，获得符号表中的位置，类型送type

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

if (rtn==-1)

semantic\_error(T->pos,T->type\_id, "ERROR 5：变量未定义");

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

semantic\_error(T->pos,T->type\_id, "ERROR 6：是函数名，类型不匹配");

else {

T->place=rtn; //结点保存变量在符号表中的位置

T->code=NULL; //标识符不需要生成TAC

T->type=symbolTable.symbols[rtn].type;

T->offset=symbolTable.symbols[rtn].offset;

T->width=0; //未再使用新单元

}

break;

case INT:

T->place=fill\_Temp(newTemp(),LEV,T->type,'T',T->offset);//为整常量生成一个临时变量

T->type=INT;

opn1.kind=INT;opn1.const\_int=T->type\_int;

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

result.offset=symbolTable.symbols[T->place].offset;

T->code=genIR(ASSIGNOP,opn1,opn2,result);

T->width=4;

break;

case FLOAT:

T->place=fill\_Temp(newTemp(),LEV,T->type,'T',T->offset);//为浮点常量生成一个临时变量

T->type=FLOAT;

opn1.kind=FLOAT; opn1.const\_float=T->type\_float;

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

result.offset=symbolTable.symbols[T->place].offset;

T->code=genIR(ASSIGNOP,opn1,opn2,result);

T->width=4;

break;

case CHAR:

T->place=fill\_Temp(newTemp(),LEV,T->type,'T',T->offset);//为字符常量生成一个临时变量

T->type=CHAR;

opn1.kind=CHAR;opn1.const\_char=T->type\_char;

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

result.offset=symbolTable.symbols[T->place].offset;

T->code=genIR(ASSIGNOP,opn1,opn2,result);

T->width=1;

break;

case ASSIGNOP:

if (T->ptr[0]->kind!=ID){

semantic\_error(T->pos,"", "ERROR 8：赋值语句必须为左值");

}

else {

Exp(T->ptr[0]); //处理左值，例中仅为变量

T->ptr[1]->offset=T->offset;

Exp(T->ptr[1]);

T->type=T->ptr[0]->type;

T->width=T->ptr[1]->width;

T->code=merge(2,T->ptr[0]->code,T->ptr[1]->code);

opn1.kind=ID; strcpy(opn1.id,symbolTable.symbols[T->ptr[1]->place].alias);//右值一定是个变量或临时变量

opn1.offset=symbolTable.symbols[T->ptr[1]->place].offset;

result.kind=ID; strcpy(result.id,symbolTable.symbols[T->ptr[0]->place].alias);

result.offset=symbolTable.symbols[T->ptr[0]->place].offset;

T->code=merge(2,T->code,genIR(ASSIGNOP,opn1,opn2,result));

}

break;

case AND: //按算术表达式方式计算布尔值，未写完

case OR: //按算术表达式方式计算布尔值，未写完

case RELOP: //按算术表达式方式计算布尔值，未写完

T->type=INT;

T->ptr[0]->offset=T->ptr[1]->offset=T->offset;

Exp(T->ptr[0]);

Exp(T->ptr[1]);

break;

case COMP\_PLUS:

case COMP\_MINUS:

printf("1111\n");

T->ptr[0]->offset=T->offset;

Exp(T->ptr[0]);

T->ptr[1]->offset=T->offset+T->ptr[0]->width;

Exp(T->ptr[1]);

//判断T->ptr[0]，T->ptr[1]类型是否正确，可能根据运算符生成不同形式的代码，给T的type赋值

//下面的类型属性计算，没有考虑错误处理情况

if (T->ptr[0]->type==FLOAT || T->ptr[1]->type==FLOAT)

T->type=FLOAT,T->width=T->ptr[0]->width+T->ptr[1]->width+4;

else if(T->ptr[0]->type==INT || T->ptr[1]->type==INT)

T->type=INT,T->width=T->ptr[0]->width+T->ptr[1]->width+2;

else if(T->ptr[0]->type==CHAR || T->ptr[1]->type==CHAR)

T->type=CHAR,T->width=T->ptr[0]->width+T->ptr[1]->width+1;

T->place=fill\_Temp(newTemp(),LEV,T->type,'T',T->offset+T->ptr[0]->width+T->ptr[1]->width);

opn1.kind=ID; strcpy(opn1.id,symbolTable.symbols[T->ptr[0]->place].alias);

opn1.type=T->ptr[0]->type;opn1.offset=symbolTable.symbols[T->ptr[0]->place].offset;

opn2.kind=ID; strcpy(opn2.id,symbolTable.symbols[T->ptr[1]->place].alias);

opn2.type=T->ptr[1]->type;opn2.offset=symbolTable.symbols[T->ptr[1]->place].offset;

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

result.type=T->type;result.offset=symbolTable.symbols[T->place].offset;

if(T->kind==COMP\_PLUS)

T->code=merge(3,T->ptr[0]->code,T->ptr[1]->code,genIR(PLUS,opn1,opn2,result));

else T->code=merge(3,T->ptr[0]->code,T->ptr[1]->code,genIR(MINUS,opn1,opn2,result));

if(T->type==INT)

T->width=T->ptr[0]->width+T->ptr[1]->width+4;

else if(T->type==FLOAT)

T->width=T->ptr[0]->width+T->ptr[1]->width+8;

else if(T->type==CHAR)

T->width=T->ptr[0]->width+T->ptr[1]->width+1;

//T->code=merge(2,T->ptr[0]->code,T->ptr[1]->code);

opn1.kind=ID; strcpy(opn1.id,symbolTable.symbols[T->place].alias);//右值一定是个变量或临时变量

opn1.offset=symbolTable.symbols[T->place].offset;

result.kind=ID; strcpy(result.id,symbolTable.symbols[T->ptr[0]->place].alias);

result.offset=symbolTable.symbols[T->ptr[0]->place].offset;

T->code=merge(2,T->code,genIR(ASSIGNOP,opn1,opn2,result));

break;

case PLUS:

case MINUS:

case STAR:

case DIV:

T->ptr[0]->offset=T->offset;

Exp(T->ptr[0]);

T->ptr[1]->offset=T->offset+T->ptr[0]->width;

Exp(T->ptr[1]);

//判断T->ptr[0]，T->ptr[1]类型是否正确，可能根据运算符生成不同形式的代码，给T的type赋值

//下面的类型属性计算，没有考虑错误处理情况

if(T->kind==DIV&&T->ptr[1]->type\_int==0)

{semantic\_error(T->pos,"", "ERROR 9：除数不能为0");break;}

if (T->ptr[0]->type==FLOAT || T->ptr[1]->type==FLOAT)

T->type=FLOAT,T->width=T->ptr[0]->width+T->ptr[1]->width+4;

else if(T->ptr[0]->type==INT || T->ptr[1]->type==INT)

T->type=INT,T->width=T->ptr[0]->width+T->ptr[1]->width+2;

else if(T->ptr[0]->type==CHAR || T->ptr[1]->type==CHAR)

T->type=CHAR,T->width=T->ptr[0]->width+T->ptr[1]->width+1;

T->place=fill\_Temp(newTemp(),LEV,T->type,'T',T->offset+T->ptr[0]->width+T->ptr[1]->width);

opn1.kind=ID; strcpy(opn1.id,symbolTable.symbols[T->ptr[0]->place].alias);

opn1.type=T->ptr[0]->type;opn1.offset=symbolTable.symbols[T->ptr[0]->place].offset;

opn2.kind=ID; strcpy(opn2.id,symbolTable.symbols[T->ptr[1]->place].alias);

opn2.type=T->ptr[1]->type;opn2.offset=symbolTable.symbols[T->ptr[1]->place].offset;

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

result.type=T->type;result.offset=symbolTable.symbols[T->place].offset;

T->code=merge(3,T->ptr[0]->code,T->ptr[1]->code,genIR(T->kind,opn1,opn2,result));

if(T->type==INT)

T->width=T->ptr[0]->width+T->ptr[1]->width+4;

else if(T->type==FLOAT)

T->width=T->ptr[0]->width+T->ptr[1]->width+8;

else if(T->type==CHAR)

T->width=T->ptr[0]->width+T->ptr[1]->width+1;

break;

case INC: //未写完整

case DEC: //未写完整

case NOT: //未写完整

case UMINUS://未写完整

rtn = searchSymbolTable(T->ptr[0]->type\_id);

if(symbolTable.symbols[rtn].type!= INT&&(T->kind==INC||T->kind==DEC))

{

semantic\_error(T->pos,T->ptr[0]->type\_id, "ERROR 10：自增自减的操作数必须是整形变量");break;

}

T->ptr[0]->offset=T->offset;

Exp(T->ptr[0]);

//判断T->ptr[0]，T->ptr[1]类型是否正确，可能根据运算符生成不同形式的代码，给T的type赋值

//下面的类型属性计算，没有考虑错误处理情况

if (T->ptr[0]->type==FLOAT)

T->type=FLOAT,T->width=T->ptr[0]->width+4;

else if(T->ptr[0]->type==INT)

T->type=INT,T->width=T->ptr[0]->width+2;

else if(T->ptr[0]->type==CHAR)

T->type=CHAR,T->width=T->ptr[0]->width+1;

int temp=fill\_Temp(newTemp(),LEV,T->type,'T',T->offset+T->ptr[0]->width);

opn1.kind=ID; strcpy(opn1.id,symbolTable.symbols[T->ptr[0]->place].alias);//右值一定是个变量或临时变量

opn1.offset=symbolTable.symbols[T->ptr[0]->place].offset;

result.kind=ID; strcpy(result.id,symbolTable.symbols[temp].alias);

result.offset=symbolTable.symbols[temp].offset;

T->code=merge(2,T->code,genIR(ASSIGNOP,opn1,opn2,result));

T->place=fill\_Temp(newTemp(),LEV,T->type,'T',T->offset+T->ptr[0]->width);

opn1.kind=ID; strcpy(opn1.id,symbolTable.symbols[T->ptr[0]->place].alias);

opn1.type=T->ptr[0]->type;opn1.offset=symbolTable.symbols[T->ptr[0]->place].offset;

opn2.kind=INT; opn2.const\_int=1;

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

result.type=T->type;result.offset=symbolTable.symbols[T->place].offset;

if(T->kind==INC){

//opn2.kind=INT; opn2.const\_int=1;

T->code=merge(3,T->code,T->ptr[0]->code,genIR(PLUS,opn1,opn2,result));

}

else {

//opn2.kind=INT; opn2.const\_int=;

T->code=merge(3,T->code,T->ptr[0]->code,genIR(MINUS,opn1,opn2,result));

}

if(T->type==INT)

T->width=T->ptr[0]->width+4;

else if(T->type==FLOAT)

T->width=T->ptr[0]->width+8;

else if(T->type==CHAR)

T->width=T->ptr[0]->width+1;

opn1.kind=ID; strcpy(opn1.id,symbolTable.symbols[T->place].alias);//右值一定是个变量或临时变量

opn1.offset=symbolTable.symbols[T->place].offset;

result.kind=ID; strcpy(result.id,symbolTable.symbols[T->ptr[0]->place].alias);

result.offset=symbolTable.symbols[T->ptr[0]->place].offset;

T->code=merge(2,T->code,genIR(ASSIGNOP,opn1,opn2,result));

if(!strcmp(T->type\_id,"POST\_INC")||!strcmp(T->type\_id,"POST\_DEC"))

T->place=temp;

break;

case FUNC\_CALL: //根据T->type\_id查出函数的定义，如果语言中增加了实验教材的read，write需要单独处理一下

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

if (rtn==-1){

semantic\_error(T->pos,T->type\_id, "ERROR 11：函数未定义");

break;

}

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

semantic\_error(T->pos,T->type\_id, "ERROR 12：不是一个函数");

break;

}

T->type=symbolTable.symbols[rtn].type;

if(T->type==INT)//存放函数返回值的单数字节数

width=4;

if(T->type==FLOAT)

width=8;

if(T->type==CHAR)

width=1;

if (T->ptr[0]){

T->ptr[0]->offset=T->offset;

Exp(T->ptr[0]); //处理所有实参表达式求值，及类型

T->width=T->ptr[0]->width+width; //累加上计算实参使用临时变量的单元数

T->code=T->ptr[0]->code;

}

else {T->width=width; T->code=NULL;}

match\_param(rtn,T->ptr[0],T->pos); //处理所以参数的匹配

//处理参数列表的中间代码

T0=T->ptr[0];

while (T0) {

result.kind=ID; strcpy(result.id,symbolTable.symbols[T0->ptr[0]->place].alias);

result.offset=symbolTable.symbols[T0->ptr[0]->place].offset;

T->code=merge(2,T->code,genIR(ARG,opn1,opn2,result));

T0=T0->ptr[1];

}

T->place=fill\_Temp(newTemp(),LEV,T->type,'T',T->offset+T->width-width);

opn1.kind=ID; strcpy(opn1.id,T->type\_id); //保存函数名

opn1.offset=rtn; //这里offset用以保存函数定义入口,在目标代码生成时，能获取相应信息

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

result.offset=symbolTable.symbols[T->place].offset;

T->code=merge(2,T->code,genIR(CALL,opn1,opn2,result)); //生成函数调用中间代码

break;

case ARGS: //此处仅处理各实参表达式的求值的代码序列，不生成ARG的实参系列

T->ptr[0]->offset=T->offset;

Exp(T->ptr[0]);

T->width=T->ptr[0]->width;

T->code=T->ptr[0]->code;

if (T->ptr[1]) {

T->ptr[1]->offset=T->offset+T->ptr[0]->width;

Exp(T->ptr[1]);

T->width+=T->ptr[1]->width;

T->code=merge(2,T->code,T->ptr[1]->code);

}

break;

}

}

}

void semantic\_Analysis(struct node \*T)

{//对抽象语法树的先根遍历,按display的控制结构修改完成符号表管理和语义检查和TAC生成（语句部分）

int rtn,num,width;

struct node \*T0;

struct opn opn1,opn2,result;

if (T){

switch (T->kind) {

case EXT\_DEF\_LIST:

if (!T->ptr[0]) break;

T->ptr[0]->offset=T->offset;

semantic\_Analysis(T->ptr[0]); //访问外部定义列表中的第一个

T->code=T->ptr[0]->code;

if (T->ptr[1]){

T->ptr[1]->offset=T->ptr[0]->offset + T->ptr[0]->width;

semantic\_Analysis(T->ptr[1]); //访问该外部定义列表中的其它外部定义

T->code=merge(2,T->code,T->ptr[1]->code);

}

break;

case EXT\_VAR\_DEF: //处理外部说明,将第一个孩子(TYPE结点)中的类型送到第二个孩子的类型域

if(!strcmp(T->ptr[0]->type\_id,"int"))

T->type=T->ptr[1]->type=INT;

else if(!strcmp(T->ptr[0]->type\_id,"float"))

T->type=T->ptr[1]->type=FLOAT;

else if(!strcmp(T->ptr[0]->type\_id,"char"))

T->type=T->ptr[1]->type=CHAR;

T->ptr[1]->offset=T->offset; //这个外部变量的偏移量向下传递

//将一个变量的宽度向下传递

if(T->type==INT)

T->ptr[1]->width=4;

else if(T->type==FLOAT)

T->ptr[1]->width=8;

else if(T->type==CHAR)

T->ptr[1]->width=1;

ext\_var\_list(T->ptr[1]); //处理外部变量说明中的标识符序列

//计算这个外部变量说明的宽度

if(T->type==INT)

T->width=4\*T->ptr[1]->num;

else if(T->type==FLOAT)

T->width=8\*T->ptr[1]->num;

else if(T->type==CHAR)

T->width=1\*T->ptr[1]->num;

T->code=NULL; //这里假定外部变量不支持初始化

break;

case FUNC\_DEF: //填写函数定义信息到符号表

//获取函数返回类型送到含函数名、参数的结点

if(!strcmp(T->ptr[0]->type\_id,"int"))

T->ptr[1]->type=INT;

if(!strcmp(T->ptr[0]->type\_id,"float"))

T->ptr[1]->type=FLOAT;

if(!strcmp(T->ptr[0]->type\_id,"char"))

T->ptr[1]->type=CHAR;

T->width=0; //函数的宽度设置为0，不会对外部变量的地址分配产生影响

T->offset=DX; //设置局部变量在活动记录中的偏移量初值

semantic\_Analysis(T->ptr[1]); //处理函数名和参数结点部分，这里不考虑用寄存器传递参数

T->offset+=T->ptr[1]->width; //用形参单元宽度修改函数局部变量的起始偏移量

T->ptr[2]->offset=T->offset;

strcpy(T->ptr[2]->Snext,newLabel()); //函数体语句执行结束后的位置属性

semantic\_Analysis(T->ptr[2]); //处理函数体结点

//计算活动记录大小,这里offset属性存放的是活动记录大小，不是偏移

symbolTable.symbols[T->ptr[1]->place].offset=T->offset+T->ptr[2]->width;

T->code=merge(3,T->ptr[1]->code,T->ptr[2]->code,genLabel(T->ptr[2]->Snext));//函数体的代码作为函数的代码

break;

case FUNC\_DEC: //根据返回类型，函数名填写符号表

rtn=fillSymbolTable(T->type\_id,newAlias(),LEV,T->type,'F',0);//函数不在数据区中分配单元，偏移量为0

if (rtn==-1){

semantic\_error(T->pos,T->type\_id, "ERROR 13：函数重复定义");

break;

}

else T->place=rtn;

result.kind=ID; strcpy(result.id,T->type\_id);

result.offset=rtn;

T->code=genIR(FUNCTION,opn1,opn2,result); //生成中间代码：FUNCTION 函数名

T->offset=DX; //设置形式参数在活动记录中的偏移量初值

if (T->ptr[0]) { //判断是否有参数

T->ptr[0]->offset=T->offset;

semantic\_Analysis(T->ptr[0]); //处理函数参数列表

T->width=T->ptr[0]->width;

symbolTable.symbols[rtn].paramnum=T->ptr[0]->num;

T->code=merge(2,T->code,T->ptr[0]->code); //连接函数名和参数代码序列

}

else symbolTable.symbols[rtn].paramnum=0,T->width=0;

break;

case PARAM\_LIST: //处理函数形式参数列表

T->ptr[0]->offset=T->offset;

semantic\_Analysis(T->ptr[0]);

if (T->ptr[1]){

T->ptr[1]->offset=T->offset+T->ptr[0]->width;

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

T->num=T->ptr[0]->num+T->ptr[1]->num; //统计参数个数

T->width=T->ptr[0]->width+T->ptr[1]->width; //累加参数单元宽度

T->code=merge(2,T->ptr[0]->code,T->ptr[1]->code); //连接参数代码

}

else {

T->num=T->ptr[0]->num;

T->width=T->ptr[0]->width;

T->code=T->ptr[0]->code;

}

break;

case PARAM\_DEC:

rtn=fillSymbolTable(T->ptr[1]->type\_id,newAlias(),1,T->ptr[0]->type,'P',T->offset);

if (rtn==-1)

semantic\_error(T->ptr[1]->pos,T->ptr[1]->type\_id, "ERROR 14：参数名重复定义");

else T->ptr[1]->place=rtn;

T->num=1; //参数个数计算的初始值

if(T->ptr[0]->type==INT)//参数宽度

T->width=4;

else if(T->ptr[0]->type==FLOAT)

T->width=8;

else if(T->ptr[0]->type==CHAR)

T->width=1;

result.kind=ID; strcpy(result.id, symbolTable.symbols[rtn].alias);

result.offset=T->offset;

T->code=genIR(PARAM,opn1,opn2,result); //生成：FUNCTION 函数名

break;

case COMP\_STM:

LEV++;

//设置层号加1，并且保存该层局部变量在符号表中的起始位置在symbol\_scope\_TX

symbol\_scope\_TX.TX[symbol\_scope\_TX.top++]=symbolTable.index;

T->width=0;

T->code=NULL;

if (T->ptr[0]) {

T->ptr[0]->offset=T->offset;

semantic\_Analysis(T->ptr[0]); //处理该层的局部变量DEF\_LIST

T->width+=T->ptr[0]->width;

T->code=T->ptr[0]->code;

}

if (T->ptr[1]){

T->ptr[1]->offset=T->offset+T->width;

strcpy(T->ptr[1]->Snext,T->Snext); //S.next属性向下传递

semantic\_Analysis(T->ptr[1]); //处理复合语句的语句序列

T->width+=T->ptr[1]->width;

T->code=merge(2,T->code,T->ptr[1]->code);

}

prn\_symbol(); //c在退出一个符合语句前显示的符号表

LEV--; //出复合语句，层号减1

symbolTable.index=symbol\_scope\_TX.TX[--symbol\_scope\_TX.top]; //删除该作用域中的符号

break;

case DEF\_LIST:

T->code=NULL;

if (T->ptr[0]){

T->ptr[0]->offset=T->offset;

semantic\_Analysis(T->ptr[0]); //处理一个局部变量定义

T->code=T->ptr[0]->code;

T->width=T->ptr[0]->width;

}

if (T->ptr[1]) {

T->ptr[1]->offset=T->offset+T->ptr[0]->width;

semantic\_Analysis(T->ptr[1]); //处理剩下的局部变量定义

T->code=merge(2,T->code,T->ptr[1]->code);

T->width+=T->ptr[1]->width;

}

break;

case VAR\_DEF://处理一个局部变量定义,将第一个孩子(TYPE结点)中的类型送到第二个孩子的类型域

//类似于上面的外部变量EXT\_VAR\_DEF，换了一种处理方法

T->code=NULL;

//确定变量序列各变量类型

if(!strcmp(T->ptr[0]->type\_id,"int"))

T->ptr[1]->type=INT;

else if(!strcmp(T->ptr[0]->type\_id,"float"))

T->ptr[1]->type=FLOAT;

else if(!strcmp(T->ptr[0]->type\_id,"char"))

T->ptr[1]->type=CHAR;

T0=T->ptr[1]; //T0为变量名列表子树根指针，对ID、ASSIGNOP类结点在登记到符号表，作为局部变量

num=0;

T0->offset=T->offset;

T->width=0;

//一个变量宽度

if(T->ptr[1]->type==INT)

width=4;

else if(T->ptr[1]->type==FLOAT)

width=8;

else if(T->ptr[1]->type==CHAR)

width=1;

while (T0) { //处理所有DEC\_LIST结点

num++;

T0->ptr[0]->type=T0->type; //类型属性向下传递

if (T0->ptr[1]) T0->ptr[1]->type=T0->type;

T0->ptr[0]->offset=T0->offset; //类型属性向下传递

if (T0->ptr[1]) T0->ptr[1]->offset=T0->offset+width;

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

rtn=fillSymbolTable(T0->ptr[0]->type\_id,newAlias(),LEV,T0->ptr[0]->type,'V',T->offset+T->width);//此处偏移量未计算，暂时为0

if (rtn==-1)

semantic\_error(T0->ptr[0]->pos,T0->ptr[0]->type\_id, "ERROR 1：变量重复定义");

else T0->ptr[0]->place=rtn;

T->width+=width;

}

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

rtn=fillSymbolTable(T0->ptr[0]->ptr[0]->type\_id,newAlias(),LEV,T0->ptr[0]->type,'V',T->offset+T->width);//此处偏移量未计算，暂时为0

if (rtn==-1)

semantic\_error(T0->ptr[0]->ptr[0]->pos,T0->ptr[0]->ptr[0]->type\_id, "ERROR 1：变量重复定义");

else {

T0->ptr[0]->place=rtn;

T0->ptr[0]->ptr[1]->offset=T->offset+T->width+width;

Exp(T0->ptr[0]->ptr[1]);

opn1.kind=ID; strcpy(opn1.id,symbolTable.symbols[T0->ptr[0]->ptr[1]->place].alias);

result.kind=ID; strcpy(result.id,symbolTable.symbols[T0->ptr[0]->place].alias);

T->code=merge(3,T->code,T0->ptr[0]->ptr[1]->code,genIR(ASSIGNOP,opn1,opn2,result));

}

T->width+=width+T0->ptr[0]->ptr[1]->width;

}

T0=T0->ptr[1];

}

break;

case RETURN:

if (T->ptr[0]){

T->ptr[0]->offset=T->offset;

Exp(T->ptr[0]);

num=symbolTable.index;

do num--; while (symbolTable.symbols[num].flag!='F');

if (T->ptr[0]->type!=symbolTable.symbols[num].type) {

semantic\_error(T->pos, "ERROR 15：返回值类型不匹配","");

T->width=0;T->code=NULL;

break;

}

T->width=T->ptr[0]->width;

result.kind=ID; strcpy(result.id,symbolTable.symbols[T->ptr[0]->place].alias);

result.offset=symbolTable.symbols[T->ptr[0]->place].offset;

T->code=merge(2,T->ptr[0]->code,genIR(RETURN,opn1,opn2,result));

}

else{

T->width=0;

result.kind=0;

T->code=genIR(RETURN,opn1,opn2,result);

}

break;

case ARRAY\_DEF:

rtn=fillSymbolTable(T->ptr[1]->type\_id,newAlias(),1,T->ptr[0]->type,'P',T->offset);

if (rtn==-1)

semantic\_error(T->ptr[1]->pos,T->ptr[1]->type\_id, "ERROR 16：数组名重复定义");

else T->ptr[1]->place=rtn;

T->num=1; //参数个数计算的初始值

break;

case STM\_LIST:

if (!T->ptr[0]) { T->code=NULL; T->width=0; break;} //空语句序列

if (T->ptr[1]) //2条以上语句连接，生成新标号作为第一条语句结束后到达的位置

strcpy(T->ptr[0]->Snext,newLabel());

else //语句序列仅有一条语句，S.next属性向下传递

strcpy(T->ptr[0]->Snext,T->Snext);

T->ptr[0]->offset=T->offset;

semantic\_Analysis(T->ptr[0]);

T->code=T->ptr[0]->code;

T->width=T->ptr[0]->width;

if (T->ptr[1]){ //2条以上语句连接,S.next属性向下传递

strcpy(T->ptr[1]->Snext,T->Snext);

T->ptr[1]->offset=T->offset; //顺序结构共享单元方式

//T->ptr[1]->offset=T->offset+T->ptr[0]->width; //顺序结构顺序分配单元方式

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

//序列中第1条为表达式语句，返回语句，复合语句时，第2条前不需要标号

if (T->ptr[0]->kind==RETURN ||T->ptr[0]->kind==EXP\_STMT ||T->ptr[0]->kind==COMP\_STM)

T->code=merge(2,T->code,T->ptr[1]->code);

else

T->code=merge(3,T->code,genLabel(T->ptr[0]->Snext),T->ptr[1]->code);

if (T->ptr[1]->width>T->width) T->width=T->ptr[1]->width; //顺序结构共享单元方式

//T->width+=T->ptr[1]->width;//顺序结构顺序分配单元方式

}

break;

case IF\_THEN:

strcpy(T->ptr[0]->Etrue,newLabel()); //设置条件语句真假转移位置

strcpy(T->ptr[0]->Efalse,T->Snext);

T->ptr[0]->offset=T->ptr[1]->offset=T->offset;

boolExp(T->ptr[0]);

T->width=T->ptr[0]->width;

strcpy(T->ptr[1]->Snext,T->Snext);

semantic\_Analysis(T->ptr[1]); //if子句

if (T->width<T->ptr[1]->width) T->width=T->ptr[1]->width;

T->code=merge(3,T->ptr[0]->code, genLabel(T->ptr[0]->Etrue),T->ptr[1]->code);

break; //控制语句都还没有处理offset和width属性

case IF\_THEN\_ELSE:

strcpy(T->ptr[0]->Etrue,newLabel()); //设置条件语句真假转移位置

strcpy(T->ptr[0]->Efalse,newLabel());

T->ptr[0]->offset=T->ptr[1]->offset=T->ptr[2]->offset=T->offset;

boolExp(T->ptr[0]); //条件，要单独按短路代码处理

T->width=T->ptr[0]->width;

strcpy(T->ptr[1]->Snext,T->Snext);

semantic\_Analysis(T->ptr[1]); //if子句

if (T->width<T->ptr[1]->width) T->width=T->ptr[1]->width;

strcpy(T->ptr[2]->Snext,T->Snext);

semantic\_Analysis(T->ptr[2]); //else子句

if (T->width<T->ptr[2]->width) T->width=T->ptr[2]->width;

T->code=merge(6,T->ptr[0]->code,genLabel(T->ptr[0]->Etrue),T->ptr[1]->code,\

genGoto(T->Snext),genLabel(T->ptr[0]->Efalse),T->ptr[2]->code);

break;

case WHILE:

strcpy(T->ptr[0]->Etrue,newLabel()); //子结点继承属性的计算

strcpy(T->ptr[0]->Efalse,T->Snext);

T->ptr[0]->offset=T->ptr[1]->offset=T->offset;

boolExp(T->ptr[0]); //循环条件，要单独按短路代码处理

T->width=T->ptr[0]->width;

strcpy(T->ptr[1]->Snext,newLabel());

semantic\_Analysis(T->ptr[1]); //循环体

if (T->width<T->ptr[1]->width) T->width=T->ptr[1]->width;

T->code=merge(5,genLabel(T->ptr[1]->Snext),T->ptr[0]->code, \

genLabel(T->ptr[0]->Etrue),T->ptr[1]->code,genGoto(T->ptr[1]->Snext));

break;

case EXP\_STMT:

T->ptr[0]->offset=T->offset;

semantic\_Analysis(T->ptr[0]);

T->code=T->ptr[0]->code;

T->width=T->ptr[0]->width;

break;

case ID:

case INT:

case FLOAT:

case CHAR:

case ASSIGNOP:

case AND:

case OR:

case RELOP:

case COMP\_PLUS:

case COMP\_MINUS:

case PLUS:

case MINUS:

case STAR:

case DIV:

case INC:

case DEC:

case NOT:

case UMINUS:

case FUNC\_CALL:

Exp(T); //处理基本表达式

break;

}

}

}

void semantic\_Analysis0(struct node \*T) {

symbolTable.index=0;

fillSymbolTable("read","",0,INT,'F',4);//事先默认的输入函数

symbolTable.symbols[0].paramnum=0;//read的形参个数

fillSymbolTable("write","",0,INT,'F',4);//事先默认的输出函数

fillSymbolTable("x","",1,INT,'P',12);

symbolTable.symbols[2].paramnum=1;

symbol\_scope\_TX.TX[0]=0;//外部变量在符号表中的起始序号为0

symbol\_scope\_TX.top=1;

T->offset=0;//外部变量在数据区的偏移量

semantic\_Analysis(T);

// objectCode(T->code);

}