

**课 程 实 验 报 告**

**课程名称： 编 译 原 理 实 验**

**专业班级： CS1707**

**学 号： U201714786**

**姓 名： 王占成**

**指导教师： 吴海**

**报告日期： 2020年 6月 28 日**

**计算机科学与技术学院**

**目录**

[1概述 1](#_Toc44233568)

[2系统描述 2](#_Toc44233569)

[2.1自定义语言概述 2](#_Toc44233570)

[2.2单词文法与语言文法 2](#_Toc44233571)

[2.3 符号表结构定义 3](#_Toc44233572)

[2.4 错误类型码定义 4](#_Toc44233573)

[2.5中间代码结构定义 5](#_Toc44233574)

[2.6 目标代码指令集选择 5](#_Toc44233575)

[3系统设计与实现 7](#_Toc44233576)

[3.1词法分析器 7](#_Toc44233577)

[3.2语法分析器 9](#_Toc44233578)

[3.3符号表管理 9](#_Toc44233579)

[3.4编译程序报错功能 11](#_Toc44233580)

[3.5中间代码生成功能 12](#_Toc44233581)

[3.6汇编代码生成功能 12](#_Toc44233582)

[4系统测试与评价 14](#_Toc44233583)

[4.1测试用例 14](#_Toc44233584)

[4.1.1词法分析、语法分析测试样例 14](#_Toc44233585)

[4.1.2报错功能测试样例 15](#_Toc44233586)

[4.1.3针对自增自减功能测试样例 16](#_Toc44233587)

[4.1.4符号表、中间代码、汇编代码生成功能正确测试样例 17](#_Toc44233588)

[4.2 正确性测试 17](#_Toc44233589)

[4.2.1语法分析与词法分析 17](#_Toc44233590)

[4.2.1 生成符号表、中间代码、汇编代码测试 22](#_Toc44233591)

[4.3 报错功能测试 26](#_Toc44233592)

[4.4 系统的优点 30](#_Toc44233593)

[4.5 系统的缺点 31](#_Toc44233594)

[5实验小结 32](#_Toc44233595)

[**参考文献** 33](#_Toc44233596)

[**附件：源代码** 34](#_Toc44233597)

# 1概述

本次实验是构造一个高级语言的子集的编译器，目标代码是汇编语言。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喜好选择实现。建议大家选用decaf语言或C语言的简单集合SC语言。

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。

在本次实验中，我选择C语言作为源语言。

# 2系统描述

## 2.1自定义语言概述

C语言包含关键字、运算符、界符、常量、标志符。在实现的简易的C语言编辑器中，包含有数据类型：char类型、int类型和float类型。能够定义基本类型的数据，使用基本类型进行赋值操作等。

运算符包含">"、"<"、">="、"<="、"=="、"!=" 、"=" 、"++" 、"--"、"+" 、"-"、"\*"、 "/" 、"&&"、"||" 、"." 、"!"、"(" 、")" 、"[" 、"]"

界符包含";"、"," 、"{"、"}"

关键字包含return，if，else，while，struct，for

词法分析的过程中还会识别注释、空白字符、换行符以及错误的标识符、字符串：处理换行符与制表符[\n]、 [ \r\t] ，识别单行注释与多行注释："/\*"(.|\n)\*"\*/" 、"\/\/".\* ，对于其他类型的参数，会进行报错处理，显示错误的位置及原因。

## 2.2单词文法与语言文法

在本次实验中，联合使用FLEX与BISO，使用LR分析法，在归约过程中，建立抽象语法树，通过优先级，结合性消除二义性。将所有的语句划分如图 2‑1：

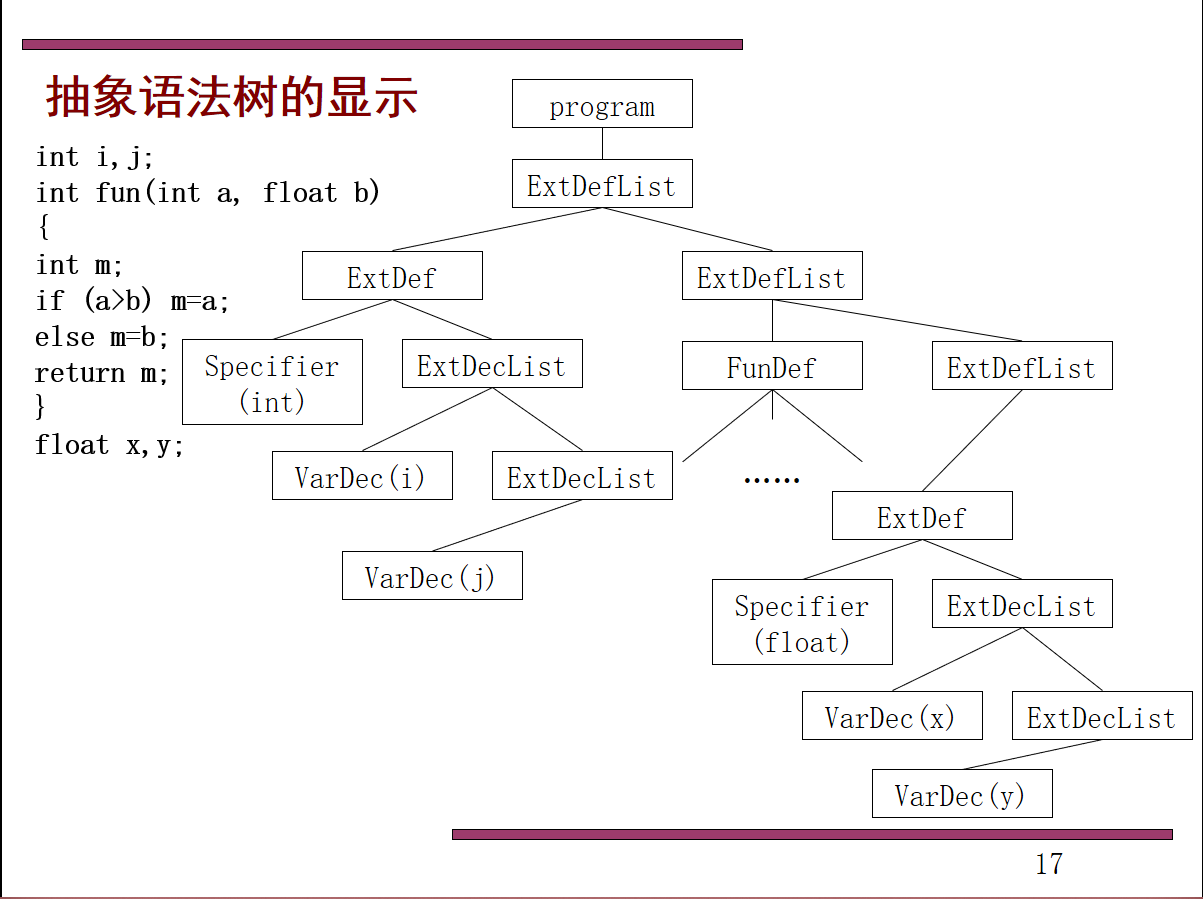


图 2‑1抽象语法树

在语句文法中包含的非终结符有

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

%type <ptr> ARRAY\_D ArrayIndex ForDec Specifier\_STRUCT CompSt\_STRUCT STRUCT\_TYPE

定义的终结符包含：

%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 //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 LC RC LB RB SEMI COMMA DOT STRUCT //用bison对该文件编译时，带参数-d，生成的exp.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 ASSIGNOP\_MINUS ASSIGNOP\_PLUS ASSIGNOP\_DIV ASSIGNOP\_STAR PLUS MINUS STAR DIV ASSIGNOP AND OR NOT IF ELSE WHILE RETURN FOR

标志结合性：

%left ASSIGNOP ASSIGNOP\_MINUS ASSIGNOP\_PLUS ASSIGNOP\_DIV ASSIGNOP\_STAR

%left OR

%left AND

%left RELOP

%left PLUS MINUS

%left STAR DIV

%left MOD

%right UMINUS NOT

%left PLUSONE MINUSONE

%left SEMI DOT

优先级：

%nonassoc LOWER\_THEN\_ELSE

%nonassoc ELSE

## 2.3 符号表结构定义

符号表包含变量名、别名、层号、类型、标记、偏移量。因此符号的符号表结构定义：

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

//其它...

};

//符号表，是一个顺序栈，index初值为0

struct symboltable{

struct symbol symbols[MAXLENGTH];

int index;

} symbolTable;

显示符号表部分：

void prn\_symbol()

{ //显示符号表

int i = 0;

printf("%10s %7s %10s %12s %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].array)

{

printf("%6s %6s %6d %10s %4c %6d\n", symbolTable.symbols[i].name,

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

symbolTable.symbols[i].type == INT ? "array\_int" : symbolTable.symbols[i].type == FLOAT ? "array\_float" : "array\_char",

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

}

else

{

printf("%6s %6s %6d %10s %4c %6d\n", symbolTable.symbols[i].name,

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

symbolTable.symbols[i].type == INT ? "int" : symbolTable.symbols[i].type == FLOAT ? "float" : "char",

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

}

}

}

## 2.4 错误类型码定义

在分析错误时，根据结点类型的进入不同的判断部分。

利用semantic\_error输出错误的位置与报错信息，包含行号、错误内容id，错误原因：

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);

}

## 2.5中间代码结构定义

使用void prnIR(struct codenode \*head)输出中间代码，包含参数与临时变量，包含两个参数与结果。根据指令的不同，生成不同的中间代码，中间代码结构如下：

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

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;

};

## 2.6 目标代码指令集选择

首先生成要在控制台输出的文字，定义入口地址为main函数

.data

\_Prompt: .asciiz "Enter an integer: "

\_ret: .asciiz "\n"

.globl main

之后要将read和write函数的汇编代码写出：

read:

li $v0,4

la $a0,\_Prompt

syscall

li $v0,5

syscall

jr $ra

write:

li $v0,1

syscall

li $v0,4

la $a0,\_ret

syscall

move $v0,$0

jr $ra

根据其他运算的类型，生成对应的汇编代码，如果遇到函数，在不是main函数的情况下，要分配函数对应给的空间，对于跳转指令、函数调用，要修改地址。

# 3系统设计与实现

## 3.1词法分析器

工作流程：



图 3‑1使用Flex和Bison构建语法分析器工作流程

在lex.l中定义词法规则，在parser.y中定义语法规则，对于不符合词法规则的单词，进行报错处理，对于不符合语法规则的语句，进行报错，表明错误位置。

在parser.y中，定义终结符，非终结符，优先级，结合性，匹配到对于的语法规则后，使用mknode存储语法树上的结点：

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;

va\_start(pArgs, pos);

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

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

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

va\_end(pArgs);

return T;

}

对于每一个非终结符，都要定义其规则，用“|”分隔不同的规则，将规则右部中需要存储到语法树种的内容，在ast.c中进行处理，以结构体处理为例：

首先在外部变量声明部分，添加对于结构体定义的语法：

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\_STRUCT SEMI {$$=mknode(1,STRUCT\_DEF,yylineno,$1);}

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

| error SEMI {$$=NULL; }

;

新增非终结符Specifier\_STRUCT标志结构体定义的语法：

Specifier\_STRUCT:

STRUCT ID CompSt\_STRUCT {$$=mknode(1,STRUCT\_DEC,yylineno,$3);strcpy($$->type\_id,$2);}//结构体名存放在$$->type\_id

| STRUCT ID STRUCT\_TYPE {$$=mknode(1,STRUCT\_D,yylineno,$3);strcpy($$->type\_id,$2);}

;

结构体有STRUCT标志符，结构体名称ID，结构体括号内内容组成。或者结构体声明的格式为结构体的名称。其中结构体定义的括号内内容格式：

CompSt\_STRUCT: LC DefList RC {$$=mknode(1,COMP\_STM\_STRUCT,yylineno,$2);};

而结构体的声明部分，由ID组成：

STRUCT\_TYPE:

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

;

对于结构体的使用，在表达式Exp中添加

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

其中函数对于所有结点信息的处理以及语法树的先根遍历输出均在ast.c中定义：

case STRUCT\_USE:

printf("%\*c结构体使用\n", indent, ' ');

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

printf("%\*c结构内部变量： \n", indent, ' ');

display(T->ptr[1],indent + 6);

break;

case STRUCT\_DEF:

printf("%\*c结构体定义\n", indent, ' ');

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

break;

case STRUCT\_DEC:

printf("%\*c结构体名称：%s \n",indent+3,' ',T->type\_id);

if(T->ptr[0]->ptr[0]){

printf("%\*c结构体内部成员：\n", indent, ' ');

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

}

break;

case COMP\_STM\_STRUCT:

printf("%\*c结构体的变量定义：\n",indent+3,' ');

display(T->ptr[0],indent+6); //显示定义部分

break;

case STRUCT\_D:

printf("%\*c结构体的声明：\n",indent,' ');

display(T->ptr[0],indent+3); //显示定义部分

printf("%\*c结构体类型：%s \n",indent+3,' ',T->type\_id);

break;

case STRUCT\_NAME:

printf("%\*c结构体名称：%s \n",indent+3,' ',T->type\_id);

break;

## 3.2语法分析器

语义分析的内容在函数semantic.c中，使用了void ext\_var\_list(struct ASTNode \*T)、int match\_param(int i, struct ASTNode \*T)、void boolExp(struct ASTNode \*T)、void Exp(struct ASTNode \*T)。

在语义分析的部分描述了符号表，符号表由变量名、别名、层号、类型、标记、偏移量构成。

与此同时，还要进行静态语义错误检查，检查了符号作用域与类型检查。对于作用域相关检查，需要语法树遍历时维护作用域栈。定义、声明与调用时等会重新产生一个新的作用域。其工作过程类似于逆波兰式的处理，遇到作用域括号“{”“}”进行相应的入栈出栈操作；在类型检查中，检查了赋值运算作用于两侧的表达式类型是否匹配，对于单目运算符两边的数据类型都是数据或者运算符右边是函数返回值的情况也进行了判断。对于不符合语义的内容，使用semantic\_error进行报错，对于正确的语句，使用符号表记录。

## 3.3符号表管理

语法分析与词法分析部分定义：

typedef union {

int type\_int;

float type\_float;

char type\_char;

char type\_id[32];

struct ASTNode \*ptr;

} YYLVAL;

标志操作定义：

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

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

};

代码结构：

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;

};

结点定义：

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

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

char type\_char;

};

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

int num;

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

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 array;

};

符号项定义：

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

int array;

//或函数活动记录大小，目标代码生成时使用

//其它...

};

符号表定义：

struct symboltable{

struct symbol symbols[MAXLENGTH];

int index;

} symbolTable;

## 3.4编译程序报错功能

实现报错功能15条：

|  |  |
| --- | --- |
| 序号 | 错误描述 |
| 1 | 除数为0 |
| 2 | 赋值语句需要左值 |
| 3 | 数组的错误赋值 |
| 4 | 使用未定义变量 |
| 5 | 使用未定义函数 |
| 6 | 对函数名称赋值 |
| 7 | 返回错误类型 |
| 8 | 函数参数个数错误 |
| 9 | 重复定义变量 |
| 10 | 不同类型变量之间赋值 |
| 11 | 函数名重复定义 |
| 12 | 调用非函数的变量 |
| 13 | 函数参数名重复 |
| 14 | 对于非int类型使用自增自减 |
| 15 | 函数参数类型错误 |

## 3.5中间代码生成功能

在遍历语法树时，在进行过语义分析后，生成中间代码，使用printIR打印中间代码。对于中间代码，使用的名称为符号表中过的别名。

在语义分析的过程中，使用genIR生成一条TAC代码的结点组成的双向循环链表：

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;

}

语义分析结束后，通过链表的内容生成中间代码，对于FUNCTION、PARAM、LABEL、GOTO即跳转打印出来，对于运算操作、赋值操作等，打印出运算过程。

## 3.6汇编代码生成功能

生成汇编代码在文件objectCode.c中实现，首先定义数据段与代码段，标志入口地址为main函数，写出read与write函数对应的汇编代码：

fp=fopen("object.s","w");

fprintf(fp,".data\n");

fprintf(fp,"\_Prompt: .asciiz \"Enter an integer: \"\n");

fprintf(fp,"\_ret: .asciiz \"\\n\"\n");

fprintf(fp,".globl main\n");

fprintf(fp,".text\n");

fprintf(fp,"read:\n");

fprintf(fp," li $v0,4\n");

fprintf(fp," la $a0,\_Prompt\n");

fprintf(fp," syscall\n");

fprintf(fp," li $v0,5\n");

fprintf(fp," syscall\n");

fprintf(fp," jr $ra\n");

fprintf(fp,"write:\n");

fprintf(fp," li $v0,1\n");

fprintf(fp," syscall\n");

fprintf(fp," li $v0,4\n");

fprintf(fp," la $a0,\_ret\n");

fprintf(fp," syscall\n");

fprintf(fp," move $v0,$0\n");

fprintf(fp," jr $ra\n");

使用MIPS指令集完成操作。对于跳转指令，根据条件跳转到目标地址，针对所有的赋值、运算操作，使用sw存储其运算结果，对于函数调用，分配活动空间，如加减乘除操作，使用lw取出操作数，使用sw存储计算结果，分别使用add，sub，mul，div指令进行对应的操作。

case PLUS:

case MINUS:

case STAR:

case DIV:

fprintf(fp, " lw $t1, %d($sp)\n", h->opn1.offset);

fprintf(fp, " lw $t2, %d($sp)\n", h->opn2.offset);

if (h->op==PLUS) fprintf(fp, " add $t3,$t1,$t2\n");

else if (h->op==MINUS) fprintf(fp, " sub $t3,$t1,$t2\n");

else if (h->op==STAR) fprintf(fp, " mul $t3,$t1,$t2\n");

else {fprintf(fp, " div $t1, $t2\n");

fprintf(fp, " mflo $t3\n");

}

fprintf(fp, " sw $t3, %d($sp)\n", h->result.offset);

break;

# 4系统测试与评价

## 4.1测试用例

由于结构体、多位数组等内容未在语义分析及之后的部分完成，所以使用了不同的测试样例，所以对于不同的实验采用了不同的测试样例。对于最后汇编代码的测试，由于Mars对于汇编代码入口地址处理方式的不同，需要在最终生成的object.s文件的首部.text段开头添加如下的代码：

main\_0:

jal main

li $v0,10

syscall

### 4.1.1词法分析、语法分析测试样例

该测试内容包含结构体与多维数组的定义、声明、使用

int a, b, c;

float m, n;

char fuck;

int test1[2];

int test2[2][3];

struct TS

{

int t;

float s;

};

struct TS testStr;

// 斐波拉契

int fibo(int a)

{

if (a == 1 || a == 2)

return 1;

return fibo(a - 1) + fibo(a - 2);

}

/\* 主函数\*/

int main()

{

int m, n, i;

char hello;

int ar[2][3];

int onear[2];

struct TS testTs1;

a++;

a--;

++b;

--b;

m = read();

hello = 'a';

ar[0][1] = 1;

c=ar[1][2];

i = 1;

testTs1.t = 1;

while (i <= m)

{

n = fibo(i);

write(n);

i++;

}

for (i = 1; i <= m; i = i + 1)

{

i = i + 1;

}

if(c>0){

c=1;

}else{

c=0;

}

i += 1;

return 1;

}

### 4.1.2报错功能测试样例

int a,b;

int fibo(int a, int a){

int d[3];

float x;

int y;

int y;

int a[1];

y ++;

ch ++;

x = y;

y = x;

a = 3 / 0;

3 = a;

1 = 2;

a /= 0;

n = a;

d = 3;

d--;

fibo = 1;

hello();

y();

if(1 || a==2){

return 1.1;

}

return d;

}

float fibo(){

return 1;

}

int main(){

int m,n,i;

float f;

i = 1;

fibo(a,b);

fibo(a,b,c);

fibo(a,f);

while(i <= m){

write(n);

i++;

}

return 1;

}

### 4.1.3针对自增自减功能测试样例

int main(){

int a = 1;

int b = 2;

b = ++a+a++;

a+++a++;

a+++--b;

a+++b--;

++a+b--;

return b;

}

### 4.1.4符号表、中间代码、汇编代码生成功能正确测试样例

int a,b,c;

float m,n;

int fibo(int a)

{

if (a == 1 || a == 2) return 1;

return fibo(a-1)+fibo(a-2);

}

int main()

{

int m,n,i;

m = read();

i=1;

while(i<=m)

{

n = fibo(i);

write(n);

i=i+1;

}

return 1;

}

## 4.2 正确性测试

### 4.2.1语法分析与词法分析

（1）对于外部变量定义

测试部分包含了int，char，float数据类型，包含一维数组、多位数组的定义：

int a, b, c;

float m, n;

char fuck;

int test1[2];

int test2[2][3];

测试结果如图所示

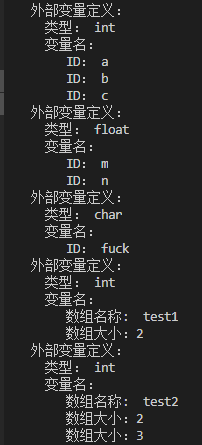


图 4‑1外部变量测试结果

（2）对于结构体的定义、声明、使用

结构体可以在外部定义、声明，可以在函数体内部声明、使用

对于函数体外部的定义：

struct TS

{

int t;

float s;

};

分析结果如图所示：

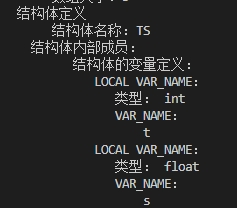


图 4‑2结构体定义测试

对于外部结构体声明：

struct TS testStr;

分析结果如图所示：



图 4‑3结构体声明

对于函数体结构体声明：

struct TS testTs1;

分析结果如图所示：



图 4‑4结构体声明-函数体内部

对于函数体结构体使用：

testTs1.t = 1;

分析结果如图所示：

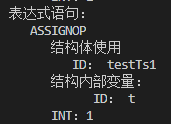


图 4‑5结构体使用

（3）对于自增自减语句

则测试结果中区分了运算符的位置，测试语句：

a++;

a--;

++b;

--b;

分析结果如图所示：

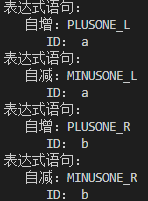


图 4‑6自增自减语句

（4）对于多维数组的使用

对于对维数组的使用与赋值：

ar[0][1] = 1;

c=ar[1][2];

分析结果如图所示：

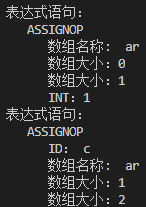


图 4‑7多维数组测试

（6）对于循环体语句

测试使用了for循环、while循环、if-else条件语句：

while (i <= m)

{

n = fibo(i);

write(n);

i++;

}

for (i = 1; i <= m; i = i + 1)

{

i = i + 1;

}

if(c>0){

c=1;

}else{

c=0;

}

for循环分析结果如图所示：

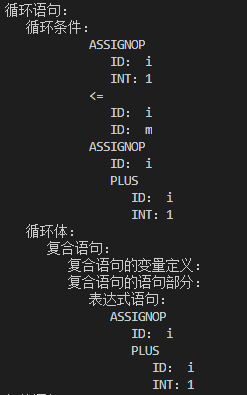


图 4‑8 for循环

while循环分析结果如图所示：

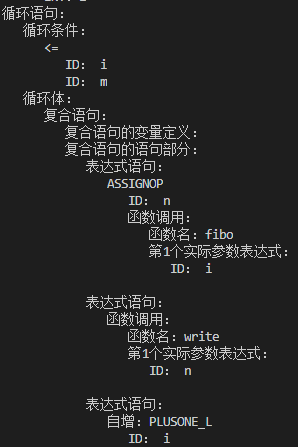


图 4‑9 while循环

if-else条件语句分析结果如图所示：

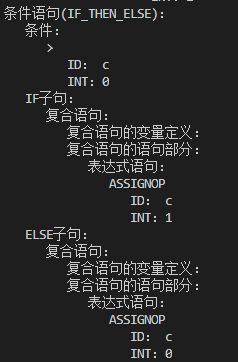


图 4‑10 if-else条件语句

### 4.2.1 生成符号表、中间代码、汇编代码测试

（1）针对自增自减功能：

测试内容：

int main(){

int a = 1;

int b = 2;

b = ++a+a++;

}

测试结果可见区分了++在前与在后的区别

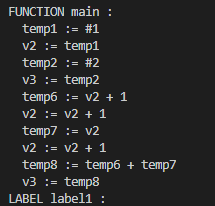


图 4‑11测试结果

（2）使用fibo测试:

测试代码：

int a,b,c;

float m,n;

int fibo(int a)

{

if (a == 1 || a == 2) return 1;

return fibo(a-1)+fibo(a-2);

}

int main()

{

int m,n,i;

m = read();

i=1;

while(i<=m)

{

n = fibo(i);

write(n);

i=i+1;

}

return 1;

}

生成符号表：



图 4‑12符号表1

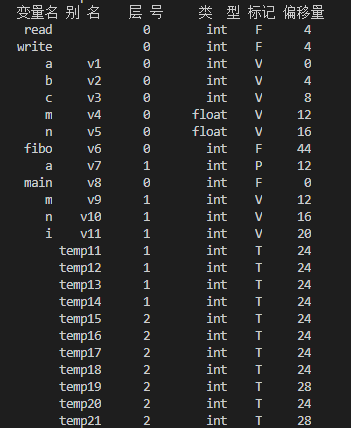


图 4‑13符号表2



图 4‑14符号表3

中间代码：

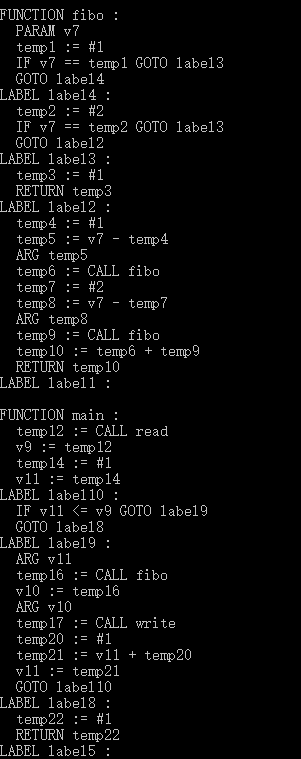


图 4‑15中间代码

汇编代码部分截图：

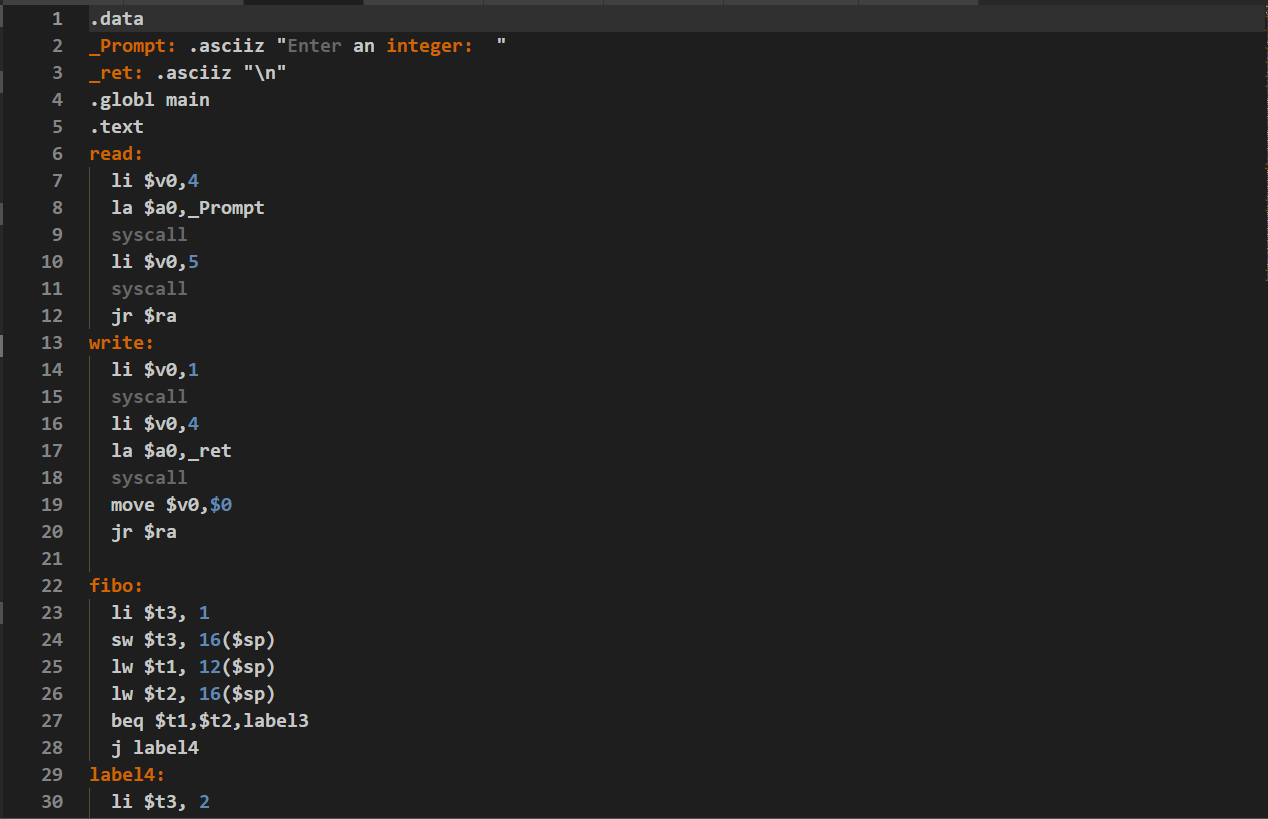


图 4‑16汇编代码

在Mars中运行汇编代码结果：

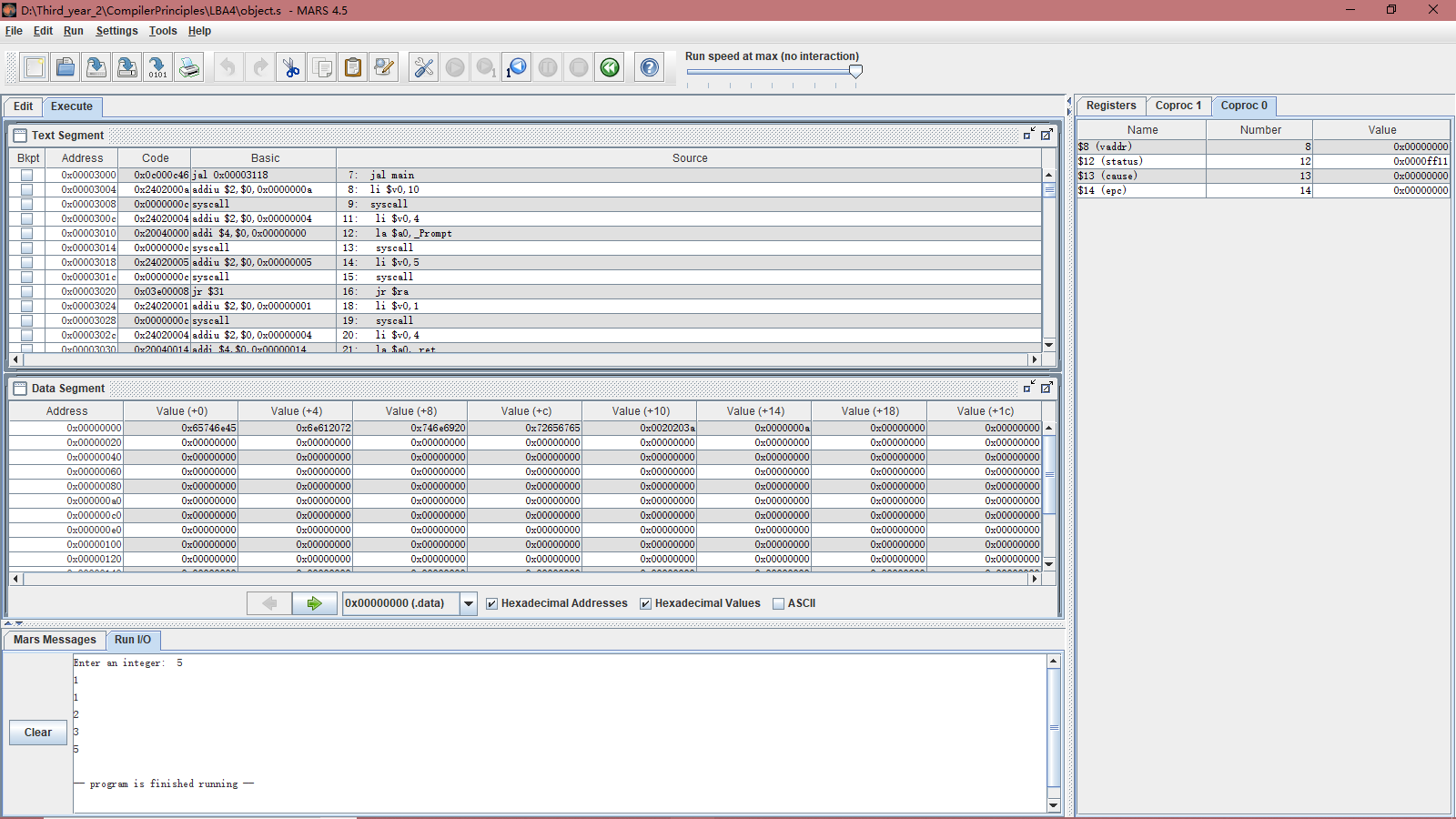


图 4‑17汇编代码运行结果

综上可见运行结果正确。

## 4.3 报错功能测试

（1）除数为0

测试内容：



图 4‑18测试语句

测试报错：



图 4‑19报错信息

（2）赋值语句需要左值

测试内容：



图 4‑20测试语句

测试报错：



图 4‑21报错信息

（3）数组的错误赋值

测试内容：





图 4‑22测试语句

测试报错：



图 4‑23报错信息

（4）使用未定义变量

测试内容：



图 4‑24测试语句

测试报错：



图 4‑25报错信息

（5）使用未定义函数

测试内容：



图 4‑26测试语句

测试报错：



图 4‑27报错信息

（6）对函数名称赋值

测试内容：



图 4‑28测试语句

测试报错：



图 4‑29报错信息

（7）返回错误类型

函数定义返回值为int：



图 4‑30函数定义

返回的数据类型维数组



图 4‑31返回类型

return语句：



图 4‑32返回语句

测试报错：



图 4‑33报错信息

（8）函数参数个数错误

测试内容：



图 4‑34测试语句

测试报错：



图 4‑35报错信息

（9）重复定义变量

测试内容：



图 4‑36测试语句

测试报错：



图 4‑37报错信息

（10）不同类型变量之间赋值

测试内容：



图 4‑38数据定义



图 4‑39变量使用语句

测试报错：



图 4‑40报错信息

（11）函数名重复定义

测试内容：



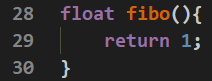


图 4‑41测试语句

测试报错：



图 4‑42报错信息

（12）定义于函数参数同名变量

测试内容：





图 4‑43测试语句

测试报错：



图 4‑44报错信息

（13）函数参数名重复

测试内容：



图 4‑45测试语句

测试报错：



图 4‑46报错信息

（14）错误调用非函数内容

测试内容：





图 4‑47测试语句

测试报错：



图 4‑48报错信息

（15）对于非int类型使用自增自减：

测试内容：





图 4‑49测试语句

测试报错：



图 4‑50报错信息

## 4.4 系统的优点

能够对多维数组、结构体的所有合法操作进行正确的词法分析与语法分析。

针对于自增自减运算，能够区分运算符的位置进行不同的操作，针对++a运算，运算时使用的值为a+1的值，同时a值加一，针对a++运算，运算时使用的值为a，a值加一，区分了两种不同的情况。

## 4.5 系统的缺点

在进行文法分析与词法分析时，实现的功能较为丰富，能过正确的使用for循环语句，能够实现多维数组的定义、赋值、使用，结构体的定义、生命、使用、赋值操作，但是在后续的实现部分，由于产生了报错，未查找到具体的出错原因，所以删除了部分功能，所以后续的实验中，缺失了对于多维数组与结构体的使用。

# 5实验小结

通过编译原理的实验，加深了对于编译原理相关知识的理解与体会，使用yacc与bison工具进行词法分析与语法分析，将我们熟知的C语言语法用类似于正则的方式表示出来，是一件很有意思的事情，在之后的语义分析中，分析语法树，查找出错误。之后生成中间代码与汇编代码，则是一个将正确的代码转化为可执行的过程。

在整个实验中，我加深了对课堂上学习的知识的理解，更好的认识了编译的过程，遗憾的是对于一部分的错误始终没有解决，最后在生成目标代码的内容中没有完全使用在语法分析中实现的内容，查找错误时也有一些问题没有解决，诸如对于函数调用参数过少的情况，能够识别出参数过多的情况但是对于参数过少的情况总是识别错误。

以上问题给我的实验完成带来了一些遗憾，但是在整个实验过程中，我学习到了很多知识，也很感谢吴海老师和我的同学们呢在实验过程中给予的执导，利于对于多维数组的内容的实现，感谢老师辛苦的付出，我感觉在此次实验中受益良多。

**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"

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;

float type\_float;

char type\_char;

char type\_id[32];

struct ASTNode \*ptr;

} 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 ('[A-Za-z0-9]')

%%

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

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

{char} {yylval.type\_char=atof(yytext); 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; }

"for" {return FOR;}

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

";" { return SEMI; }

"," { return COMMA; }

"--" { return MINUSONE; }

"++" { return PLUSONE; }

"\*=" { return DMUL; }

"+=" { return DPLUS; }

"-=" { return DMIN; }

"/=" { return DDIV;}

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

"=" { return ASSIGNOP; }

"+" { return PLUS; }

"-" { return MINUS; }

"\*" { return STAR; }

"/" { return DIV; }

"&&" { return AND; }

"||" { return OR; }

"!" { return NOT; }

"(" { return LP; }

")" { return RP; }

"[" { return LS; }

"]" { return RS; }

"{" { return LC; }

"}" { return RC; }

"%" { return MOD; }

[\n] { yycolumn=1; }

[ \r\t] {}

"//"[^\n]\* {/\*单行注释\*/}

"/\*"([^\\*]|(\\*)\*[^\\*/])\*(\\*)\*"\*/" {/\* 多行注释 \*/}

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

%%

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, ...);

void display(struct ASTNode \*,int);

%}

%union {

int type\_int;

float type\_float;

char type\_char;

char type\_id[32];

struct ASTNode \*ptr;

};

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

%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 //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 LS RS LC RC SEMI COMMA //用bison对该文件编译时，带参数-d，生成的exp.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 LCOM RCOM

%token PLUS MINUS STAR DIV ASSIGNOP AND OR NOT IF ELSE WHILE RETURN PLUSONE MINUSONE MOD FOR

%token DMUL DPLUS DMIN DDIV

%left ASSIGNOP

%left OR

%left AND

%left RELOP

%left PLUS MINUS

%left STAR DIV

%left MOD

%left DMUL DPLUS DMIN DDIV

%right UMINUS NOT PLUSONE MINUSONE

%nonassoc LOWER\_THEN\_ELSE

%nonassoc ELSE

%%

program: ExtDefList {

display($1,0);

semantic\_Analysis0($1);

} /\*显示语法树,语义分析 \*/

;

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

;

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

| error SEMI {$$=NULL;}

;

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

;

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);}

;

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 ArrayIndex {$$=mknode(1,ARRAY\_ID,yylineno,$2);strcpy($$->type\_id,$1);}

;

ArrayIndex: LS INT RS {$$=mknode(0,ARRAY\_INDEX,yylineno);$$->type\_int = $2;}

;

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

;

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

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

;

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

;

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

;

StmList: {$$=NULL; }

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

;

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(3,FOR,yylineno,$5,$7,$9);}

;

DefList: {$$=NULL; }

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

| error SEMI {$$=NULL;}

;

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");}

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

;

ArrayDec: LC Args RC {$$=mknode(1,ARRAY\_DEC,yylineno,$2);}

;

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 PLUSONE {$$=mknode(1,PLUSONE\_L,yylineno,$1);strcpy($$->type\_id,"PLUSONE");}

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

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

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

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

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

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

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

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

| 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");}

| 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 ArrayIndex {$$=mknode(1,ARRAY,yylineno,$2);strcpy($$->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 {$$=mknode(0,CHAR,yylineno);$$->type\_float=$1;$$->type=CHAR;}

;

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

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

;

%%

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");

}

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,

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,

ARRAY\_INDEX,ARRAY\_ID,ARRAY\_DEC,ARRAY,

PLUSONE\_L,PLUSONE\_R,MINUSONE\_L,MINUSONE\_R};

#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

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 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

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

char type\_char;

};

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

int num;

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

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 array;

};

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

int array;

//或函数活动记录大小，目标代码生成时使用

//其它...

};

//符号表，是一个顺序栈，index初值为0

struct symboltable{

struct symbol symbols[MAXLENGTH];

int index;

} symbolTable;

struct symbol\_scope\_begin { /\*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\*/

int TX[30];

int top;

} symbol\_scope\_TX;

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

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

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

void boolExp(struct ASTNode \*T);

void Exp(struct ASTNode \*T);

void prnIR(struct codenode \*head);

void objectCode(struct codenode \*head);

ast.c

#include "def.h"

#include "parser.tab.h"

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;

va\_start(pArgs, pos);

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

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

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

va\_end(pArgs);

return T;

}

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

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[1]->ptr[0]){

printf("%\*c数组类型：%s, 参数名：%s, 数组大小： %d \n", indent, ' ', \

T->ptr[0]->type==INT?"int": "float",T->ptr[1]->type\_id,T->ptr[1]->ptr[0]->type\_int);

}

else{

printf("%\*c类型：%s, 参数名：%s \n", indent,' ', \

T->ptr[0]->type==INT?"int": "float",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 FOR: //here

printf("%\*c循环语句：\n",indent,' ');

printf("%\*c循环条件：\n",indent+3,' ');

display(T->ptr[0],indent+6); //display loop condition

display(T->ptr[1],indent+6);

printf("%\*c循环体：\n",indent+3,' ');

display(T->ptr[2],indent+6);

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("%\*cLOCAL VAR\_NAME：\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("%\*cVAR\_NAME：\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);

}

else if (T0->ptr[0]->kind==ARRAY\_ID)

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

T0=T0->ptr[1];

}

break;

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

break;

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

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

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：%f\n",indent,' ',T->type\_char);

break;

case ASSIGNOP:

case PLUSONE\_L:

case MINUSONE\_L:

case PLUSONE\_R:

case MINUSONE\_R:

case MOD:

case AND:

case OR:

case RELOP:

case PLUS:

case MINUS:

case STAR:

case DMUL:

case DPLUS:

case DDIV:

case DMIN:

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：\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 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("\n");

break;

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

}

}

}

semantic.c

#include "def.h"

char \*strcat0(char \*s1, char \*s2)

{

static char result[10];

strcpy(result, s1);

strcat(result, s2);

return result;

}

char \*newAlias()

{

static int no = 1;

char s[10];

sprintf(s, "%d", no++);

return strcat0("v", s);

}

char \*newLabel()

{

static int no = 1;

char s[10];

sprintf(s, "%d", no++);

return strcat0("label", s);

}

char \*newTemp()

{

static int no = 1;

char s[10];

sprintf(s, "%d", no++);

return strcat0("temp", s);

}

//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;

}

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);

}

void prn\_symbol()

{ //显示符号表

int i = 0;

printf("%10s %7s %10s %12s %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].array)

{

printf("%6s %6s %6d %10s %4c %6d\n", symbolTable.symbols[i].name,

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

symbolTable.symbols[i].type == INT ? "array\_int" : symbolTable.symbols[i].type == FLOAT ? "array\_float" : "array\_char",

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

}

else

{

printf("%6s %6s %6d %10s %4c %6d\n", symbolTable.symbols[i].name,

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

symbolTable.symbols[i].type == INT ? "int" : symbolTable.symbols[i].type == FLOAT ? "float" : "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;

}

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

{

//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;

symbolTable.symbols[symbolTable.index].array = isArray;

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 ASTNode \*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, 0); //最后一个变量名

if (rtn == -1)

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

else

T->place = rtn;

T->num = 1;

break;

}

}

int match\_param(int i, struct ASTNode \*T)

{

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(T->pos, "", "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, "", "参数类型不匹配");

return 0;

}

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, "", "函数调用参数太多");

return 0;

}

return 1;

}

void boolExp(struct ASTNode \*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 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, "变量未定义");

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

semantic\_error(T->pos, T->type\_id, "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->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 ASTNode \*T)

{ //处理基本表达式，参考文献[2]p82的思想

int rtn, num, width;

struct ASTNode \*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, "变量未定义");

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

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

else if (symbolTable.symbols[rtn].array)

semantic\_error(T->pos, T->type\_id, "是数组名,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 ARRAY:

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

if (rtn == -1)

{

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

}

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

{

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

}

else if (!symbolTable.symbols[rtn].array)

{

semantic\_error(T->pos, T->type\_id, "不是数组类型");

}

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 ASSIGNOP:

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

{

semantic\_error(T->pos, "", "赋值语句需要左值");

}

else

{

Exp(T->ptr[0]);

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

Exp(T->ptr[1]);

if(T->ptr[1]->kind == ARRAY\_DEC){

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

semantic\_error(T->pos,T->type\_id,"类型不匹配");

}

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

{

semantic\_error(T->pos, T->type\_id, "类型不匹配");

}

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 DMUL:

case DPLUS:

case DMIN:

case DDIV:

if(T->kind == DDIV && T->ptr[1] == 0)

{

semantic\_error(T->pos,T->type\_id,"除数不能为0");

}

case AND: //按算术表达式方式计算布尔值，未写wan

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

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

{

semantic\_error(T->pos, "", "逻辑运算必须为int类型");

}

else

{

T->type = INT;

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

Exp(T->ptr[0]);

Exp(T->ptr[1]);

}

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, T->type\_id, "除数不能为0");

}

else

{

//semantic\_error(T->pos, T->ptr[0]->type, "shujuleixing");

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

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

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));

T->width = T->ptr[0]->width + T->ptr[1]->width + (T->type == INT ? 4 : (T->type==FLOAT ? 4 : 1));

}

break;

case PLUSONE\_L:

case MINUSONE\_L:

case PLUSONE\_R:

case MINUSONE\_R:

/\* if (T->ptr[0]->type != INT)

{

semantic\_error(T->pos, T->type\_id, "类型不是INT");

semantic\_error(T->pos,T->ptr[0], "类型不是INT");

}\*/

//else

//{

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

Exp(T->ptr[0]);

T->type = INT;

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

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;

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

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

/\*\*/

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

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;

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

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

\*/

//}

break;

case NOT: //未写完整

break;

case UMINUS: //未写完整

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

semantic\_error(T->pos, T->type\_id, "类型不正确");

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

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

else

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

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;

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

T->width = T->ptr[0]->width + (T->type == INT ? 4 : (T->type==FLOAT ? 4 : 1));

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, "函数未定义");

break;

}

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

{

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

break;

}

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

width = T->type == INT ? 4 : (T->type==FLOAT ? 4 : 1); //存放函数返回值的单数字节数

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]); //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 ASTNode \*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 ASTNode \*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

T->type = T->ptr[1]->type = !strcmp(T->ptr[0]->type\_id, "int") ? INT : !strcmp(T->ptr[0]->type\_id, "float") ? FLOAT : CHAR;

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

T->ptr[1]->width = T->type == INT ? 4 : (T->type==FLOAT ? 4 : 1); //将一个变量的宽度向下传递

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

T->width = (T->type == INT ? 4 : (T->type==FLOAT ? 4 : 1)) \* T->ptr[1]->num; //计算这个外部变量说明的宽度

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

T->ptr[1]->type = !strcmp(T->ptr[0]->type\_id, "int") ? INT : !strcmp(T->ptr[0]->type\_id, "float") ? FLOAT : CHAR; //获取函数返回类型送到含函数名、参数的结点

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, 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, "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, 0);

if (rtn == -1)

semantic\_error(T->ptr[1]->pos, T->ptr[1]->type\_id, "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

T->width = T->ptr[0]->type == INT ? 4 : (T->ptr[0]->type ==FLOAT ? 4 : 1); //参数宽度

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;

T->ptr[1]->type = !strcmp(T->ptr[0]->type\_id, "int") ? INT : !strcmp(T->ptr[0]->type\_id, "float") ? FLOAT : CHAR; //确定变量序列各变量类型

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;

width = T->ptr[1]->type == INT ? 4 : (T->ptr[1]->type==FLOAT ? 4 : 1); //一个变量宽度

while (T0)

{ //处理所以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, 0); //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, "变量重复定义");

else

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

T->width += width;

}

else if(T0->ptr[0]->kind == ARRAY\_ID)

{

T0->ptr[0]->array = 1;

int tmp\_offset = 4 \* T0->ptr[0]->ptr[0]->type\_int;

rtn = fillSymbolTable(T0->ptr[0]->type\_id, newAlias(), LEV, T0->ptr[0]->type, 'V', T->offset + tmp\_offset, 1);

if(rtn == -1)

{

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

}

}

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, 0); //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, "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 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 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, "返回值类型错误", "");

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 ID:

case INT:

case FLOAT:

case ASSIGNOP:

case AND:

case OR:

case RELOP:

case PLUS:

case MINUS:

case PLUSONE\_L:

case MINUSONE\_L:

case PLUSONE\_R:

case MINUSONE\_R:

case STAR:

case DIV:

case NOT:

case UMINUS:

case FUNC\_CALL:

Exp(T); //处理基本表达式

break;

}

}

}

void semantic\_Analysis0(struct ASTNode \*T)

{

symbolTable.index = 0;

fillSymbolTable("read", "", 0, INT, 'F', 4, 0);

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

fillSymbolTable("write", "", 0, INT, 'F', 4, 0);

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);

prnIR(T->code);

objectCode(T->code);

}

//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 == 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 == 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 PLUS:

case MINUS:

case STAR:

case DIV:

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

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

break;

case PLUSONE\_L:

case MINUSONE\_L:

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

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

h->op == PLUSONE\_L ? '+' : h->op == MINUSONE\_L ? '-' : '\\', '1');

break;

case PLUSONE\_R:

case MINUSONE\_R:

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

h->op == PLUSONE\_R ? '+' : h->op == MINUSONE\_R ? '-' : '\\', '1');

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

h->op == PLUSONE\_R ? '+' : h->op == MINUSONE\_R ? '-' : '\\', '1');

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:

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

break;

case CALL:

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);

}

objectCode.c

#include "def.h"

void objectCode(struct codenode \*head)

{

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

struct codenode \*h=head,\*p;

int i;

FILE \*fp;

fp=fopen("object.s","w");

fprintf(fp,".data\n");

fprintf(fp,"\_Prompt: .asciiz \"Enter an integer: \"\n");

fprintf(fp,"\_ret: .asciiz \"\\n\"\n");

fprintf(fp,".globl main\n");

fprintf(fp,".text\n");

fprintf(fp,"read:\n");

fprintf(fp," li $v0,4\n");

fprintf(fp," la $a0,\_Prompt\n");

fprintf(fp," syscall\n");

fprintf(fp," li $v0,5\n");

fprintf(fp," syscall\n");

fprintf(fp," jr $ra\n");

fprintf(fp,"write:\n");

fprintf(fp," li $v0,1\n");

fprintf(fp," syscall\n");

fprintf(fp," li $v0,4\n");

fprintf(fp," la $a0,\_ret\n");

fprintf(fp," syscall\n");

fprintf(fp," move $v0,$0\n");

fprintf(fp," jr $ra\n");

do {

switch (h->op) {

case ASSIGNOP:

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

fprintf(fp, " li $t3, %d\n", h->opn1.const\_int);

else {

fprintf(fp, " lw $t1, %d($sp)\n", h->opn1.offset);

fprintf(fp, " move $t3, $t1\n");

}

fprintf(fp, " sw $t3, %d($sp)\n", h->result.offset);

break;

case PLUS:

case MINUS:

case STAR:

case DIV:

fprintf(fp, " lw $t1, %d($sp)\n", h->opn1.offset);

fprintf(fp, " lw $t2, %d($sp)\n", h->opn2.offset);

if (h->op==PLUS) fprintf(fp, " add $t3,$t1,$t2\n");

else if (h->op==MINUS) fprintf(fp, " sub $t3,$t1,$t2\n");

else if (h->op==STAR) fprintf(fp, " mul $t3,$t1,$t2\n");

else {fprintf(fp, " div $t1, $t2\n");

fprintf(fp, " mflo $t3\n");

}

fprintf(fp, " sw $t3, %d($sp)\n", h->result.offset);

break;

case PLUSONE\_L:

case MINUSONE\_L:

fprintf(fp, " lw $t1, %d($sp)\n", h->opn1.offset);

//fprintf(fp, " lw $t2, %d($sp)\n", h->opn2.offset);

if (h->op==PLUSONE\_L) {fprintf(fp, " move $t3, $t1\n");

fprintf(fp, " addi $t1, $t1, 1\n");

}

else if (h->op==MINUSONE\_L) {fprintf(fp, " move $t3, $t1\n");

fprintf(fp, " addi $t1, $t1, -1\n");

}

fprintf(fp, " sw $t3, %d($sp)\n", h->result.offset);

break;

case PLUSONE\_R:

case MINUSONE\_R:

fprintf(fp, " lw $t1, %d($sp)\n", h->opn1.offset);

if (h->op==PLUSONE\_R) {

fprintf(fp, " add $t3,$t1,1\n");

fprintf(fp, " addi $t1, $t1, 1\n");

}

else if (h->op==MINUSONE\_R) {

fprintf(fp, " sub $t3,$t1,1\n");

fprintf(fp, " addi $t1, $t1, -1\n");

}

fprintf(fp, " sw $t3, %d($sp)\n", h->result.offset);

break;

case FUNCTION:

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

if (!strcmp(h->result.id,"main"))

fprintf(fp, " addi $sp, $sp, -%d\n",symbolTable.symbols[h->result.offset].offset);

break;

case PARAM:

break;

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

break;

case GOTO: fprintf(fp, " j %s\n", h->result.id);

break;

case JLE:

case JLT:

case JGE:

case JGT:

case EQ:

case NEQ:

fprintf(fp, " lw $t1, %d($sp)\n", h->opn1.offset);

fprintf(fp, " lw $t2, %d($sp)\n", h->opn2.offset);

if (h->op==JLE) fprintf(fp, " ble $t1,$t2,%s\n", h->result.id);

else if (h->op==JLT) fprintf(fp, " blt $t1,$t2,%s\n", h->result.id);

else if (h->op==JGE) fprintf(fp, " bge $t1,$t2,%s\n", h->result.id);

else if (h->op==JGT) fprintf(fp, " bgt $t1,$t2,%s\n", h->result.id);

else if (h->op==EQ) fprintf(fp, " beq $t1,$t2,%s\n", h->result.id);

else fprintf(fp, " bne $t1,$t2,%s\n", h->result.id);

break;

case ARG:

break;

case CALL: if (!strcmp(h->opn1.id,"read")){

fprintf(fp, " addi $sp, $sp, -4\n");

fprintf(fp, " sw $ra,0($sp)\n");

fprintf(fp, " jal read\n");

fprintf(fp, " lw $ra,0($sp)\n");

fprintf(fp, " addi $sp, $sp, 4\n");

fprintf(fp, " sw $v0, %d($sp)\n",h->result.offset);

break;

}

if (!strcmp(h->opn1.id,"write")){

fprintf(fp, " lw $a0, %d($sp)\n",h->prior->result.offset);

fprintf(fp, " addi $sp, $sp, -4\n");

fprintf(fp, " sw $ra,0($sp)\n");

fprintf(fp, " jal write\n");

fprintf(fp, " lw $ra,0($sp)\n");

fprintf(fp, " addi $sp, $sp, 4\n");

break;

}

for(p=h,i=0;i<symbolTable.symbols[h->opn1.offset].paramnum;i++)

p=p->prior;

fprintf(fp, " move $t0,$sp\n");

fprintf(fp, " addi $sp, $sp, -%d\n", symbolTable.symbols[h->opn1.offset].offset);

fprintf(fp, " sw $ra,0($sp)\n");

i=h->opn1.offset+1;

while (symbolTable.symbols[i].flag=='P')

{

fprintf(fp, " lw $t1, %d($t0)\n", p->result.offset);

fprintf(fp, " move $t3,$t1\n");

fprintf(fp, " sw $t3,%d($sp)\n", symbolTable.symbols[i].offset);

p=p->next; i++;

}

fprintf(fp, " jal %s\n",h->opn1.id);

fprintf(fp, " lw $ra,0($sp)\n");

fprintf(fp, " addi $sp,$sp,%d\n",symbolTable.symbols[h->opn1.offset].offset);

fprintf(fp, " sw $v0,%d($sp)\n", h->result.offset);

break;

case RETURN:fprintf(fp, " lw $v0,%d($sp)\n",h->result.offset);

fprintf(fp, " jr $ra\n");

break;

}

h=h->next;

} while (h!=head);

fclose(fp);

}