

**课 程 实验 报 告**

**题目： 编译原理实验**

**课程名称： 编译原理实验**

**专业班级： CS1703**

**学 号： U201714660**

**姓 名： 饶鸿斌**

**指导教师： 徐丽萍**

**报告日期： 2020.7.27**

**计算机科学与技术学院**

**目录**

[1概述 1](#_Toc46734809)

[2系统描述 2](#_Toc46734810)

[2.1自定义语言概述 2](#_Toc46734811)

[2.2单词文法与语言文法 2](#_Toc46734812)

[2.3 符号表结构定义 2](#_Toc46734813)

[2.4 错误类型码定义 2](#_Toc46734814)

[2.5中间代码结构定义 2](#_Toc46734815)

[2.6目标代码指令集选择 2](#_Toc46734816)

[3系统设计与实现 3](#_Toc46734817)

[**3.1** **词法分析器** 3](#_Toc46734818)

[**3.2** **语法分析器** 3](#_Toc46734819)

[**3.3** **符号表管理** 3](#_Toc46734820)

[**3.4** **语义检查** 3](#_Toc46734821)

[**3.5** **报错功能** 3](#_Toc46734822)

[**3.6** **中间代码生成** 3](#_Toc46734823)

[**3.7** **代码优化** 3](#_Toc46734824)

[**3.8** **汇编代码生成** 3](#_Toc46734825)

[4系统测试与评价 4](#_Toc46734826)

[**5.1** **测试用例** 4](#_Toc46734827)

[**5.2** **正确性测试** 4](#_Toc46734828)

[**5.3** **报错功能测试** 4](#_Toc46734829)

[**5.4** **系统的优点** 4](#_Toc46734830)

[**5.5** **系统的缺点** 4](#_Toc46734831)

[5实验小结或体会 5](#_Toc46734832)

[**参考文献** 6](#_Toc46734833)

[**附件：源代码** 7](#_Toc46734834)

# 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。

# 2系统描述

### 2.1自定义语言概述

本节给出的是一个简化的C语言的文法，文法如下：

G[program]:

program → ExtDefList

//program文件开始符号 ExtDefList指程序语句列表

ExtDefList→ExtDef ExtDefList | ε

//右递归：ExtDef表示的是某一行的语句 ExtDefList表示的是后面多行语句

ExtDef→Specifier ExtDecList ; |Specifier FunDec CompSt

//Specifier：标识符；ExtDecList：变量声明列表；

// Specifier：标识符；FunDec：定义函数；CompSt：函数体

Specifier→int | float|char

// Specifier：标识符； int | float|char：变量类型

ExtDecList→VarDec | VarDec , ExtDecList

//变量名列表，由一个或多个变量组成，多个变量之间用逗号隔开

VarDec→ID

//变量名称，由一个ID组成

FucDec→ID ( VarList ) | ID ( )

//函数定义:函数名+参数定义

VarList→ParamDec , VarList | ParamDec

//参数列表：有一个或者多个参数定义组成，用逗号隔开

ParamDec→Specifier VarDec

//参数的定义：固定有一个类型和一个变量组成

CompSt→{ DefList StmList }

//复合语句：左右分别用大括号括起来，中间有定义列表和语句列表

StmList→Stmt StmList | ε

//语句列表：多个stmt语句构成

Stmt→Exp ; | CompSt | return Exp ;

| if ( Exp ) Stmt | if ( Exp ) Stmt else Stmt | while ( Exp ) Stmt

DefList→Def DefList | ε

Def→Specifier DecList ;

DecList→Dec | Dec , DecList

Dec→VarDec | VarDec = Exp

//表示对语句分析时，有错误就会跳过分号，继续进行下去

/\*语句，可能为表达式，复合语句，return语句，if语句，if-else语句，while语句，break、continue,空语句\*/

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

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

| Exp + Exp | Exp - Exp | Exp \* Exp | Exp / Exp | ID | INT | FLOAT

| ( Exp ) | - Exp | ! Exp | ID ( Args ) | ID ( )

Args→Exp , Args | Exp

以上只是给出了一个很简单的语言文法，数据类型支持int、char、float；基本运算至少包括算术运算、比较运算、自增自减运算和复合赋值运算；控制语句包括if语句、while语句和break、continue语句以及for、switch语句；支持多维数组；语言支持行注释与块注释，不要求支持编译预处理命令和多文件程序

### 2.2单词文法与语言文法

按照语法定义列出所有的终结符以及非终结符，如表2.1所示：

表 2.1 单词文法描述

|  |  |  |
| --- | --- | --- |
| 单词符号类型 | 单词种类码 | 正则表达式 |
| {id} | ID | [A-Za-z][A-Za-z0-9]\* |
| {int} | INT | ([+-]?[0-9]+)|(0[xX][0-9a-fA-F]+)|(0[0-7]+) |
| {float} | FLOAT | ([0-9]\*\.[0-9]+)|([0-9]+\.) |
| {char} | CHAR | \'([^'\\]|\\['"?\\abfnrtv]|\\[0-7]{1,3}|\\[Xx][0-9A-Fa-f]+|{UCN})+\' |
| "int" | TYPE |  |
| "float" | TYPE |  |
| "char" | TYPE |  |
| "return" | RETURN |  |
| "if" | IF |  |
| "else" | ELSE |  |
| "while" | WHILE |  |
| "break" | BREAK |  |
| "continue" | CONTINUE |  |
| "switch" | SWITCH |  |
| "case" | CASE |  |
| "default" | DEFAULT |  |
| "for" | FOR |  |
| ";" | SEMI |  |
| "," | COMMA |  |
| ">"|"<"|">="|"<="|"=="|"!=" | RELOP |  |
| "+="|"\*="|"/="|"-=" | ROPR |  |
| "=" | ASSIGNOP |  |
| "+" | PLUS |  |
| "-" | MINUS |  |
| "\*" | MULTIPLY |  |
| "/" | DIVIDE |  |
| "++" | MPLUS |  |
| "--" | MMINUS |  |
| "&&" | AND |  |
| "||" | OR |  |
| "!" | NOT |  |
| "|" | ORBIT |  |
| "^" | XOR |  |
| "(" | LP |  |
| ")" | RP |  |
| "{" | LC |  |
| "}" | RC |  |
| "[" | LB |  |
| "]" | RB |  |

对于注释：

"//".\* { } /\*跳过单行注释\*/

"/\*" {BEGIN COMMENT;}/\*"/\*"这表示开头应为/\* BEFIN COMMENT为条件模式，COMMENT为条件名\*/

<COMMENT>"\*/" {BEGIN INITIAL;}/\*BEGIN INITIAL将休眠所有的条件模式，如果匹配到 退出整个条件模式\*/

<COMMENT>([^\*]|\n)+|. /\*跳过多行注释\*/

2.2.2 语法文法的描述

//辅助定义部分,union将各种类型统一起来

/\*Bison中默认将所有的语义值都定义为int类型，可以通过定义宏YYSTYPE来改变值的类型。

如果有多个值类型，则需要通过在Bison声明中使用%union列举出所有的类型。\*/

%union {

int type\_int;

float type\_float;

char type\_id[32];

char type\_char[3];

struct ASTNode \*ptr;

};

/\*

%type定义非终结符的语义值类型 %type <union 的成员名> 非终结符

%type <ptr> program ExtDefList

这表示非终结符 ExtDefList 属性值的类型对应联合中成员 ptr 的类型，在本实验中对应一个树结点的指针

\*/

// %type 定义非终结符的语义值类型

/\*%token<type\_id>ID表示的是识别出来的一个标识符后，标识符的字符串串值保存在成员type\_id中\*/

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

//% token 定义终结符的语义值类型

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

%token <type\_id> ID RELOP TYPE /\*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\_char> CHAR

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

//=assignop

%token PLUS MINUS STAR DIV MOD ASSIGNOP AND OR NOT IF ELSE WHILE RETURN FOR SWITCH CASE COLON DEFAULT COMADD COMSUB COMSTAR COMDIV COMMOD DOUBLEADD DOUBLESUB

%token BREAK CONTINUE

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

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

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

%token SWITCH\_STMT CASE\_STMT DEFAULT\_STMT EXP\_ARRAY ARRAY\_DEC DOUBLEADD\_R DOUBLEADD\_L DOUBLESUB\_L DOUBLESUB\_R

//这里是优先级定义

//left表示左结合，right表示右结合，前面符号的优先级低，后面的优先级高。

%left COMADD COMSUB COMSTAR COMDIV COMMOD //复合运算

%left ASSIGNOP

%left OR

%left AND

%left RELOP

%left PLUS MINUS

%left STAR DIV MOD

%left DOUBLEADD DOUBLESUB

//优先级高的单目-符号UMINUS

%right UMINUS NOT DPLUS

%right LB

%left RB

/\*%nonassoc的含义是没有结合性,它一般与%prec结合使用表示该操作有同样的优先级\*/

%nonassoc LOWER\_THEN\_ELSE

%nonassoc ELSE

接下来是对语法分析核心的定义，用于进行词法分析识别和构建对应的语法树结点，然后对其进行标记：

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

;

//作用：定义整个语法树

/\*ExtDefList：外部定义列表，即是整个语法树\*/

ExtDefList: {$$=NULL;}//语法树为空

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

;

//作用：外部变量或函数声明

//例如：int a;

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;printf("missing SML（分号）\n");}

;

//作用：Specifier表示的是一个类型:int、float、char

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

;

//变量名列表，由一个或多个变量组成，多个变量之间用逗号隔开

ExtDecList: VarDec {$$=mknode(1,EXT\_DEC\_LIST,yylineno,$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(int、char、float)的结点),第二棵子树对应剩下的外部变量名\*/

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

;

//变量名称，由一个ID组成

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

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

;

//数组定义

Arraylist:LB Exp RB {$$=mknode(1,ARRAY\_LAST,yylineno,$2);}

|LB Exp RB Arraylist {$$=mknode(2,ARRAY\_LIST,yylineno,$2,$4);}

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

;

//作用：函数定义:函数名+参数定义

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

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

|error RP {$$=NULL;printf("definr function wrong!\n");}

;

//参数列表：有一个或者多个参数定义组成，用逗号隔开

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

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

;

//参数的定义：固定有一个类型和一个变量组成

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

;

//复合语句：左右分别用大括号括起来，中间有定义列表和语句列表

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

;

//语句列表：多个stmt语句构成

StmList: {$$=NULL; }

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

;

//表示对语句分析时，有错误就会跳过分号，继续进行下去

/\*语句，可能为表达式，复合语句，return语句，if语句，if-else语句，while语句，break、continue,空语句\*/

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

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

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

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

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

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

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

| BREAK SEMI {$$=mknode(0,BREAK,yylineno);}

| CONTINUE SEMI {$$=mknode(0,CONTINUE,yylineno);}

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

| CASE CaseType COLON StmList {$$=mknode(3, CASE\_STMT, yylineno, $2, $4);}

| DEFAULT COLON Stmt {$$=mknode(1, DEFAULT\_STMT, yylineno,$3);}

;

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

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

;

ForDec: Exp SEMI Exp SEMI Exp {$$=mknode(3, FOR\_DEC, yylineno, $1, $3, $5);}

;

//作用:定义列表，由0个或者多个定义语句组成

DefList: {$$=NULL; }

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

| error SEMI {$$=NULL;}

;

//定义一个或者多个语句语句，分号隔开

//例如：int a=1,b=1;

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

;

//语句列表，由一个或者多个语句组成，逗号隔开，最终都成一个表达式

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

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

;

//语句：一个变量名称或者是一个赋值语句

Dec: VarDec {$$=$1;}

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

;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

| {$$=NULL;}

;

//用逗号隔开参数

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

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

;

### 2.3 符号表结构定义

在编译过程中，编译器使用符号表来记录语法树结点的特性信息。在语法分析时语法分析树和符号表是同时构建的。

符号表的作用：

* 是用来存放有关**标识符（符号）**的属性信息
  + 这些信息会在编译的不同阶段用到
  + 符号表的内容将用于**静态语义检查和产生中间代码**
  + 在目标代码生成阶段，**符号表是对符号名进行地址分配的依据**
  + 对一个多遍扫描的编译程序，不同遍所用的符号表也会有所不同，因为每遍所关心的信息或所能得到的信息会有差异
* 用来体现作用域与可见性信息

符号的常见属性

① 符号名

② 符号的类型：常量、变量、过程/函数、类的名称等

③ 符号的存储类别：常量、变量的数据类型，过程函数的返回类型等【决定了其存储格式和允许的操作】

④ 符号的作用域及可视性

⑤ 符号变量的存储类别和存储分配信息 存储类别确定其分配的区域，静态或动态数据区，堆区或栈区，存储分配信息如单元的大小，相对于某个存储区域的偏移位置等等

⑥ 符号的其它属性

* 数组内情向量
* 记录结构的成员信息
* 函数及过程的形参

本次实验采用的是最简单的顺序表结构：

第一步：定义符号表项

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

char name[33]; //变量或函数名

int level; //层号

int type; //变量类型或函数返回值类型

int paramnum; //对函数适用，记录形式参数个数

char alias[10]; //别名，为解决嵌套层次使用

char flag; //符号标记，函数：'F' 变量：'V' 参数：'P' 临时变量：'T'，增加数组：“A”

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，

//或记录函数活动记录大小，目标代码生成时使用

//函数入口等实验可能会用到的属性...

int const\_int;

int array[10];

};

第二步：用一个顺序表（符号栈）来表示当前的符号表，其中用一个index来标记栈顶。当加入一个新的符号时，栈顶向上移动一位，当作用域结束后退栈时栈顶向下移动一位。

//符号表,是一个顺序栈，index初始值为0

struct symboltable{

struct symbol symbols[MAXLENGTH];

int index;

} symbolTable;

//每到达一个复合语句时，将符号表的index值进栈，离开复合语句时，取出其退栈值修改符号表的index，完成删除该复合语句中的所有变量和临时变量

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,当使用顺序表作为符号表时，进入、退出一个作用域时需要对其操作，以完成符号表的管理。对其它形式的符号表，不一定需要此数据结构

int TX[30];

int top;

} symbol\_scope\_TX;

### 2.4 错误类型码定义

（1）使用未定义的变量；

（2）调用未定义或未声明的函数；

（3）在同一作用域，名称的重复定义（如变量名、函数名）。

（4）对非函数名采用函数调用形式；

（5）对函数名采用非函数调用形式访问；

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

（7）函数调用时实参和形参类型不匹配；

（8）对非数组变量采用下标变量的形式访问；

（9）数组变量的下标不是整型表达式；

（11）赋值号左边不是左值表达式；

（12）对非左值表达式进行自增、自减运算；

（13）对结构体变量进行自增、自减运算；

（14）类型不匹配。

（15）函数没有返回语句（当函数返回值类型不是void时）；

（16）break语句不在循环语句或switch语句中；

（17）continue语句不在循环语句中；

### 2.5中间代码结构定义

采用三地址代码作为中间代码，首先需要定义操作数的结构如下：

struct opn{

int kind; //标识联合成员的属性

int type; //标识操作数的数据类型

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

struct Array \*type\_array;

};

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; //偏移量，目标代码生成时用

};

然后对三地址代码结构进行定义：

struct codenode { //三地址TAC代码结点,采用单链表存放中间语言代码

int op;

struct opn opn1,opn2,result;

struct codenode \*next,\*prior;

};

并且按照如下表2-1定义中间代码的格式。

表2-1 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 | MULTIPLY | Y | Z | X |
| x := y / z | 除法操作 | DIVIDE | Y | Z | X |
| x\*:=y | 复合操作 | ROPE | X | Y | X |
| x++ | 自加操作 | MMUNIS | 1 |  | 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 |

### 2.6目标代码指令集选择

目标语言采用MIPS32指令指令序列，具体如下表2-2所示。

表2-2 MIPS32指令集

|  |  |
| --- | --- |
| 中间代码 | MIPS32指令 |
| LABEL x | x： |
| x :=#k | li reg(x),k |
| x := y | move reg(x), reg(y) |
| x := y + z | add reg(x), reg(y) , reg(z) |
| x := y - z | sub reg(x), reg(y) , reg(z) |
| x := y \* z | mul reg(x), reg(y) , reg(z) |
| x := y / z | div reg(y) , reg(z)  mflo reg(x) |
| GOTO x | j x |
| RETURN x | move $v0, reg(x)  jr $ra |
| IF x==y GOTO z | beq reg(x),reg(y),z |
| IF x!=y GOTO z | bne reg(x),reg(y),z |
| IF x>y GOTO z | bgt reg(x),reg(y),z |
| IF x>=y GOTO z | bge reg(x),reg(y),z |
| IF x<y GOTO z | ble reg(x),reg(y),z |
| IF x<=y GOTO z | blt reg(x),reg(y),z |
| X:=CALL f | jal f  move reg(x),$v0 |

# 3系统设计与实现

1. **词法分析器**

借助flex工具，编写lex文件来对指定的高级语言进行词法分析。

3.1.1定义部分

定义简单的可以理解成对后面需要使用的名词或正则表达式取一个别名来达到简化词法规则的目的。

/\*Flex 读入给定的输入文件，如果没有给定文件名的话，则从标准输入读取，\*/

/\*从而获得一个关于需要生成的扫描器的描述。\*/

/\*此描述叫做 规则，由正则表达式和 C代码对组成。\*/

/\*Flex 的输出是一个 C 代码文件——lex.yy.c——其中定义了yylex() 函数。\*/

/\*编译输出文件并且和 -lfl 库链接生成一个可执行文件。\*/

/\*当运行可执行文件的时候，它分析输入文件，为每一个正则表达式寻找匹配。\*/

/\*当发现一个匹配时，它执行与此正则表达式相关的C代码。\*/

#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 {

char type\_char[3];

int type\_int;

float type\_float;

char type\_id[32];

struct node \*ptr;

} YYLVAL;

#define YYSTYPE YYLVAL

3.1.2规则部分

规则部分对应的是正则表达式和与之对应的响应函数。

格式可以书写为：

Pattern {action}

其中pattern为正则表达式，其书写规则与前面部分的正则表达式定义相同。而action则为将要进行的具体操作。

其中由于flex工具的特点是由上往下尽可能的匹配最长的输入创进行词法分析，所以在这步需要将关键字写在前面然后将ID的识别写在后面。

同时对于注释的处理可以直接将其在action中用一段代码进行吞并，再词法分析阶段就直接注释。

按照以下的代码进行读取：

/\*通用字符名\*/

UCN (\\u[0-9a-fA-F]{4}|\\U[0-9a-fA-F]{8})

/\*识别16进制等\*/

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

errorid [0-9][0-9]\*[A-Za-z]\*

int ([+-]?[0-9]+)|(0[xX][0-9a-fA-F]+)|(0[0-7]+)

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

CHAR \'([^'\\]|\\['"?\\abfnrtv]|\\[0-7]{1,3}|\\[Xx][0-9A-Fa-f]+|{UCN})+\'

/\*转换规则：通过在声明部分的正则变量，识别输入流中的单词，并可以执行一些动作\*/

/\*词法分析器识别出一个单词后，将该单词对应的字符串保存在 yytext中，其长度为yyleng\*/

/\* 通过yylval.type\_int=strtol(yytext,NULL,0);将整常数的值保存在yylval的成员type\_in中，这里 yylval 是一个 Flex 和 Bison 共用的内部变量，类型为 YYLVAL，按这样的方式，在 Flex 中通过 yylval 的成员保存单词属性值，在 Bison 中就可以通过yylval 的成员取出属性值，实现了数据的传递\*/

/\*规则=模式+动作：分析器识别相应模式（在定义中定义的正则表达式）后对相应模式进行处理的C语言动作\*/

%%

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

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

{CHAR} {strcpy(yylval.type\_char,yytext);return CHAR;}

{errorid} {printf("非法定义int型数据 \"%s\"\n\t at Line %d\n",yytext,yylineno);}

"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;}

"break" {return BREAK;}

"continue" {return CONTINUE;}

"switch" {return SWITCH;}

"case" {return CASE;}

"default" {return DEFAULT;}

"for" {return FOR;}

{id} {strcpy(yylval.type\_id, yytext); return ID;}

";" {return SEMI;}

"," {return COMMA;}

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

"=" {return ASSIGNOP;}

"+" {return PLUS;}

":" {return COLON;}

"-" {return MINUS;}

"\*" {return STAR;}

"/" {return DIV;}

"%" {return MOD;}

"++" {return DOUBLEADD;}

"--" {return DOUBLESUB;}

"+=" {return COMADD;}

"-=" {return COMSUB;}

"\*=" {return COMSTAR;}

"/=" {return COMDIV;}

"%=" {return COMMOD;}

"&&" {return AND;}

"||" {return OR;}

"!" {return NOT;}

"(" {return LP;}

")" {return RP;}

"{" {return LC;}

"}" {return RC;}

"[" {return LB;}

"]" {return RB;}

3.1.3用户自定义代码

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

代码如下所示：

void main()

{

yylex();

return 0;

}

\*/

int yywrap()

{

return 1;

}

1. **语法分析器**

语法分析器与词法分析器同时进行，且用语法分析器驱动词法分析器。与之前一致Bison源代码分为三部分：定义部分、规则部分、用户定义部分。但是为了让词法分析与语法分析同时进行所以需要生成抽象语法树程序，对其进行显示，整个操作流程如下图所示。

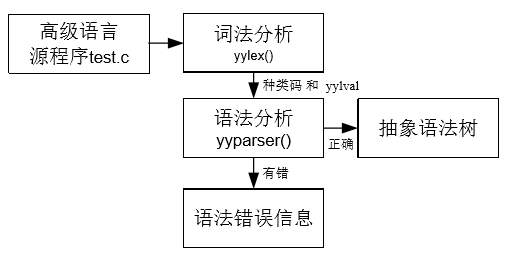


图3.2.1 语法分析架构

3.2.1定义部分

所有对词法单元的定义都可以放到这个部分。“%{“和”%}“包含的是代码，“%{“和”%}“包含的内容是对stdio.h的引用。以%token开头的此法单元（终结符）定义，与终结符相对的，所有未被定义为%token的符号都会被看作非终结符，这些非终结符要求必须在任意产生式的左边至少出现一次。

/\*LR分析法\*/

%error-verbose/\*指示bison生成详细的错误消息\*/

%locations/\*记录行号\*/

//此处是声明部分

%{

#include "stdio.h"

#include "math.h"

#include "string.h"

#include "def.h"

extern int yylineno;

extern char \*yytext;

extern FILE \*yyin;

int yylex();

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

void display(struct ASTNode \*,int);

%}

//辅助定义部分,union将各种类型统一起来

/\*Bison中默认将所有的语义值都定义为int类型，可以通过定义宏YYSTYPE来改变值的类型。

如果有多个值类型，则需要通过在Bison声明中使用%union列举出所有的类型。\*/

%union {

int type\_int;

float type\_float;

char type\_id[32];

char type\_char[3];

struct ASTNode \*ptr;

};

3.2.2规则部分

规则定义部分首先需要定义相关的语法然后定义相关的语法动作。

产生式中终结符和非终结符都各自对应一个属性值，同时根据头文件中的mknode函数进行参数的传递和结点的生成。这组语义动作会在整条产生式的最后可以添加一组产生式规约完成之后执行，如果不明确指定语义动作，那么bison将采用默认的语义动作{$$=$1}。对于有冲突的需要按照所指示的规定，限定各种操作符的优先度来避免冲突。

规则部分可参见上述的内容。

3.2.3用户函数部分

用户自定义函数如下：

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

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

if (!yyin) return -1;

yylineno=1;

yyparse();

return 0;

}

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

{

va\_list ap;

va\_start(ap, fmt);

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

vfprintf(stderr, fmt, ap);

fprintf(stderr, ".\n");

}

3.2.4抽象语法树节点的建立

抽象语法树的作用是将词法分析与语法分析得到的结果保存出来，然后方便之后语法的生成。

抽象语法树结点的数据结构为：

struct ASTNode {

//以下对结点属性定义没有考虑存储效率，只是简单地列出要用到的一些属性

int kind;

union {

char type\_char[3];

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

};

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

int place; //存放（临时）变量在符号表的位置序号

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

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

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

int type; //用以标识表达式结点的类型

int pos; //语法单位所在位置行号

int offset; //偏移量

int width; //占数据字节数

int num; //计数器，可以用来统计形参个数

};

抽象语法树的结点生成数据结构为：

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;

}

3.2.5 显示抽象语法树

抽象语法树的显示是根据先序遍历的方式，采用递归调用的方式输出整棵树。

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（外部变量和函数）列表中的第一个

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

break;

case EXT\_VAR\_DEF:

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

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

printf("%\*c变量名：\n",indent+3,' ');

display(T->ptr[1],indent+6); //显示变量列表

break;

case TYPE:

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

break;

case EXT\_DEC\_LIST:

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

{

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

printf("%\*c数组大小是：\n", indent, ' ');

}

display(T->ptr[0],indent); //依次显示外部变量名，

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

break;

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

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

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

display(T->ptr[2],indent+3); //显示函数体

break;

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

if (T->ptr[0]) {

printf("%\*c函数形参：\n",indent,' ');

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

}

else printf("%\*c此函数无参数\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: 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：(%d)\n",indent,' ',T->pos);

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

break;

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

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

break;

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

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

display(T->ptr[0],indent+6); //显示定义部分

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

display(T->ptr[1],indent+6); //显示语句部分

break;

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

display(T->ptr[1],indent); //显示剩下语句

break;

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

printf("%\*c循环条件：\n",indent+3,' ');

display(T->ptr[0],indent+6); //显示循环条件

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

display(T->ptr[1],indent+6); //显示循环体

break;

case BREAK:

printf("%\*cbreak语句:(%d)\n", indent, ' ');

break;

case BLANK:

//printf("%\*c空语句\n", indent, ' ');

break;

case CONTINUE:

printf("%\*ccontinue语句:(%d)\n", indent, ' ');

break;

case FOR:

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

printf("%\*c循环条件(FOR)：\n", indent+3, ' ');

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

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

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

break;

case SWITCH\_STMT:

printf("%\*cSWITCH语句：(%d)\n", indent, ' ', T->pos);

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

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

break;

case CASE\_STMT:

printf("%\*cCASE语句：(%d)\n", indent, ' ', T->pos);

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

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

break;

case FOR\_DEC:

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

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

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

break;

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

printf("%\*c条件：\n",indent+3,' ');

display(T->ptr[0],indent+6); //显示条件

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

display(T->ptr[1],indent+6); //显示if子句

break;

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

printf("%\*c条件：\n",indent+3,' ');

display(T->ptr[0],indent+6); //显示条件

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

display(T->ptr[1],indent+6); //显示if子句

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

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

break;

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

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

break;

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

display(T->ptr[0],indent+3); //显示变量类型

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

break;

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

T0=T;

while (T0) {

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

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

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

{

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

display(T0->ptr[0]->ptr[1],indent+strlen(T0->ptr[0]->ptr[0]->type\_id)+7);

}

T0=T0->ptr[1];

}

break;

case ARRAY\_LIST:

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

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

break;

case ARRAY\_ID:

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

printf("%\*c访问下标：\n", indent, ' ');

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

break;

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

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: %s\n", indent, ' ', T->type\_char);

break;

case ASSIGNOP:

case AND:

case OR:

case RELOP:

case PLUS:

case MINUS:

case STAR:

case MOD:

case DIV:

/\*add\*/

case COMADD:

case COMSUB:

case COMSTAR:

case COMDIV:

case COMMOD:

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

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

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

break;

case DOUBLEADD\_L:

case DOUBLESUB\_L:

case DOUBLEADD\_R:

case DOUBLESUB\_R:

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

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

break;

case NOT:

case UMINUS:

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

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

break;

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

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

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

break;

case ARGS: i=1;

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

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

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

display(T0,indent+3);

T=T->ptr[1];

}

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

// display(T,indent+3);

printf("\n");

break;

}

}

}

1. **符号表管理**

在进行语义分析时，对抽象语法树进行先序遍历扫描的同时进行符号表的管理，大致思想是根据抽象语法树的结点类型，如果为声明语句则创建符号然后入栈；当程序退出该模块时，则整体将符号表中的局部变量出栈；对于常量将其定义为一个没有名称的局部变量。同时需要定义LEV的全局变量，来标记并管理不同作用域内的符号。

对符号表的操作如下：

1. 当递归遍历到复合语句结点即COM\_STM时，符号表的管理进入一个新的作用域，需要对于LEV++，并符号栈symbol\_scope\_TX进栈用来记录符号表中该作用域的起点，方便之后的查找。
2. 当一个COM\_STM遍历结束时，符号表需要删除该复合语句内定义的全部变量，即将symbol\_scope\_TX退栈，取出该复合语句作用域的起点，根据值修改symbolTable.index，同时LEV减一。
3. 外部定义列表时，需要将定义语句中的每个标识符都插入到符号表中，并起上别名，方便之后中间代码的生成，避免冲突。
4. 当遇见函数定义时，需要将函数名同样压如符号表中，并对于有参数的函数需要将参数压入，并标记为‘P’。
5. 对于符号表的查找，需要保证就近优先的原则，从栈顶到栈底依次扫描，且不能将其他函数的形式参数用于本函数体。同时需要根据AST结点中的属性匹配相应的符号，保证函数对应函数，变量对应变量。
6. **语义检查**

该阶段的作用是进行静态语义检查，在之前构建的抽象语法树的基础上，进行语义检查和符号表的管理。

具体实现部分如下：

1. 控制流检查。控制流语句必须使得程序跳转到合法的地方。例如break、continue语句必须出现在循环语句当中。

具体的实现是通过采用在树节点中添加is-while继承属性

1. 唯一性检查。对于某些不能重复定义的对象或者元素，如同一作用域的标识符不能同名，需要在语义分析阶段检测出来。

具体的实现是通过符号表的查询来实现，保证同一作用域中不能定义之前定义过的变量。

1. 名字的上下文相关性检查。名字的出现在遵循作用域与可见性的前提下应该满足一定的上下文的相关性。

该过程就与唯一性一遍检查，从符号栈的栈顶到栈顶依次扫描，遇见第一个满足要求的就直接返回结果。

1. 类型检查包括检查函数参数传递过程中形参与实参类型是否匹配、是否进行自动类型转换等等。

通过抽象语法树结点的kind属性来进行判断并保证两边的数据类型一致，在实际参数与形式参数匹配的时候则是需要查看符号表，获取函数的信息并进行类型检测。

1. **报错功能**

报错功能包括了词法错误、语法错误、语义分析错误：

1. 词法错误：由flex的其他定义“{.}”来进行识别。
2. 语法错误：在抽象语法树生成阶段，使用C语言代码来进行判断。
3. 语义分析则通过语义检查，查看符号表或者树根结点信息来进行判断。

在整个判断错误的过程中，为了方便让编译器能够表示出准确的错误位置，需要对抽象语法树结点所标记的信息来确定，例如：确定行号、出错的类型以及终结符或者语句类型然后通过函数来显示。

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);

}

1. **中间代码生成**

3.6.1基本表达式翻译模式

（1）当Exp产生标识符ID,则修改传入的place属性赋值为ID所在的符号表的位置。

（2）当Exp产生整型INT，则将传入的place前面加上一个#的对应数值。

（3）当Exp产生算术表达式Exp op Exp，则首先传递继承属性offset并递归调用左部Exp，获取左部的综合属性width，同时将之前的offset+width作为右部的offset继续递归调用右部，最后在生成在符号表中加入一个新的匿名符号作为表达式计算的结果，并生成两数相加的中间代码例如place:=t1+t2，将这三段中间代码连接起来并返回。

（4）当Exp产生了赋值表达式Exp ASSIGNOP Exp，则需要对右部进行计算，然后对左部进行计算，并生成一条赋值语句的三地址代码，将三条代码合并后返回即可。

3.6.2语句翻译模式

（1）RETURN Exp；语句：先递归调用表达式函数处理Exp，然后生成一个return语句，调用函数将两者连接起来返回。

2）IF LP Exp RP Stmt语句：先需要根据生成两个新的标号，一个为条件表达式为真需要执行的地方，一个放在IF语句的结尾来表示条件表达式不为真需要跳转的地方的。同时将其传入到条件表达式中进行计算，并翻译得到该表达式代码1，然后递归调用程序块得到三地址代码2，将其按照（code1+label1+code2+label2）的顺序连接起来并返回。

3）IF LP Exp RP Stmt1 ELSE Stmt2语句：同上述（2），此外需要多生成一个标号，保证两个程序块开头都有标号，并按其顺序连接起来返回。

**3.6.3条件表达式翻译模式**

条件表达式的翻译，需要使用label\_true和label\_false两个继承属性。

对于多个逻辑表达式and和or的操作，需要进行优化处理，比如or中当前一个表达式成立时，直接调转到snext中，否则在跳转带第二个表达式进行计算，这样避免了重复不必要的逻辑表达式判断，

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->offset;

boolExp(T->ptr[0]);

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

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

boolExp(T->ptr[1]);

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;

3.6.4函数调用翻译模式

针对函数调用需要使用到翻译参数函数将实际的参数计算出来，然后构建一个临时变量列表同时对于参数需要将其按照从左到右的顺序翻译为par语句，具体的实现如下：

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

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

if (SymbolTable==-1){

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

return;

}

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

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

return;

}

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

//存放函数返回值的单数字节数

if(T->type == INT) {

width = 4;

}

else if(T->type == FLOAT) {

width = 8;

}

else 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;}

T1 = T->ptr[0];

while(T1 != NULL) {

count++;

num = SymbolTable;

if(symbolTable.symbols[SymbolTable+count].flag == 'P' && T1->type != symbolTable.symbols[SymbolTable+count].type){

semantic\_error(T1->pos, "", "实参与形参类型不符");

}

T1 = T1->ptr[1];

}

param\_num = symbolTable.symbols[SymbolTable].paramnum;

if (count > param\_num) {

semantic\_error(T->pos, "", "函数参数数量过多");

}

else if(count < param\_num) {

semantic\_error(T->pos, "", "函数参数数量过少");

}

else

match\_param(SymbolTable,T->ptr[0]); //处理所有参数的匹配

//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=SymbolTable; //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;

1. **代码优化**

本次实验没有针对代码进行优化。

1. **汇编代码生成**

本次实验的寄存器分配方式采取的是朴素寄存器分配法，实现起来比较简单，通过三个寄存器来完成一个语句的翻译，确保每次翻译一条中间代码之前都把需要用到的变量先加载到寄存器中，得到该代码的计算结果。然后把这些结果写回到内存中。这种方法简单且容易实现，但是最大的缺点就是寄存器的利用率过低。

具代汇编代码翻译关系如下：

表3-1中间代码到目标代码翻译

|  |  |
| --- | --- |
| 中间代码节点类型 | 目标代码形式 |
| x :=#k | li $t3,k sw $t3, x 的偏移量($sp) |
| x:=y | lw $t1, y的偏移量($sp)  move $t3,$t1  sw $t3, x的偏移量($sp) |
| x := y + z | lw $t1, y的偏移量($sp)  lw $t2, z的偏移量($sp)  add $t3,$t1,$t2  sw $t3, x的偏移量($sp) |
| x := y - z | lw $t1, y的偏移量($sp)  lw $t2, z的偏移量($sp)  sub $t3,$t1,$t2  sw $t3, x的偏移量($sp) |
| x := y \* z | lw $t1, y的偏移量($sp)  lw $t2, z的偏移量($sp)  mul $t3,$t1,$t2 |
| x := y / z | lw $t1, y的偏移量($sp)  lw $t2, z的偏移量($sp)  mul $t3,$t1,$t2  div $t1,$t2  mflo $t3  sw $t3, x 的偏移量($sp) |
| LABEL | Label\_name: #直接打印标号名 |
| Return x | move $v0, x 的偏移量($sp)  jr $ra |
| IF x==y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y的偏移量($sp)  beq $t1,$t2,z |
| IF x!=y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y 的偏移量($sp)  bne $t1,$t2,z |
| IF x>y GOTO z | lw $t1, x的偏移量($sp) lw $t2, y 的偏移量($sp) bgt $t1,$t2,z |
| x （JLE、JLT、JGE、JGT、EQ、NEQ ）y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y的偏移量($sp)  ble(blt bge bgt beq bne) $t1,$t2,z |
| FUN\_CALL | addi $fp,$sp,实参所占空间大小 #调整帧指针  addi $sp,$fp,-活动记录大小 #调整栈指针  jal x #跳转到函数x |

# 4系统测试与评价

1. **测试用例**

测试用例一：

|  |
| --- |
| int a, b, c;  float d, e, f;  char g;  int array[5];  float b; //变量名重复定义(3)  char array[10]; //数组变量名重复定义（3）  int arr[-5][0]; //数组定义错误  int funcA(int i, int j) {  float x = 1.0;  return x; //函数返回值类型与函数定义的返回值类型不匹配(16)  }  int funcA(int i, int j) { //函数名重复定义(3)  }  int funcB(){  ; //函数没有返回语句(17)  }  int main() {  k++; //使用未定义的变量(1)  z = y \* 1; //使用未定义的变量(1)  NoFunc(a); //调用未定义或未声明的函数(2)  array[1][-1]; //数组访问出错(3)  array[1]['A']; //数组访问出错(3)  a(i, j); //对非函数名采用函数调用形式（4）  funcA--; //对函数名采用非函数调用形式访问；(5)  funcA(a, b, c); // 实参表达式个数太多(6)  funcA(b); //参表达式个数太少(6)  funcA(d, e); //函数调用时实参和形参类型不匹配(7)  a[1] = 5; //对非数组变量采用下标变量的形式访问(8)  arr[1.1]; //数组变量的下标不是整型表达式(9)  arr['A']; //数组变量的下标不是整型表达式(9)  10=a; //赋值号左边不是左值表达式(12)  funcA(i, j)++;//对非左值表达式进行自增、自减运算(13)  b = 1 + 'A'; //类型不匹配(15)  d = 10 \* 12.3; //类型不匹配(15)  break; //break语句不在循环语句或switch语句中；(18)  continue; //continue语句不在循环语句中；(19)  //检查：  funcA(i,j) = i;  10++;  ++10;  'A'--;  --'A';  'A' = g; //赋值号左边不是左值表达式  g = 'A'; //合法  g--;  --g;  ++g;  g++;  'A' += 1; //复合运算类型匹配  1.01 -= 'A';  a \*= 'A';  1.11 /= 1;  a[10] %= 1.0;  a += 1.0;    return 'A'; //函数返回值类型与函数定义的返回值类型不匹配；  } |

该测试用例主要是为了测试对语义分析，同时针对不同的静态语义错误完成检查的工作。

测试用例二：

|  |
| --- |
| int a, b, c;  float m, n;  int d[10][10];  int fibo(int a) {  if (a == 1 || a == 2)  return 1;  return fibo(a - 1) + fibo(a - 2);  }  int main(int argc)  {  int i, temp;    for(i = 1;i<10;i++)  {  if(i>9)  {  d[i][i]=i;  temp = fibo(i);  write(temp);  break;  }  else  {  temp = fibo(i);  write(temp);  continue;  }  i++;  }  i=2;  switch(i)  {  case 0:  i++;  break;  case 1:  i++;  break;  default:  i++;  break;  }  return 1;  } |

该测试用例可以检验词法分析和语法分析是否正确；同时也可以对抽象语法树进行生成和对符号表进行生成和检查。同时也会生成其中间代码。上述内容的检查可以通过汇编编译来实现。

1. **正确性测试**

针对测试用例二的输出分析：

1. 测试词法分析和语法分析：

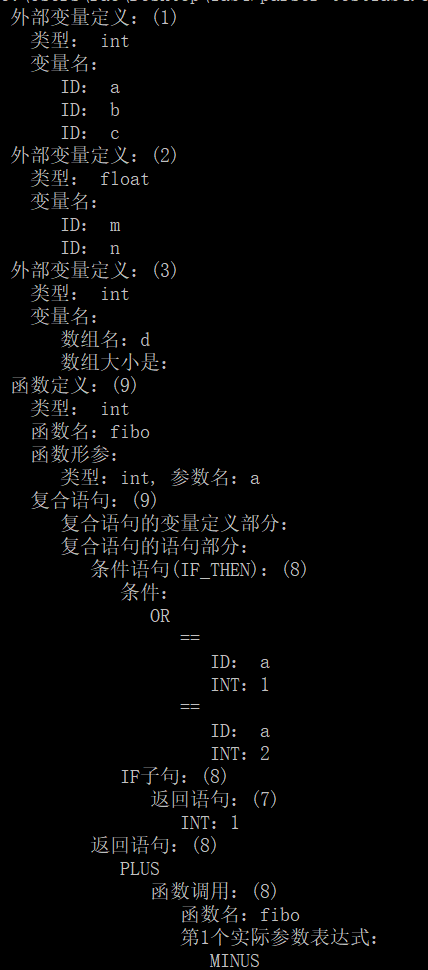


图4.1词法分析和语法分析1

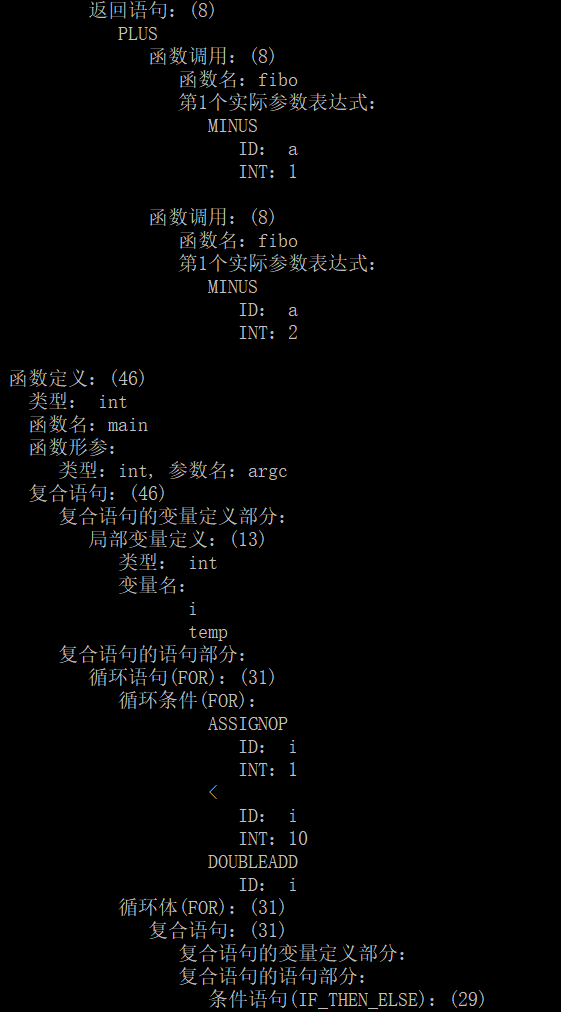


图4.2词法分析和语法分析2

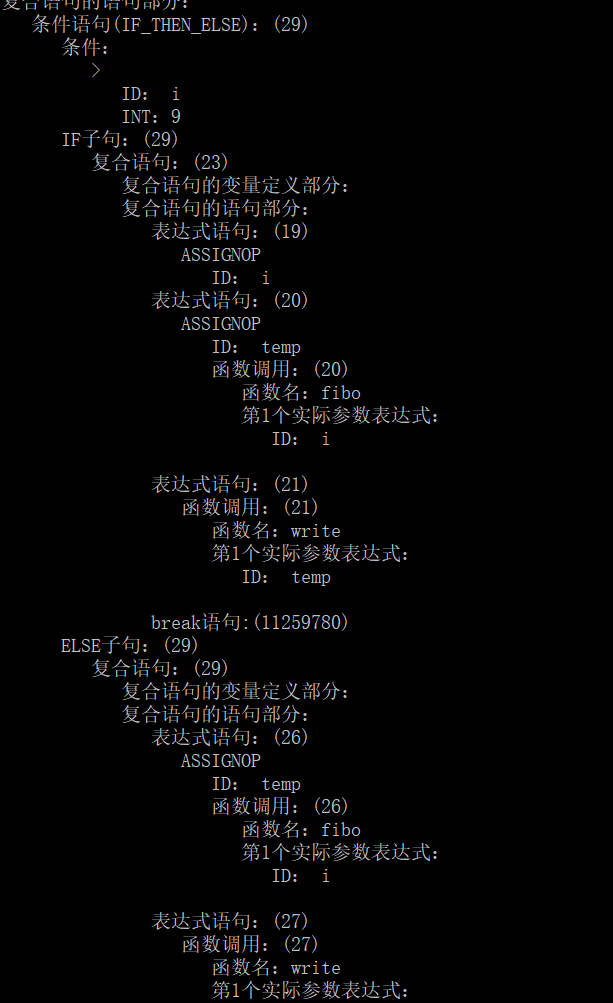


图4.3词法分析和语法分析3

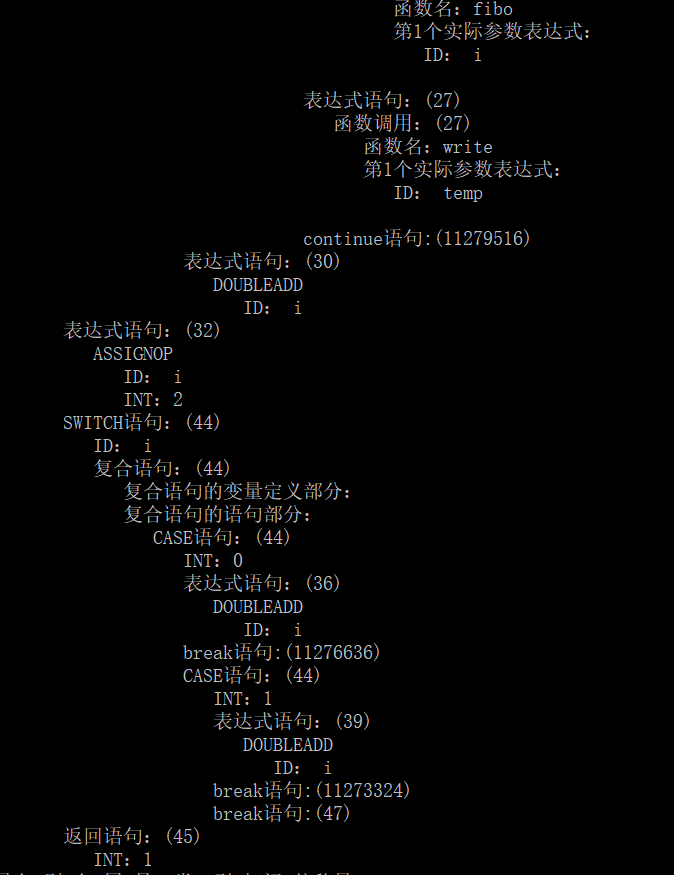


图4.4词法分析和语法分析4

1. 针对语义和符号表分析：

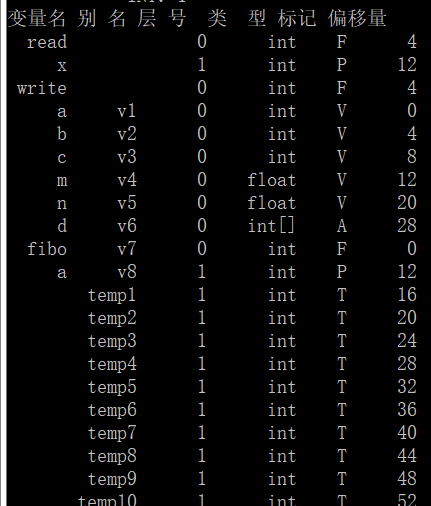


图4.5 fibo函数符号表

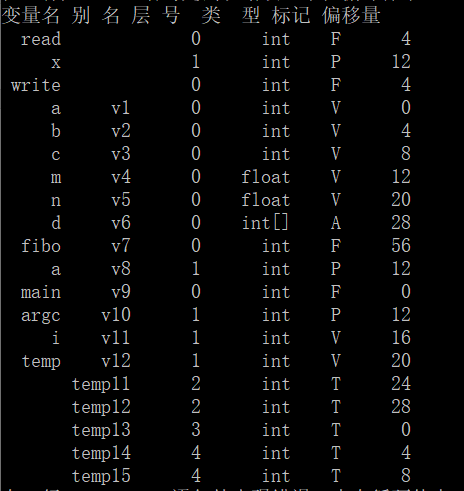


图4.6 main函数符号表

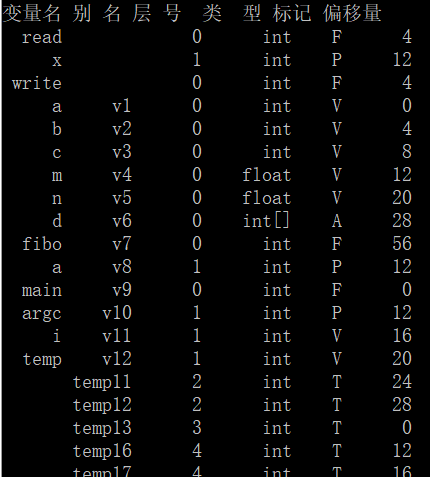


图4.7 main函数调用write函数符号表

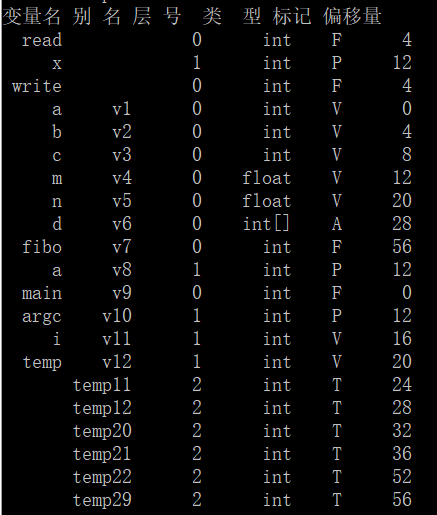


图4.8 main函数符号表

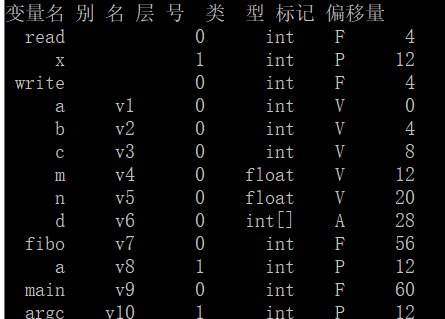


图4.9 最终输出符号表

1. 中间代码生成：

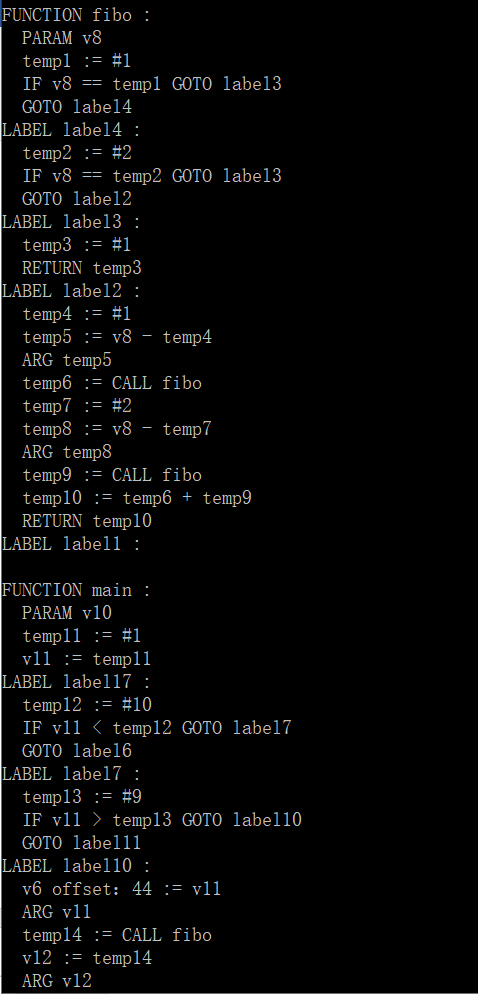


图4.10 中间代码生成

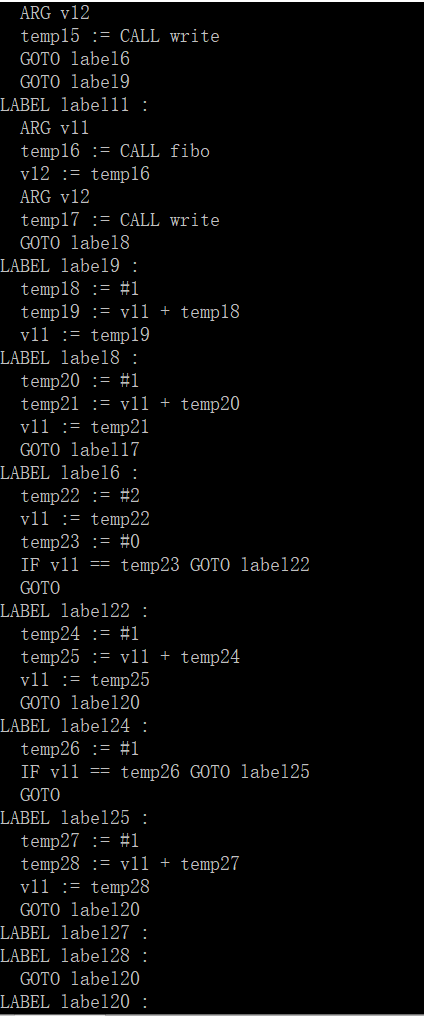


图4.11 中间代码生成

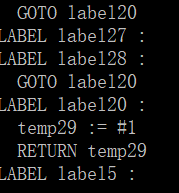


图4.12 中间代码生成

（4）测试目标代码：

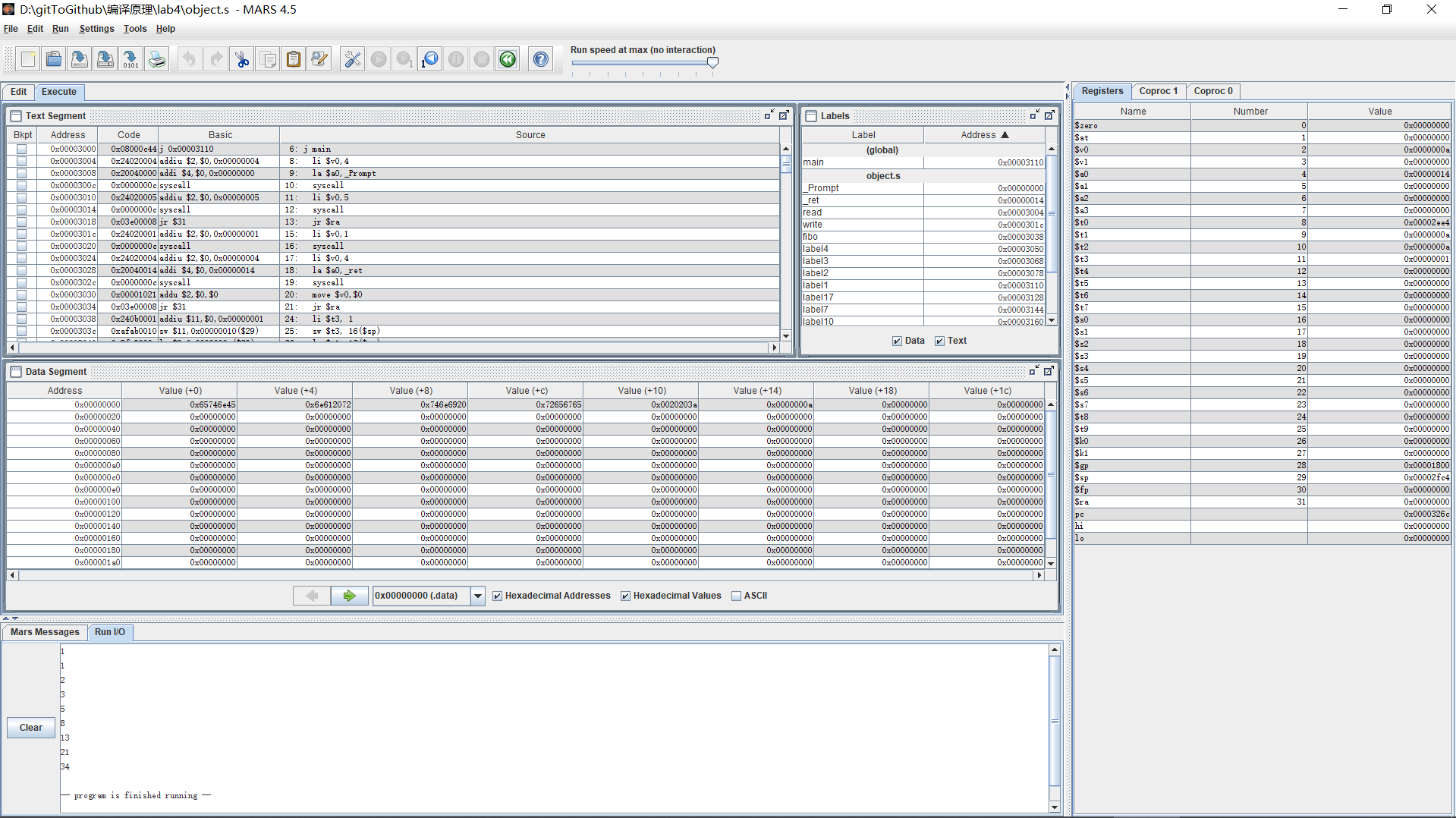
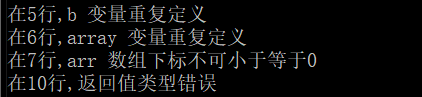


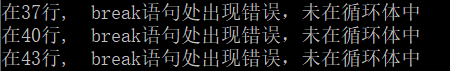
图4.13 测试目标代码

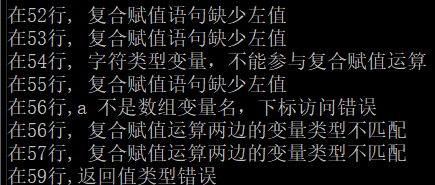
1. **报错功能测试**

针对测试用例一的静态语义分析的检查：









1. **系统的优点**

整个系统实现的基本功能比较全，对于错误的检查以及输出对应的错误提示比较细致。对于多维数组、数组下标的取值等问题都考虑到了。该系统比较中和的考虑了各个方面。

1. **系统的缺点**

没有进行代码的优化，整体的实现比较简单，没有涉及到比较好的算法。对于寄存器的使用分配等问题没有进一步进行讨论。同时相较于C语言，该系统实现的功能十分有限。

# 5实验小结或体会

四次编译实验总体的耗时很长，整个实验完成的时间跨度很长。加之疫情情况，大家都是线上完成，没有和老师、同学进行充分密切的沟通的可能性，这无疑加大了我们完成实验的难度。

编译实验，我个人认为是属于很难的实验了，这个实验不单单要求我们对编译原理这门课的知识点要深入理解和掌握，同时也要求我们要学习编译实验特定工具的使用和相互之间的联系。由于bison和flex工具在网上的资料十分有限，在刚开始接触的时候，我感觉十分棘手，很难上手。但是随着深入的不断完成实验，我渐渐的明白了这些工具的具体使用，对老师给的指导书中内容也慢慢明白了。从一步步语法分析、词法分析等到完成整个系统，我对课程上相关知识也越来越理解和掌握。

整个实验做下来，问题很多，遇到的困难也很多。在一些细节上，我一开始没有处理的思路，在查阅了相关资料，在同学的指点和阅读老师给的框架和他人的代码后才慢慢明白了一些方向。

最后，十分感谢认真负责的徐丽萍老师！感谢在我实验过程中对我解疑和提供很多帮助的同学们！

**参考文献**

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

[2] 胡伦俊等. 编译原理(第二版). 北京：电子工业出版社，2005

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

[4] 王雷等. 编译原理课程设计. 北京：机械工业出版社，2005

[5] 曹计昌等. C语言程序设计. 北京：科学出版社，2008

**附件：源代码**

**Lex.l**

%{

/\*Flex 读入给定的输入文件，如果没有给定文件名的话，则从标准输入读取，\*/

/\*从而获得一个关于需要生成的扫描器的描述。\*/

/\*此描述叫做 规则，由正则表达式和 C代码对组成。\*/

/\*Flex 的输出是一个 C 代码文件——lex.yy.c——其中定义了yylex() 函数。\*/

/\*编译输出文件并且和 -lfl 库链接生成一个可执行文件。\*/

/\*当运行可执行文件的时候，它分析输入文件，为每一个正则表达式寻找匹配。\*/

/\*当发现一个匹配时，它执行与此正则表达式相关的C代码。\*/

#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 {

char type\_char[3];

int type\_int;

float type\_float;

char type\_id[32];

struct node \*ptr;

} YYLVAL;

#define YYSTYPE YYLVAL

%}

%option yylineno

%x COMMENT

/\*通用字符名\*/

UCN (\\u[0-9a-fA-F]{4}|\\U[0-9a-fA-F]{8})

/\*识别16进制等\*/

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

errorid [0-9][0-9]\*[A-Za-z]\*

int ([+-]?[0-9]+)|(0[xX][0-9a-fA-F]+)|(0[0-7]+)

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

CHAR \'([^'\\]|\\['"?\\abfnrtv]|\\[0-7]{1,3}|\\[Xx][0-9A-Fa-f]+|{UCN})+\'

/\*转换规则：通过在声明部分的正则变量，识别输入流中的单词，并可以执行一些动作\*/

/\*词法分析器识别出一个单词后，将该单词对应的字符串保存在 yytext中，其长度为yyleng\*/

/\* 通过yylval.type\_int=strtol(yytext,NULL,0);将整常数的值保存在yylval的成员type\_in中，这里 yylval 是一个 Flex 和 Bison 共用的内部变量，类型为 YYLVAL，按这样的方式，在 Flex 中通过 yylval 的成员保存单词属性值，在 Bison 中就可以通过yylval 的成员取出属性值，实现了数据的传递\*/

/\*规则=模式+动作：分析器识别相应模式（在定义中定义的正则表达式）后对相应模式进行处理的C语言动作\*/

%%

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

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

{CHAR} {strcpy(yylval.type\_char,yytext);return CHAR;}

{errorid} {printf("非法定义int型数据 \"%s\"\n\t at Line %d\n",yytext,yylineno);}

"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;}

"break" {return BREAK;}

"continue" {return CONTINUE;}

"switch" {return SWITCH;}

"case" {return CASE;}

"default" {return DEFAULT;}

"for" {return FOR;}

{id} {strcpy(yylval.type\_id, yytext); return ID;}

";" {return SEMI;}

"," {return COMMA;}

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

"=" {return ASSIGNOP;}

"+" {return PLUS;}

":" {return COLON;}

"-" {return MINUS;}

"\*" {return STAR;}

"/" {return DIV;}

"%" {return MOD;}

"++" {return DOUBLEADD;}

"--" {return DOUBLESUB;}

"+=" {return COMADD;}

"-=" {return COMSUB;}

"\*=" {return COMSTAR;}

"/=" {return COMDIV;}

"%=" {return COMMOD;}

"&&" {return AND;}

"||" {return OR;}

"!" {return NOT;}

"(" {return LP;}

")" {return RP;}

"{" {return LC;}

"}" {return RC;}

"[" {return LB;}

"]" {return RB;}

[\n] {yycolumn=1;}

[ \r\t] {}

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

"//".\* { } /\*跳过单行注释\*/

"/\*" {BEGIN COMMENT;}/\*"/\*"这表示开头应为/\* BEFIN COMMENT为条件模式，COMMENT为条件名\*/

<COMMENT>"\*/" {BEGIN INITIAL;}/\*BEGIN INITIAL将休眠所有的条件模式，如果匹配到 退出整个条件模式\*/

<COMMENT>([^\*]|\n)+|. /\*跳过多行注释\*/

%%

/\* 和bison联用时，不需要这部分

void main()

{

yylex();

return 0;

}

\*/

int yywrap()

{

return 1;

}

**Parser.y**

/\*LR分析法\*/

%error-verbose/\*指示bison生成详细的错误消息\*/

%locations/\*记录行号\*/

//此处是声明部分

%{

#include "stdio.h"

#include "math.h"

#include "string.h"

#include "def.h"

extern int yylineno;

extern char \*yytext;

extern FILE \*yyin;

int yylex();

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

void display(struct ASTNode \*,int);

%}

//辅助定义部分,union将各种类型统一起来

/\*Bison中默认将所有的语义值都定义为int类型，可以通过定义宏YYSTYPE来改变值的类型。

如果有多个值类型，则需要通过在Bison声明中使用%union列举出所有的类型。\*/

%union {

int type\_int;

float type\_float;

char type\_id[32];

char type\_char[3];

struct ASTNode \*ptr;

};

/\*

%type定义非终结符的语义值类型 %type <union 的成员名> 非终结符

%type <ptr> program ExtDefList

这表示非终结符 ExtDefList 属性值的类型对应联合中成员 ptr 的类型，在本实验中对应一个树结点的指针

\*/

// %type 定义非终结符的语义值类型

/\*%token<type\_id>ID表示的是识别出来的一个标识符后，标识符的字符串串值保存在成员type\_id中\*/

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

//% token 定义终结符的语义值类型

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

%token <type\_id> ID RELOP TYPE /\*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\_char> CHAR

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

//=assignop

%token PLUS MINUS STAR DIV MOD ASSIGNOP AND OR NOT IF ELSE WHILE RETURN FOR SWITCH CASE COLON DEFAULT COMADD COMSUB COMSTAR COMDIV COMMOD DOUBLEADD DOUBLESUB

%token BREAK CONTINUE

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

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

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

%token SWITCH\_STMT CASE\_STMT DEFAULT\_STMT EXP\_ARRAY ARRAY\_DEC DOUBLEADD\_R DOUBLEADD\_L DOUBLESUB\_L DOUBLESUB\_R

//这里是优先级定义

//left表示左结合，right表示右结合，前面符号的优先级低，后面的优先级高。

%left COMADD COMSUB COMSTAR COMDIV COMMOD //复合运算

%left ASSIGNOP

%left OR

%left AND

%left RELOP

%left PLUS MINUS

%left STAR DIV MOD

%left DOUBLEADD DOUBLESUB

//优先级高的单目-符号UMINUS

%right UMINUS NOT DPLUS

%right LB

%left RB

/\*%nonassoc的含义是没有结合性,它一般与%prec结合使用表示该操作有同样的优先级\*/

%nonassoc LOWER\_THEN\_ELSE

%nonassoc ELSE

%%

//规则部分

/\*

$$是一个bison定义的特殊标记，其意义是当前语法栈的规约元素。如果没有规约动作代码，缺省情况下赋予$$为NULL

$1表示这个产生式的第1个语法成分所在的语法栈中对应的值

\*/

/\*

Exp: Exp ASSIGNOP Exp {$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno); } Exp →Exp =Exp

规则后面{}中的是当完成归约时要执行的语义动作。规则左部的 Exp 的属性值用$$表示，右部有 2 个 Exp，位置序号分别是 1 和 3，其属性值分别用$1 和$3表示

$$是子树的根结点指针，$1表示的是左值表达式的子树,$3表示的是右值表达式的子树，$2表示的是=号

yylineno表示赋值语句的行号

Stmt(statement) → error SEMI 表示对语句分析时，一旦有错，跳过分号（SEMI），继续进行语法分析

\*/

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

;

//作用：定义整个语法树

/\*ExtDefList：外部定义列表，即是整个语法树\*/

ExtDefList: {$$=NULL;}//语法树为空

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

;

//作用：外部变量或函数声明

//例如：int a;

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;printf("missing SML（分号）\n");}

;

//作用：Specifier表示的是一个类型:int、float、char

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

;

//变量名列表，由一个或多个变量组成，多个变量之间用逗号隔开

ExtDecList: VarDec {$$=mknode(1,EXT\_DEC\_LIST,yylineno,$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(int、char、float)的结点),第二棵子树对应剩下的外部变量名\*/

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

;

//变量名称，由一个ID组成

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

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

;

//数组定义

Arraylist:LB Exp RB {$$=mknode(1,ARRAY\_LAST,yylineno,$2);}

|LB Exp RB Arraylist {$$=mknode(2,ARRAY\_LIST,yylineno,$2,$4);}

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

;

//作用：函数定义:函数名+参数定义

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

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

|error RP {$$=NULL;printf("definr function wrong!\n");}

;

//参数列表：有一个或者多个参数定义组成，用逗号隔开

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

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

;

//参数的定义：固定有一个类型和一个变量组成

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

;

//复合语句：左右分别用大括号括起来，中间有定义列表和语句列表

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

;

//语句列表：多个stmt语句构成

StmList: {$$=NULL; }

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

;

//表示对语句分析时，有错误就会跳过分号，继续进行下去

/\*语句，可能为表达式，复合语句，return语句，if语句，if-else语句，while语句，break、continue,空语句\*/

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

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

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

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

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

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

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

| BREAK SEMI {$$=mknode(0,BREAK,yylineno);}

| CONTINUE SEMI {$$=mknode(0,CONTINUE,yylineno);}

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

| CASE CaseType COLON StmList {$$=mknode(3, CASE\_STMT, yylineno, $2, $4);}

| DEFAULT COLON Stmt {$$=mknode(1, DEFAULT\_STMT, yylineno,$3);}

;

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

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

;

ForDec: Exp SEMI Exp SEMI Exp {$$=mknode(3, FOR\_DEC, yylineno, $1, $3, $5);}

;

//作用:定义列表，由0个或者多个定义语句组成

DefList: {$$=NULL; }

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

| error SEMI {$$=NULL;}

;

//定义一个或者多个语句语句，分号隔开

//例如：int a=1,b=1;

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

;

//语句列表，由一个或者多个语句组成，逗号隔开，最终都成一个表达式

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

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

;

//语句：一个变量名称或者是一个赋值语句

Dec: VarDec {$$=$1;}

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

;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

| {$$=NULL;}

;

//用逗号隔开参数

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;

}

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");

}

**Display.c**

#include "def.h"

#define DEBUG 1

int LEV=0;//定义层号

int flag = 0;//表示break标志

char break\_label[30];//break打断跳出后要执行的中间代码

char continue\_label[30];//continue执行后要执行的中间代码

char array\_name[30];

int array\_index = 0;//数组的维数

int array\_flag=0;

char case\_temp[30];

char case\_label[30];

char array\_name[30];

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];

snprintf(s, 10, "%d", no++);

//itoa(no++,s,10);

return strcat0("v",s);

}

char \*newLabel() {

static int no=1;

char s[10];

snprintf(s, 10, "%d", no++);

//itoa(no++,s,10);

return strcat0("label",s);

}

char \*newTemp(){

static int no=1;

char s[10];

snprintf(s, 10, "%d", no++);

//itoa(no++,s,10);

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;

}

//输出中间代码

void prnIR(struct codenode \*head){

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

struct codenode \*h=head;

if(h){

do {

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

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

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

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

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 == CHAR)

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

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:

case MOD:

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

h->op == PLUS ? '+' : h->op == MINUS ? '-' : h->op == STAR ? '\*' : h->op == DIV ? '\\' : '%', 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: printf(" ARG %s\n",h->result.id);

break;

case CALL: /\* if (!strcmp(opnstr1,"write"))

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

else\*/

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("%6s %6s %6s %8s %4s %6s\n","变量名","别 名","层 号","类 型","标记","偏移量");

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

{

printf("%6s ",symbolTable.symbols[i].name);

printf("%6s ",symbolTable.symbols[i].alias);

printf("%6d ",symbolTable.symbols[i].level);

switch(symbolTable.symbols[i].type)

{

case INT:

if(symbolTable.symbols[i].flag=='A')

{

printf("%8s ","int[]");

}

else

{

printf("%8s ","int");

}

break;

case FLOAT:

if(symbolTable.symbols[i].flag=='A')

{

printf("%8s ","float[]");

}

else

{

printf("%8s ","float");

}

break;

case CHAR:

if(symbolTable.symbols[i].flag=='A')

{

printf("%8s ","char[]");

}

else

{

printf("%8s ","char");

}

break;

default:

break;

}

printf("%4c ",symbolTable.symbols[i].flag);

printf("%6d\n",symbolTable.symbols[i].offset);

}

}

//对符号表进行搜索

int searchSymbolTable(char \*name) {

/\* int i,flag=0;

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

if (symbolTable.symbols[i].level==0)

flag=1;

if (flag && symbolTable.symbols[i].level==1)

continue; //跳过前面函数的形式参数表项

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

}

return -1;\*/

int i;

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

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

return i;

return -1;

}

//对符号表进行填充,同时会进行符号查重,常量存储在符号表中

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

}

//获得宽度

int get\_width(struct ASTNode \*T){

if(T){

if(T->type==INT){

return T->type\_int;

}

return T->ptr[0]->type\_int\*get\_width(T->ptr[1]);

}

else

{

return 1;

}

}

int compute\_arraywidth(int \*array,int index){

int res=1;

while(array[index]!=0&&index<10){

res\*=array[index];

index++;

}

return res;

}

//获得多维数组的宽度

int getArray\_width(struct ASTNode \*T, int \*array, int index){

if(T) {

if(T->type == INT){

return T->type\_int;

}

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

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

return (symbolTable.symbols[SymbolTable].const\_int) \* compute\_arraywidth(array, index+1) + getArray\_width(T->ptr[1], array, index+1);

}

else

return (T->ptr[0]->type\_int) \* compute\_arraywidth(array, index+1) + getArray\_width(T->ptr[1], array, index+1);

}

else{

return 1;

}

}

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

int SymbolTable,num=1;

if (T) {

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

if (T->ptr[1]) {

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]);

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

if(T->ptr[1]) {

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

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

}

break;

case ID:

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

if (SymbolTable==-1)

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

else T->place=SymbolTable;

T->num=1;

break;

case ARRAY\_DEC:

array\_index = 0;

strcpy(array\_name, T->type\_id);

//对数组进行定义

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

if (SymbolTable==-1)

{semantic\_error(T->pos,T->type\_id, "变量重复定义");/\*break\*/}

else{

T->place = SymbolTable;

T->num = get\_width(T->ptr[0]);

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

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

array\_flag = 1;

}

break;

//多维数组

case ARRAY\_LIST:

SymbolTable = searchSymbolTable(array\_name);

if(SymbolTable == -1) {

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

}

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

semantic\_error(T->pos, "", "数组下标不是int型");

}

else{

symbolTable.symbols[SymbolTable].array[array\_index] = T->ptr[0]->type\_int;

array\_index++;

T->num = get\_width(T->ptr[1]);

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

}

break;

//数组的最后一维，或者就是一维数组

case ARRAY\_LAST:

SymbolTable = searchSymbolTable(array\_name);

if(SymbolTable == -1) {

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

}

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

semantic\_error(T->pos, "", "数组下标不是int型");

}

else{

symbolTable.symbols[SymbolTable].array[array\_index] = T->ptr[0]->type\_int;

}

break;

default:

break;

}

}

}

//对函数实参和型参进行检查

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

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

int type1, type2;

struct ASTNode \*T0 = T;

if (num == 0 && T == NULL)

return 1;

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

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

type2 = T0->ptr[0]->type;

if (type1 != type2) {

semantic\_error(T0->pos, "", "参数类型不匹配");

return 0;

}

T0 = T0->ptr[1];

}

return 1;

}

/\*当表达式作为控制语句的条件时，都会在其控制语句结点处给条件表达式子树的根结点属性.Etrue和.Efalse赋值，

所以在处理表达式结点时，如果属性.Etrue和.Efalse的值为空就按基本表达式处理，否则按布尔表达式处理。\*/

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 SymbolTable;

if (T)

{

switch (T->kind) {

//（int a=0表示就是false，反之就是true）

case INT:

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

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

else

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

T->width=4;

break;

case FLOAT:

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

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

else

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

T->width=8;

break;

case ID:

//这个地方需要查看符号表，找到符号表中的位置，然后保存type

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

if (SymbolTable == -1)

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

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

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

else if (symbolTable.symbols[SymbolTable].flag == 'A')

semantic\_error(T->pos, T->type\_id, "定义为数组名,类型不匹配");

else {

//opn1、opn2表示两个操作树，result表示运算结果

opn1.kind = ID;

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

opn1.offset = symbolTable.symbols[SymbolTable].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 ARRAY\_DEC:

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

if(SymbolTable==-1)semantic\_error(T->pos,T->type\_id,"数组尚未定义");

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

semantic\_error(T->pos,T->type\_id,"数组被定义成函数，类型不匹配");

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

semantic\_error(T->pos,T->type\_id,"数组被定义成其他变量，类型不匹配");

else

{

T->place=SymbolTable;

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

}

break;

//条件转移

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->offset;

Exp(T->ptr[0]);

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

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

boolExp(T->ptr[0]);

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

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

boolExp(T->ptr[1]);

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;

default:

break;

}

}

}

void Exp(struct ASTNode \*T)

{//处理基本表达式，参考文献[2]p82的思想

int SymbolTable,SymbolTable2,flag=0,num,width;

int count=0,param\_num;

struct ASTNode \*T0;

struct ASTNode \*T1;

struct opn opn1,opn2,result,result2;

if (T)

{

switch (T->kind) {

case ID: //查符号表，获得符号表中的位置，类型送type

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

if (SymbolTable==-1)

semantic\_error(T->pos,T->type\_id, "变量未定义，语义错误");

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

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

else {

T->place=SymbolTable; //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[SymbolTable].type;

T->offset=symbolTable.symbols[SymbolTable].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=8;

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[1];

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 EXP\_ARRAY:

//到符号表中找

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

if(SymbolTable==-1)semantic\_error(T->pos,T->type\_id,"数组尚未定义");

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

semantic\_error(T->pos,T->type\_id,"数组被定义成函数，类型不匹配");

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

semantic\_error(T->pos,T->type\_id,"数组被定义成其他变量，类型不匹配");

else

{

int index = 0;

T0 = T->ptr[0];

T->place = SymbolTable; //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[SymbolTable].type; // 标记ID类型

T->offset = (T->type == INT ? 4 : (T->type == FLOAT ? 8 : 1)) \* (getArray\_width(T->ptr[0], symbolTable.symbols[SymbolTable].array, 0)-1); // 内存中偏移值

T->width = 0; //未再使用新单元

while(T0->kind==ARRAY\_LIST){

Exp(T0->ptr[0]);

if(T0->ptr[0]->type != INT){

semantic\_error(T->pos,"", "数组维数需要是整型");

break;

}

else if(symbolTable.symbols[SymbolTable].array[index] <= T0->type\_int){

semantic\_error(T->pos,"", "数组维度过大");

break;

}

index++;

T0=T0->ptr[1];

T->width += 4;

}

//处理最后一维

if(T0->kind == ARRAY\_LAST){

Exp(T0->ptr[0]);

if(T0->ptr[0]->type!=INT){

semantic\_error(T->pos,"", "数组维数需要整型");

}

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

if(symbolTable.symbols[SymbolTable].array[index] <= symbolTable.symbols[T0->ptr[0]->place].const\_int){

semantic\_error(T->pos,"", "数组维度过大");

}

}

else{

if(symbolTable.symbols[SymbolTable].array[index] <= T0->ptr[0]->type\_int){

semantic\_error(T->pos,"", "数组维度过大");

}

}

T->width += 4;

}

}

break;

//=号

case ASSIGNOP:

if (T->ptr[0]->kind!=ID&& T->ptr[0]->kind != EXP\_ARRAY){

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

}

else {

Exp(T->ptr[0]); //处理左值，例中仅为变量

T->ptr[1]->offset=T->offset;//将地址偏移量往子树上传递

Exp(T->ptr[1]);

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

{

semantic\_error(T->pos, "", "=号两边的类型不匹配");

return;

}

if (T->ptr[0]->type == CHAR && T->kind != ASSIGNOP) {

semantic\_error(T->pos, T->ptr[0]->type\_id, "左边字符类型变量不能参与运算");

return;

}

if (T->ptr[1]->type == CHAR && T->kind != ASSIGNOP) {

semantic\_error(T->pos, T->ptr[1]->type\_id, "右边字符类型变量不能参与运算");

return;

}

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

symbolTable.symbols[T->ptr[0]->place].const\_int = T->ptr[1]->type\_int;

}

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

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

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

if(T->ptr[1]->kind != EXP\_ARRAY ) {

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;

}

else{

opn1.kind=ID;

char s[10];

char str[80];

int id;

strcpy (str,symbolTable.symbols[T->ptr[1]->place].alias);

strcat (str," offset：");

sprintf(s,"%d",T->ptr[1]->offset);

strcat (str,s);

strcpy(opn1.id,str);

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

}

if(T->ptr[0]->kind != EXP\_ARRAY) {

result.kind = ID;

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

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

}

else{

result.kind=ID;

char s[10];

char str[80];

int id;

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

strcat (str," offset：");

sprintf(s,"%d",T->ptr[0]->offset);

strcat (str,s);

strcpy(result.id,str);

result.offset = symbolTable.symbols[T->ptr[0]->place].offset + T->ptr[0]->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 PLUS:

case MINUS:

case MOD:

case STAR:

case DIV: //算数运算，+-\*/

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 == CHAR) {

semantic\_error(T->pos, T->ptr[0]->type\_id, "为字符类型变量，不能参与运算");

break;

}

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

semantic\_error(T->pos, T->ptr[1]->type\_id, "为字符类型变量，不能参与运算");

break;

}

//表达式两边是float型

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;

//表达式两边是int型

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

{

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

semantic\_error(T->pos, " ", "运算符两边的变量类型不匹配");

break;

}

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 NOT: //未写完整

T->type=INT;

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

Exp(T->ptr[0]);

break;

case UMINUS://未写完整

/\*T->type=T->ptr[0]->type;

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

Exp(T->ptr[0]);\*/

break;

case DOUBLEADD\_L:

case DOUBLEADD\_R:

case DOUBLESUB\_L:

case DOUBLESUB\_R:

double\_add\_or\_sub\_exp(T);

break;

case COMADD:

case COMSUB:

case COMDIV:

case COMSTAR:

case COMMOD:

if (T->ptr[0]->kind != ID && T->ptr[0]->kind != EXP\_ARRAY) {

semantic\_error(T->pos, "", "复合赋值语句缺少左值");

break;

}

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

semantic\_error(T->pos, T->ptr[0]->type\_id, "为字符类型变量，不能参与复合赋值运算");

break;

}

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

semantic\_error(T->pos, "", "字符类型变量，不能参与复合赋值运算");

break;

}

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

Exp(T->ptr[0]);

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

Exp(T->ptr[1]);

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

T->type = FLOAT;

}

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

T->type = INT;

}

else {

semantic\_error(T->pos, "", "复合赋值运算两边的变量类型不匹配");

}

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[0]->place].alias); //右值一定是个变量或临时变量

//opn1.type = INT;

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

opn2.kind = ID;

//opn2.type = INT;

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, newTemp());

if (T->kind == COMADD)

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

else if(T->kind == COMSUB)

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

else if(T->kind == COMSTAR)

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

else if(T->kind == COMDIV)

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

else

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

result2.kind = ID;

strcpy(result2.id, symbolTable.symbols[T->ptr[0]->place].alias);

result2.type = T->type;

result2.offset = symbolTable.symbols[T->ptr[0]->place].offset;

T->code = merge(2, T->code, genIR(ASSIGNOP, result, opn2, result2));

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

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

if (SymbolTable==-1){

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

return;

}

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

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

return;

}

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

//存放函数返回值的单数字节数

if(T->type == INT) {

width = 4;

}

else if(T->type == FLOAT) {

width = 8;

}

else 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;}

T1 = T->ptr[0];

while(T1 != NULL) {

count++;

num = SymbolTable;

if(symbolTable.symbols[SymbolTable+count].flag == 'P' && T1->type != symbolTable.symbols[SymbolTable+count].type){

semantic\_error(T1->pos, "", "实参与形参类型不符");

}

T1 = T1->ptr[1];

}

param\_num = symbolTable.symbols[SymbolTable].paramnum;

if (count > param\_num) {

semantic\_error(T->pos, "", "函数参数数量过多");

}

else if(count < param\_num) {

semantic\_error(T->pos, "", "函数参数数量过少");

}

else

match\_param(SymbolTable,T->ptr[0]); //处理所有参数的匹配

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

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 double\_add\_or\_sub\_exp(struct ASTNode \*T)

{

int SymbolTable, num, width;

struct opn opn1, opn2, result;

struct opn result2;

if(T->ptr[0]->kind!=ID&& T->ptr[0]->kind != EXP\_ARRAY)

{

semantic\_error(T->pos, "", "--语句缺少左值");

}

else{

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

Exp(T->ptr[0]);

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

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

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

opn1.kind = INT;

opn1.const\_int = 1;

result.kind = ID;

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

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

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

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

opn1.kind = ID;

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

opn1.type = INT;

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

result2.kind = ID;

SymbolTable = fill\_Temp(newTemp(), LEV, INT, 'T', T->offset+4);

strcpy(result2.id, symbolTable.symbols[SymbolTable].alias);

result2.offset = symbolTable.symbols[SymbolTable].offset;

if (T->kind == DOUBLEADD\_L || T->kind == DOUBLEADD\_R)

T->code = merge(2, T->code, genIR(PLUS, opn1, result, result2));

else if(T->kind == DOUBLESUB\_L || T->kind == DOUBLESUB\_R)

T->code = merge(2, T->code, genIR(MINUS, opn1, result, result2));

T->code = merge(2, T->code, genIR(ASSIGNOP, result2, opn2, opn1));

}

}

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 SymbolTable,num,width;

int flag=0;

struct ASTNode \*T0;

struct opn opn1,opn2,result;

struct ASTNode \*left = (struct ASTNode \*)malloc(sizeof(struct ASTNode));

struct ASTNode \*right = (struct ASTNode \*)malloc(sizeof(struct ASTNode));

if (T)

{

switch (T->kind) {

//外部定义列表

case EXT\_DEF\_LIST:

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

//此处是在语义分析之前，进行偏移地址的设置

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;

// 可为空

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;

//外部变量列表

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:FLOAT;

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

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

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

}

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

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

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

}

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

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

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

}

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

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

T->width=(T->ptr[1]->width)\*(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:FLOAT;//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->type = T->ptr[1]->type = INT;

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

}

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

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

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

}

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

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

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

}

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]); //FuncDec：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()); //CompSt：函数体语句执行结束后的位置属性

semantic\_Analysis(T->ptr[2]); //CompSt：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;

printf("%d\n", symbolTable.symbols[T->ptr[1]->place].offset);

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

SymbolTable=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 (SymbolTable==-1){

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

return;

}

else T->place=SymbolTable;

result.kind=ID;

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

result.offset=SymbolTable;

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[SymbolTable].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[SymbolTable].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]);//剩下的函数形式参数列表

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); //连接函数形式参数代码

}

else {

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

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

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

}

break;

case PARAM\_DEC:

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

if (SymbolTable==-1)

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

else

T->ptr[1]->place=SymbolTable;

T->num=1; //参数个数计算的初始值

//T->width=T->ptr[0]->type==INT?4:8; //参数宽度

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

T->width = 4;

}

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

T->width = 8;

}

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

T->width = 1;

}

result.kind=ID;

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

result.offset=T->offset;

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

break;

//复合语句

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]) {//处理复合语句中的定义列表

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]){//处理复合语句中的语句列表

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

strcpy(T->ptr[1]->Snext,T->Snext); //S.next属性向下传递，这样执行完整个条件语句就可以返回到最开始的位置

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

// system("pause");

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

flag = 0;

struct ASTNode\* T0;

struct opn opn1, opn2, result;

T->code=NULL;

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

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

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

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

width=4;

}

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

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

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

width = 8;

}

else {

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

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

width = 1;

}

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?4:8; //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;//如果还有剩下的DEC\_LIST

//DEC语句

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

SymbolTable=fillSymbolTable(T0->ptr[0]->type\_id,newAlias(),LEV,T0->ptr[0]->type,'V',T0->offset+T0->width);//此处偏移量未计算，暂时为0

if (SymbolTable==-1)

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

else

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

T->width+=width;

}

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

SymbolTable=fillSymbolTable(T0->ptr[0]->ptr[0]->type\_id,newAlias(),LEV,T0->ptr[0]->type,'V',T0->offset+T0->width);//此处偏移量未计算，暂时为0

if (SymbolTable==-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=SymbolTable;

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

Exp(T0->ptr[0]->ptr[1]);//是一个EXP

opn1.kind=ID;

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

opn1.offset = symbolTable.symbols[T0->ptr[0]->ptr[1]->place].offset;

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(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;

//语句列表

case STM\_LIST:

if (!T->ptr[0])

{

T->code=NULL;

T->width=0;

return;

} //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]);//每次取列表中的第一个语句进行再次遍历分析

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+ T->ptr[0]->width; //顺序结构共享单元方式

semantic\_Analysis(T->ptr[1]);//剩下的语句列表

//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//剩下的是循环判断语句

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

}

break;

//处理复合语句中的局部变量列表

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); //条件语句错误就跳过if子句中的内容

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

boolExp(T->ptr[0]);

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

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

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

semantic\_Analysis(T->ptr[1]); //if子句

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->offset;

boolExp(T->ptr[0]); //条件，要单独按短路代码处理

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

// printf("!!%d\n", T->ptr[0]->width);

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

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

semantic\_Analysis(T->ptr[1]); //if子句

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

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

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

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

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，判断表达式

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

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

prn\_symbol();

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());

flag++;//flag的作用是为了控制break在while中

strcpy(break\_label, T->Snext);

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

semantic\_Analysis(T->ptr[1]); //循环体

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

flag--;

break;

case BREAK:

if(flag == 0)

semantic\_error(T->pos, " ", "break语句处出现错误，未在循环体中");

T->code = merge(2, T->code, genGoto(break\_label));

break;

case CONTINUE:

if (flag == 0)

semantic\_error(T->pos, " ", "continue语句处出现错误，未在循环体中");

T->code = merge(2, T->code, genGoto(continue\_label));

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]);//子树节点中为表达式

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;

return;

}

/\*需要判断返回值类型是否匹配\*/

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 SWITCH\_STMT:

// switch选择表达式

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

Exp(T->ptr[0]);

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

strcpy(T->Snext, newLabel());

// case语句

strcpy(case\_temp, symbolTable.symbols[T->ptr[0]->place].name);

strcpy(break\_label, T->Snext);

strcpy(case\_label, newLabel());

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

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

break;

case CASE\_STMT:

left = (struct ASTNode\*)malloc(sizeof(struct ASTNode));

right = (struct ASTNode\*)malloc(sizeof(struct ASTNode));

left->kind = ID;

strcpy(left->type\_id, case\_temp);

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

right->kind = INT;

right->type\_int = T->ptr[0]->type\_int;

}

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

right->kind = CHAR;

strcpy(right->type\_char, T->ptr[0]->type\_char);

}

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

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

T->ptr[0]->kind = RELOP;

strcpy(T->ptr[0]->type\_id, "==");//比较case中的和switch条件是否相等

//case x:

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;

case FOR:

LEV++;

// 处理循环初始语句

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

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

Exp(T->ptr[0]->ptr[0]);

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

// 处理循环条件

strcpy(T->ptr[0]->ptr[1]->Etrue, newLabel()); //子结点继承属性的计算

strcpy(T->ptr[0]->ptr[1]->Efalse, T->Snext);

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

boolExp(T->ptr[0]->ptr[1]);

//if(T->ptr[0]->width < T->ptr[0]->ptr[1]->width)

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

// 循环体

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

strcpy(break\_label, T->Snext);

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

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

// 自动运算条件

T->ptr[0]->ptr[2]->offset = T->ptr[0]->offset + T->ptr[0]->width;

strcpy(T->ptr[0]->ptr[2]->Snext, newLabel());

Exp(T->ptr[0]->ptr[2]);

// if(T->ptr[0]->width < T->ptr[0]->ptr[2]->width)

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

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

T->code = merge(8, T->ptr[0]->ptr[0]->code,

genLabel(T->ptr[0]->ptr[2]->Snext),

T->ptr[0]->ptr[1]->code,

genLabel(T->ptr[0]->ptr[1]->Etrue),

T->ptr[1]->code,

genLabel(T->ptr[1]->Snext),

T->ptr[0]->ptr[2]->code,

genGoto(T->ptr[0]->ptr[2]->Snext));

break;

case ID:

case INT:

case FLOAT:

case CHAR:

case ASSIGNOP:

case AND:

case OR:

case RELOP:

case PLUS:

case MINUS:

case STAR:

case DIV:

case MOD:

case COMADD:

case COMSUB:

case COMDIV:

case COMSTAR:

case COMMOD:

case DOUBLEADD\_L:

case DOUBLEADD\_R:

case DOUBLESUB\_L:

case DOUBLESUB\_R:

case NOT:

case UMINUS:

case FUNC\_CALL:

case EXP\_ARRAY:

case ARGS:

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);

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

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

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

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);

prn\_symbol();

prnIR(T->code);

object\_code(T->code);

}

**Def.h**

#include "stdio.h"

#include "stdlib.h"

#include "string.h"

#include "stdarg.h"

#include "parser.tab.h"

#define MAXLENGTH 200

#define DX 3\*sizeof(int) /\*活动记录控制信息需要的单元数，这个根据实际系统调整\*/

//以下语法树结点类型、三地址结点类型等定义仅供参考，实验时一定要根据自己的理解来定义

int LEV; //层号

// 使用链表存储多个变量

union Value {

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;

char type\_string[32];

struct Array \*arrayPtr;

};

struct Array

{

int kind;

union Value value;

int index;

struct Array \*next;

};

struct opn{

int kind; //标识联合成员的属性

int type; //标识操作数的数据类型

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

struct Array \*type\_array;

};

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; //偏移量，目标代码生成时用

};

struct codenode { //三地址TAC代码结点,采用单链表存放中间语言代码

int op;

struct opn opn1,opn2,result;

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

int kind;

union {

char type\_char[3];

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

};

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

int place; //存放（临时）变量在符号表的位置序号

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

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

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

int type; //用以标识表达式结点的类型

int pos; //语法单位所在位置行号

int offset; //偏移量

int width; //占数据字节数

int num; //计数器，可以用来统计形参个数

};

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

char name[33]; //变量或函数名

int level; //层号

int type; //变量类型或函数返回值类型

int paramnum; //对函数适用，记录形式参数个数

char alias[10]; //别名，为解决嵌套层次使用

char flag; //符号标记，函数：'F' 变量：'V' 参数：'P' 临时变量：'T'，增加数组：“A”

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，

//或记录函数活动记录大小，目标代码生成时使用

//函数入口等实验可能会用到的属性...

int const\_int;

int array[10];

};

//符号表,是一个顺序栈，index初始值为0

struct symboltable{

struct symbol symbols[MAXLENGTH];

int index;

} symbolTable;

//每到达一个复合语句时，将符号表的index值进栈，离开复合语句时，取出其退栈值修改符号表的index，完成删除该复合语句中的所有变量和临时变量

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,当使用顺序表作为符号表时，进入、退出一个作用域时需要对其操作，以完成符号表的管理。对其它形式的符号表，不一定需要此数据结构

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 objectCode(struct codenode \*head);

void ArrayCall(struct ASTNode\* T);

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

void semantic\_error(int line, char\* msg1, char\* msg2);

int searchSymbolTable(char\* name);

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

char\* newAlias();

char\* newLabel();

char\* newTemp();

void prn\_symbol();

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

void ext\_var\_list(struct ASTNode\* T);

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

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

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

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

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

void prnIR(struct codenode\* head);

void double\_add\_or\_sub\_exp(struct ASTNode \*T);

int getArray\_width(struct ASTNode \*T, int \*array, int index);

int compute\_arraywidth(int \*array,int index);

int get\_width(struct ASTNode \*T);

void sublen(char a[],char b[],int start,int len);

int searchSymbolTableByAlias(char \*alias);

void object\_code(struct codenode \*head);

**Object.c**

#include "def.h"

#include<string.h>

extern struct symboltable symbolTable;

int call\_flag = 0;

int main\_call=0;

int LEV;

int main\_flag = 0;//主函数标志位

//截取指定长度的字符数组

int sublen(char a[],char b[],int start,int len)

{

/\*strncpy(a,b+start,len);

a[len]='\0';\*/

char\* p = b + start; //定义指针变量指向需要提取的字符的地址

int n = strlen(p); //求字符串长度

int i = 0;

if(n < len)

{

len = n;

}

while(len != 0)

{

a[i] = b[i+start];

len --;

i++;

} //复制字符串到dst中

a[i] = '\0';

return 0;

}

int searchSymbolTableByAlias(char \*alias)

{

int i;

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

if (!strcmp(symbolTable.symbols[i].alias, alias))

return i;

return -1;

}

void object\_code(struct codenode \*head) {

char opnstr1[32],opnstr2[32],resultstr[32];//寄存器编号

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

int i;

FILE \*fp;

/\*

如MARS，要求MIPS汇编程序的入口函数要放在最前面，

可以考虑目标代码生成时，

把main函数调整到最前面，或增加一个放在最前面的入口函数（假定为main0），

在入口函数中调用main函数。\*/

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, "j main\n");//跳转到main函数

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");

int cnt = 0;

do{

cnt++;

switch (h->op)

{

//约定操作数使用$t1和$t2，运算结果使用$t3

case ASSIGNOP:

if(h->opn1.kind == INT) {//int型常量

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

}

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

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

}

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

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

}

else{

int SymbolTable;

char array[20];

if(strstr(h->opn1.id, " offset：") != NULL) {

sublen(array, h->opn1.id, 0, 2);

SymbolTable = searchSymbolTableByAlias(array);

}

else

SymbolTable = searchSymbolTableByAlias(h->opn1.id);

if(SymbolTable != -1 && symbolTable.symbols[SymbolTable].level == 0){

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

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

}

else{

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

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

}

}

//数组

int SymbolTable;

char array[20] = {'\0'};

if(strstr(h->result.id, " offset：") != NULL) {

for(int i = 0; i < strlen(h->result.id); i++){

if(h->result.id[i] != ' '){

array[i] = h->result.id[i];

}

}

SymbolTable = searchSymbolTableByAlias(array);

}

else

SymbolTable = searchSymbolTableByAlias(h->result.id);

if(SymbolTable != -1 && symbolTable.symbols[SymbolTable].level == 0){

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

}

else{

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

}

break;

case PLUS:

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

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

fprintf(fp, " add $t3, $t1, $t2\n");

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

break;

case MINUS:

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

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

fprintf(fp, " sub $t3, $t1, $t2\n");

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

break;

case STAR:

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

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

fprintf(fp, " mul $t3, $t1, $t2\n");

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

break;

case DIV:

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

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

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

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

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

break;

case MOD:

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

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

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

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

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

break;

case FUNCTION:

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

main\_flag = 1;

LEV=0;

main\_call=0;

}

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);//-main函数的活动记录大小

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:

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

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

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

break;

case JLT:

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

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

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

break;

case JGE:

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

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

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

break;

case JGT:

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

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

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

break;

case EQ:

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

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

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

break;

case NEQ:

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

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

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

break;

case ARG:

break;

case RETURN:

if(!main\_flag){

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

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

}

else{

fprintf(fp, " li $v0, 10\n");

fprintf(fp, " syscall\n");

main\_flag = 0;

}

call\_flag = 0;

LEV--;

break;

case CALL:

call\_flag = 1;

LEV++;

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;

}

//函数定义参数，获得各个实参值所存放的单元，取出后送到形式参数的单元中

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;

default:

break;

}

h = h->next;

}while(h != head);

fclose(fp);

}

**AST.C**

#include "def.h"

#include "parser.tab.h"

#define MAX\_LOOP 100

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（外部变量和函数）列表中的第一个

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

break;

case EXT\_VAR\_DEF:

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

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

printf("%\*c变量名：\n",indent+3,' ');

display(T->ptr[1],indent+6); //显示变量列表

break;

case TYPE:

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

break;

case EXT\_DEC\_LIST:

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

{

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

printf("%\*c数组大小是：\n", indent, ' ');

}

display(T->ptr[0],indent); //依次显示外部变量名，

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

break;

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

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

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

display(T->ptr[2],indent+3); //显示函数体

break;

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

if (T->ptr[0]) {

printf("%\*c函数形参：\n",indent,' ');

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

}

else printf("%\*c此函数无参数\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: 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：(%d)\n",indent,' ',T->pos);

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

break;

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

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

break;

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

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

display(T->ptr[0],indent+6); //显示定义部分

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

display(T->ptr[1],indent+6); //显示语句部分

break;

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

display(T->ptr[1],indent); //显示剩下语句

break;

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

printf("%\*c循环条件：\n",indent+3,' ');

display(T->ptr[0],indent+6); //显示循环条件

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

display(T->ptr[1],indent+6); //显示循环体

break;

case BREAK:

printf("%\*cbreak语句:(%d)\n", indent, ' ');

break;

case BLANK:

//printf("%\*c空语句\n", indent, ' ');

break;

case CONTINUE:

printf("%\*ccontinue语句:(%d)\n", indent, ' ');

break;

case FOR:

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

printf("%\*c循环条件(FOR)：\n", indent+3, ' ');

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

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

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

break;

case SWITCH\_STMT:

printf("%\*cSWITCH语句：(%d)\n", indent, ' ', T->pos);

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

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

break;

case CASE\_STMT:

printf("%\*cCASE语句：(%d)\n", indent, ' ', T->pos);

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

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

break;

case FOR\_DEC:

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

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

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

break;

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

printf("%\*c条件：\n",indent+3,' ');

display(T->ptr[0],indent+6); //显示条件

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

display(T->ptr[1],indent+6); //显示if子句

break;

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

printf("%\*c条件：\n",indent+3,' ');

display(T->ptr[0],indent+6); //显示条件

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

display(T->ptr[1],indent+6); //显示if子句

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

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

break;

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

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

break;

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

display(T->ptr[0],indent+3); //显示变量类型

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

break;

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

T0=T;

while (T0) {

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

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

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

{

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

display(T0->ptr[0]->ptr[1],indent+strlen(T0->ptr[0]->ptr[0]->type\_id)+7);

}

T0=T0->ptr[1];

}

break;

case ARRAY\_LIST:

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

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

break;

case ARRAY\_ID:

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

printf("%\*c访问下标：\n", indent, ' ');

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

break;

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

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: %s\n", indent, ' ', T->type\_char);

break;

case ASSIGNOP:

case AND:

case OR:

case RELOP:

case PLUS:

case MINUS:

case STAR:

case MOD:

case DIV:

/\*add\*/

case COMADD:

case COMSUB:

case COMSTAR:

case COMDIV:

case COMMOD:

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

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

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

break;

case DOUBLEADD\_L:

case DOUBLESUB\_L:

case DOUBLEADD\_R:

case DOUBLESUB\_R:

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

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

break;

case NOT:

case UMINUS:

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

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

break;

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

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

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

break;

case ARGS: i=1;

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

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

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

display(T0,indent+3);

T=T->ptr[1];

}

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

// display(T,indent+3);

printf("\n");

break;

}

}

}