华中科技大学

课程实验报告

题目：Mini-C语言编译器

课程名称：编译原理实验

专业班级：CS1706

学　　号：U201714762

姓　　名：梁一飞

指导教师：徐丽萍

报告日期：2020.6.23

计算机科学与技术学院

目录

目录

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

采用简化的 C 语言的文法：mini-c 作为本次实验的语言。

其文法如下所示：

G[program]:

program → ExtDefList

ExtDefList→ExtDef ExtDefList | ε

ExtDef→Specifier ExtDecList ; |Specifier FunDec CompSt

Specifier→int | float | char

ExtDecList→VarDec | VarDec , ExtDecList

VarDec→ID

FucDec→ID (VarList) | ID ( )

VarList→ParamDec , VarList | ParamDec

ParamDec→Specifier VarDec

CompSt→{ DefList StmList }

StmList→Stmt StmList | ε

Stmt→Exp ; | CompSt | return Exp ;

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

DefList→Def DefList | ε

Def→Specifier DecList ;

DecList→Dec | Dec , DecList

Dec→VarDec | VarDec = Exp

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

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

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

| (Exp) | - Exp | ! Exp | ID (Args) | ID ( ) | Exp + + | Exp - - | Exp + = Exp | Exp - = Exp | Exp * = Exp | Exp / = Exp

Args→Exp , Args | Exp

说明：program 为文件开始符号；ExtDefList 指程序语句列表；ExtDef 指某一行的语句；ExtDefList 指后面多行语句；标识符 Specifier；变量声明列表 ExtDecList；函数 FunDec；函数体 CompSt；函数声明 FunDec；变量列表 VarList；参数声明 ParamDec；变量声明 VarDec；Dec 代表声明语句；Stmt 代表函数内部声明语句；StmtList 代表声明语句列表；Exp 代表表达式；ID 为对应的变量或函数名字。

2.2 单词文法与语言文法

1.单词文法

按照语法定义列出所有的终结符以及非终结符，如表 1 所示：

表 1: 语法符号

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]+
{float}	FLOAT	([0-9]*\.[0-9]+)|([0-9]+\.)
"int"	TYPE	
"float"	TYPE	
"return"	RETURN	
"if"	IF	
"else"	ELSE	
"while"	WHILE	
"for"	FOR	
";"	SEMI	
","	COMMA	
">"|"<"|">="|"<="|"=="|"!="	RELOP	
"="	ASSIGNOP	
"+"	PLUS	
“+=”	COMADD	
“-=”	COMSUB	
“++”	AUTOADD	
“—"	AUTOSUB	
"-"	MINUS	
"*"	STAR	
"/"	DIV	
"&&"	AND	
"||"	OR	
"!"	NOT	
"("	LP	
")"	RP	
"{"	LC	
"}"	RC	
"["	LB	

"]"	RB	
"//"[^\n]*	代表单行注释	
“/”*(\s|.)*?*”/”	代表多行注释	

2.语言文法

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

定义的非终结符如下：

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

举一个例子进行说明，其中%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。

其次，利用%token 定义终结符的语义值类型。%token <type_id> ID，表示识别出来一个标识符后，标识符的字符串串值保存在成员 type_id 中。

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

%token <type_id> ID RELOP TYPE //指定 ID,RELOP 的语义值是 type_id,由词法分析得到的标识符字符串。

%token <type_float> FLOAT //指定 ID 的语义值是 type_id,由词法分析得到的标识符字符串。

%token <type_char> CHAR

%token LP RP LC RC SEMI COMMA LB RB

%token PLUS MINUS STAR DIV ASSIGNOP AND OR NOT IF ELSE WHILE RETURN COMADD COMSUB FOR //+=comadd，-=comsub，for，=assignop

然后，定义运算符的优先级与结合性，如表 2 所示

表 2 算符优先级与结合性

优先级	结合性	符号
高	左	“+=”，”-="
	左	“=”
	左	“||”
	左	“&&”
	左	">"|"<"|">="|"<="|"=="|"!="
	左	“+”, “-“
	左	“*”，“/”
低	右	“!”，“++”，“—”

再次，是语法定义的核心部分：语法规则部分。语法规则由 1.1 中所定义的语法规则编写，具体实现代码如下，具体的分析写到每一句当中：

```
program: ExtDefList { display($1,0);} /*归约到 program，开始显示语法树,语义分析*/
        ;
/*ExtDefList：外部定义列表，即是整个语法树*/
ExtDefList: {$$=NULL;}/*整个语法树为空*/
          | ExtDef ExtDefList {$$=mknode(EXT_DEF_LIST,$1,$2,NULL,yylineno);
```

```
}//每一个 EXTDEFLIST 的结点,其第 1 棵子树对应一个外部变量声明或函数
        ;
  /*外部声明，声明外部变量或者声明函数*/
  ExtDef: Specifier ExtDecList SEMI {$$=mknode(EXT_VAR_DEF,$1,$2,NULL,yylineno);}//该结点对应一个外部变量声明
        | Specifier ArrayDec SEMI {$$=mknode(ARRAY_DEF,$1,$2,NULL,yylineno);}//数组定义
        | Specifier FuncDec CompSt {$$=mknode(FUNC_DEF,$1,$2,$3,yylineno);}//该结点对应一个函数定义,类型+函数声明+复合语句
        | error SEMI {$$=NULL; printf("---缺少分号---\n");}
        ;
  /*表示一个类型，int、float 和 char*/
  Specifier: TYPE {$$=mknode(TYPE,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);$$->type=(!strcmp($1,"int")?INT:(!strcmp($1,"float")?FLOAT:CHAR));}
        ;
  /*变量名称列表，由一个或多个变量组成，多个变量之间用逗号隔开*/
  ExtDecList: VarDec {$$=$1;}/*每一个 EXT_DECLIST 的结点,其第一棵子树对应一个变量名(ID 类型的结点),第二棵子树对应剩下的外部变量名*/
        | VarDec COMMA ExtDecList {$$=mknode(EXT_DEC_LIST,$1,$3,NULL,yylineno);}
        ;
  /*变量名称，由一个 ID 组成*/
  VarDec: ID {$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);}//ID 结点,标识符符号串存放结点的 type_id
        ;
  /*函数名+参数定义*/
  FuncDec: ID LP VarList RP {$$=mknode(FUNC_DEC,$3,NULL,NULL,yylineno);strcpy($$->type_id,$1);}//函数名存放在$$->type_id
        | ID LP RP {$$=mknode(FUNC_DEC,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);}//函数名存放在$$->type_id
        | error RP {$$=NULL; printf("---函数左括号右括号不匹配---\n");}
        ;
  /*数组声明*/
  ArrayDec: ID LB Exp RB {$$=mknode(ARRAY_DEC,$3,NULL,NULL,yylineno);strcpy($$->type_id,$1);}
        | ID LB RB {$$=mknode(ARRAY_DEC,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);}
        | error RB {$$=NULL;printf("---数组定义错误---\n");}
  /*参数定义列表，有一个到多个参数定义组成，用逗号隔开*/
  VarList: ParamDec {$$=mknode(PARAM_LIST,$1,NULL,NULL,yylineno);}
        | ParamDec COMMA VarList {$$=mknode(PARAM_LIST,$1,$3,NULL,yylineno);}
        ;
  /*参数定义，固定有一个类型和一个变量组成*/
```

```
  ParamDec: Specifier VarDec {$$=mknode(PARAM_DEC,$1,$2,NULL,yylineno);}
          ;
  /*复合语句，左右分别用大括号括起来，中间有定义列表和语句列表*/
  CompSt: LC DefList StmList RC {$$=mknode(COMP_STM,$2,$3,NULL,yylineno);}
        | error RC {$$=NULL; printf("---复合语句内存在错误---\n");}
        ;
  /*语句列表，由 0 个或多个语句 stmt 组成*/
  StmList: {$$=NULL;}
        | Stmt StmList {$$=mknode(STM_LIST,$1,$2,NULL,yylineno);}
        ;
  /*语句，可能为表达式，复合语句，return 语句，if 语句，if-else 语句，while 语句，for*/
  Stmt: Exp SEMI {$$=mknode(EXP_STMT,$1,NULL,NULL,yylineno);}
      | CompSt {$$=$1;}//复合语句结点直接最为语句结点,不再生成新的结点
      | RETURN Exp SEMI {$$=mknode(RETURN,$2,NULL,NULL,yylineno);}
      | IF LP Exp RP Stmt %prec LOWER_THEN_ELSE {$$=mknode(IF_THEN,$3,$5,NULL,yylineno);}
      | IF LP Exp RP Stmt ELSE Stmt {$$=mknode(IF_THEN_ELSE,$3,$5,$7,yylineno);}
      | WHILE LP Exp RP Stmt {$$=mknode(WHILE,$3,$5,NULL,yylineno);}
      | FOR LP Exp RP Stmt {$$=mknode(FOR,$3,$5,NULL,yylineno);}
      ;
  /*定义列表，由 0 个或多个定义语句组成*/
  DefList: {$$=NULL; }
        | Def DefList {$$=mknode(DEF_LIST,$1,$2,NULL,yylineno);}
        ;
  /*定义一个或多个语句语句，由分号隔开*/
  Def: Specifier DecList SEMI {$$=mknode(VAR_DEF,$1,$2,NULL,yylineno);}
     | Specifier ArrayDec SEMI {$$=mknode(ARRAY_DEF,$1,$2,NULL,yylineno);}
     ;
  /*语句列表，由一个或多个语句组成，由逗号隔开，最终都成一个表达式*/
  DecList: Dec {$$=mknode(DEC_LIST,$1,NULL,NULL,yylineno);}
        | Dec COMMA DecList {$$=mknode(DEC_LIST,$1,$3,NULL,yylineno);}
        ;
  /*语句，一个变量名称或者一个赋值语句（变量名称等于一个表达式）*/
  Dec: VarDec {$$=$1;}
     | VarDec ASSIGNOP Exp {$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type_id,"ASSIGNOP");}
     ;
  /*表达式*/
  Exp: Exp ASSIGNOP Exp {$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type_id,"ASSIGNOP");}//$$结点 type_id 空置未用,正好存放运算符
     | Exp AND Exp {$$=mknode(AND,$1,$3,NULL,yylineno);strcpy($$->type_id,"
```

```
AND");}
    | Exp OR Exp {$$=mknode(OR,$1,$3,NULL,yylineno);strcpy($$->type_id,"OR
");}
    | Exp RELOP Exp {$$=mknode(RELOP,$1,$3,NULL,yylineno);strcpy($$->type
_id,$2);}//词法分析关系运算符号自身值保存在$2 中
    | Exp PLUS Exp {$$=mknode(PLUS,$1,$3,NULL,yylineno);strcpy($$->type_id
,"PLUS");}
    | Exp MINUS Exp {$$=mknode(MINUS,$1,$3,NULL,yylineno);strcpy($$->typ
e_id,"MINUS");}
    | Exp STAR Exp {$$=mknode(STAR,$1,$3,NULL,yylineno);strcpy($$->type_id
,"STAR");}
    | Exp DIV Exp {$$=mknode(DIV,$1,$3,NULL,yylineno);strcpy($$->type_id,"D
IV");}
    | Exp COMADD Exp {$$=mknode(COMADD,$1,$3,NULL,yylineno);strcpy($$
->type_id,"COMADD");}
    | Exp COMSUB Exp {$$=mknode(COMSUB,$1,$3,NULL,yylineno);strcpy($$-
>type_id,"COMSUB");}
    | LP Exp RP {$$=$2;}/*遇到左右括号，可直接忽略括号，Exp 的值就为括号
里面的 Exp*/
    | MINUS Exp %prec UMINUS {$$=mknode(UMINUS,$2,NULL,NULL,yyline
no);strcpy($$->type_id,"UMINUS");}
    | NOT Exp {$$=mknode(NOT,$2,NULL,NULL,yylineno);strcpy($$->type_id,"
NOT");}
    | AUTOADD Exp {$$=mknode(AUTOADD_L,$2,NULL,NULL,yylineno);strcp
y($$->type_id,"AUTOADD");}
    | AUTOSUB Exp {$$=mknode(AUTOSUB_L,$2,NULL,NULL,yylineno);strcp
y($$->type_id,"AUTOSUB");}
    | Exp AUTOADD {$$=mknode(AUTOADD_R,$1,NULL,NULL,yylineno);strc
py($$->type_id,"AUTOADD");}
    | Exp AUTOSUB {$$=mknode(AUTOSUB_R,$1,NULL,NULL,yylineno);strcp
y($$->type_id,"AUTOSUB");}
    | ID LP Args RP {$$=mknode(FUNC_CALL,$3,NULL,NULL,yylineno);strcpy(
$$->type_id,$1);}/*函数定义后面的括号部分，只需要把括号里面的内容传入即可
*/
    | ID LP RP {$$=mknode(FUNC_CALL,NULL,NULL,NULL,yylineno);strcpy($
$->type_id,$1);}/*函数定义后面的括号部分没有参数*/
    | ID {$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);}
    | INT {$$=mknode(INT,NULL,NULL,NULL,yylineno);$$->type_int=$1;$$->ty
pe=INT;}
    | FLOAT {$$=mknode(FLOAT,NULL,NULL,NULL,yylineno);$$->type_float=
$1;$$->type=FLOAT;}
    | CHAR {$$=mknode(CHAR,NULL,NULL,NULL,yylineno); $$->type_char=$
1;$$->type=CHAR;}
    ;
```

/*用逗号隔开的参数*/
Args: Exp COMMA Args {$$=mknode(ARGS,$1,$3,NULL,yylineno);}
| Exp {$$=mknode(ARGS,$1,NULL,NULL,yylineno);}
;

举其中一个例子进行说明，对如下语句
Exp:Exp ASSIGNOP Exp {$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno);}
规则后面{}中的是当完成归约时要执行的语义动作。规则左部的 Exp 的属性值用$$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。
其中，错误处理部分，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。

2.3 符号表结构定义

符号表定义如下：
//符号表,是一个顺序栈,index 初值为 0
typedef struct symboltable{
struct symbol symbols[MAXLENGTH];
int index;
}SYMBOLTABLE;
其中 symbol 的结构为：
typedef struct symbol {//这里只列出了一个符号表项的部分属性,没考虑属性间的互斥
char name[33];//变量或函数名
int level;//层号,外部变量名或函数名层号为 0,形参名为 1,每到 1 个复合语句层号加 1,退出减 1
int type; //变量类型或函数返回值类型
int paramnum;//形式参数个数
char alias[10];//别名,为解决嵌套层次使用,使得每一个数据名称唯一
char flag; //符号标记,函数:'F' 变量:'V' 参数:'P' 临时变量:'T'
char offset; //外部变量和局部变量在其静态数据区或活动记录中的偏移量 或 函数活动记录大小,目标代码生成时使用
};

2.4 错误类型码定义

1）错误类型 1：变量在使用时未定义
2）错误类型 2：函数在调用时为经定义
3）错误类型 3：变量出现重复定义或变量域前面定义过的其它语法结构重名
4）错误类型 4：函数出现重复定义

5）错误类型 5：赋值号两边的表达式类型不匹配

6）错误类型 6：赋值号左边只有一个右值的表达式

7）错误类型 7：操作数类型不匹配或操作数类型域操作符不撇皮（例如整形变量域数组变量相加减，或数组（或结构体）变量域数组（或结构体）结构体变量相加减）。

8）错误类型 8：return 语句的返回类型域函数定义的返回类型不匹配

9）错误类型 9：函数调用时实参或形参的数目或类型不匹配。

2.5 中间代码结构定义

中间代码采用三地址码 TAC 作为中间语言，中间代码格式定义如表 3 所示。

表 3 中间代码定义

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

生成规则：

1.基本表达式翻译模式

1）如果 Exp 产生了一个整数 INT，那么我们只需要为传入的 place 变量赋值成前面加上一个“#”的相应数值即可。

2）如果 Exp 产生了一个标识符 ID，那么我们只需要为传入的 place 变量赋值成 ID 对应的变量名（或该变量对应的中间代码中的名字）

3）如果 Exp 产生了赋值表达式 Exp ASSIGNOP Exp，由于之前提到过作为左值的 Exp 只能是三种情况之一（单个变量访问、数组元素访问或结构体特定于的访问）。我们需要通过擦汗表找到 ID 对应的变量，然后对 Exp 进行翻译（运算结果保存在临时变量 t1 中），再将 t1 中的值赋于 ID 所对应的变量并将结果再存辉 place，最后把刚翻译好的这两段代码合并随后返回即可。

4）如果 Exp 产生了算数运算表达式 Exp PLUS Exp，则先对 Exp 进行翻译（运算结果储存在临时变量 t1 中），再对 Exp 进行翻译（运算结果储存在临时变量 t2

中），最后生成一句中间代码 place：=t1+t2，并将刚翻译好的这三段代码合并后返回即可。使用类似的翻译模式也可以对剑法、乘法和除法进行翻译。

5）如果 Exp 产生了屈服表达式 MINUS Exp，则先对 Exp 进行翻译（运算结果储存在临时变量 t1 中），再生成一句中间代码 place：=#0-t1 从而实现对 t1 取负，最后将翻译好的这两段代码合并后返回。使用类似的翻译模式可以对括号表达式进行翻译。

6）如果 Exp 产生了条件表达式（包括与、或、非运算以及比较运算的表达式），我们则会调用翻译函数进行翻译。如果条件为真，那么为 palce 赋值 1；否则，为其赋值 0。

translate_Exp(Exp, sym_table, place) = case Exp of	
INT	value = get_value(INT) return [place := #value] ①
ID	variable = lookup(sym_table, ID) return [place := variable.name]
Exp_1 ASSIGNOP Exp_2 ② (Exp_1 → ID)	variable = lookup(sym_table, Exp_1.ID) t1 = new_temp() code1 = translate_Exp(Exp_2, sym_table, t1) code2 = [variable.name := t1] + ③ [place := variable.name] return code1 + code2
Exp_1 PLUS Exp_2	t1 = new_temp() t2 = new_temp() code1 = translate_Exp(Exp_1, sym_table, t1) code2 = translate_Exp(Exp_2, sym_table, t2) code3 = [place := t1 + t2] return code1 + code2 + code3
MINUS Exp_1	t1 = new_temp() code1 = translate_Exp(Exp_1, sym_table, t1) code2 = [place := #0 - t1] return code1 + code2
Exp_1 RELOP Exp_2 NOT Exp_1 Exp_1 AND Exp_2 Exp_1 OR Exp_2	label1 = new_label() label2 = new_label() code0 = [place := #0] code1 = translate_Cond(Exp, label1, label2, sym_table) code2 = [LABEL label1] + [place := #1] return code0 + code1 + code2 + [LABEL label2]

图 1 基本表达式翻译模式

2.语句翻译模式

Mini-c 的语句包括表达式语句、复合语句、返回语句、跳转语句和循环语句。

translate_Stmt(Stmt, sym_table) = case Stmt of	
Exp SEMI	return translate_Exp(Exp, sym_table, NULL)
CompSt	return translate_CompSt(CompSt, sym_table)
RETURN Exp SEMI	t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) code2 = [RETURN t1] return code1 + code2
IF LP Exp RP $Stmt_1$	label1 = new_label() label2 = new_label() code1 = translate_Cond(Exp, label1, label2, sym_table) code2 = translate_Stmt($Stmt_1$, sym_table) return code1 + [LABEL label1] + code2 + [LABEL label2]
IF LP Exp RP $Stmt_1$ ELSE $Stmt_2$	label1 = new_label() label2 = new_label() label3 = new_label() code1 = translate_Cond(Exp, label1, label2, sym_table) code2 = translate_Stmt($Stmt_1$, sym_table) code3 = translate_Stmt($Stmt_2$, sym_table) return code1 + [LABEL label1] + code2 + [GOTO label3] + [LABEL label2] + code3 + [LABEL label3]
WHILE LP Exp RP $Stmt_1$	label1 = new_label() label2 = new_label() label3 = new_label() code1 = translate_Cond(Exp, label2, label3, sym_table) code2 = translate_Stmt($Stmt_1$, sym_table) return [LABEL label1] + code1 + [LABEL label2] + code2 + [GOTO label1] + [LABEL label3]

图 2 语句翻译模式

3.条件表达式翻译模式

将跳转的两个目标 label_true 和 label_false 作为继承属性（函数参数）进行处理，再这种情况下每当我们在条件表达式内部需要跳到外部时，跳转目标都已经从父节点哪里通过参数得到了。而回填技术在此处没有关注。

translate_Cond(Exp, label_true, label_false, sym_table) = case Exp of	
Exp_1 RELOP Exp_2	t1 = new_temp() t2 = new_temp() code1 = translate_Exp(Exp_1, sym_table, t1) code2 = translate_Exp(Exp_2, sym_table, t2) op = get_relop(RELOP); code3 = [IF t1 op t2 GOTO label_true] return code1 + code2 + code3 + [GOTO label_false]
NOT Exp_1	return translate_Cond(Exp_1, label_false, label_true, sym_table)
Exp_1 AND Exp_2	label1 = new_label() code1 = translate_Cond(Exp_1, label1, label_false, sym_table) code2 = translate_Cond(Exp_2, label_true, label_false, sym_table) return code1 + [LABEL label1] + code2
Exp_1 OR Exp_2	label1 = new_label() code1 = translate_Cond(Exp_1, label_true, label1, sym_table) code2 = translate_Cond(Exp_2, label_true, label_false, sym_table) return code1 + [LABEL label1] + code2
(other cases)	t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) code2 = [IF t1 != #0 GOTO label_true] return code1 + code2 + [GOTO label_false]

图 3 条件表达式翻译模式

4.函数调用翻译模式

在实验中遇到 read 和 write 函数时不直接生成函数调用代码。对于非 read 和 write 函数而言，我们需要调用翻译参数的函数将计算实参的代码翻译出来，并构造浙西参数所对应的临时变量列表 arg_list。

```
translate_Exp(Exp, sym_table, place) = case Exp of
ID LP RP | function = lookup(sym_table, ID)
         | if (function.name == "read") return [READ place]
         | return [place := CALL function.name]
ID LP Args RP | function = lookup(sym_table, ID)
              | arg_list = NULL
              | code1 = translate_Args(Args, sym_table, arg_list)
              | if (function.name == "write") return code1 + [WRITE arg_list[1]]
              | for i = 1 to length(arg_list) code2 = code2 + [ARG arg_list[i]]
              | return code1 + code2 + [place := CALL function.name]
```

图 4 函数调用翻译模式

```
translate_Args(Args, sym_table, arg_list) = case Args of
Exp | t1 = new_temp()
    | code1 = translate_Exp(Exp, sym_table, t1)
    | arg_list = t1 + arg_list
    | return code1
Exp COMMA Args1 | t1 = new_temp()
                | code1 = translate_Exp(Exp, sym_table, t1)
                | arg_list = t1 + arg_list
                | code2 = translate_Args(Args1, sym_table, arg_list)
                | return code1 + code2
```

图 5 函数参数的翻译模式

2.6 目标代码指令集选择

目标语言选定 MIPS32 指令序列，可以在 SPIM Simulator 上运行。TAC 指令和 MIPS32 指令的对应关系，如表 4 所示。其中 reg(x)表示变量 x 所分配的寄存器。

表 4 中间代码与 MIPS32 指令对应关系

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

3.1 词法分析器

使用 flex 工具编写 lex 文件，对指定的高级语言程序进行词法分析。

1.定义部分

定义部分实际上就是给后面某些可能经常用到的正则表达式取一个别名，从而简化词法规则的书写。定义部分的格式一般为:

Name definition

其中 name 是名字，definition 是任意的正则表达式。

在 lex 文件中定义部分如下：

```
%{
    #include "parser.tab.h"
    #include <string.h>
    #include "def.h"
    int yycolumn = 1;
    #define YY_USER_ACTION yylloc.first_line=yylloc.last_line=yylineno; yylloc.
first_column=yycolumn; yylloc.last_column=yycolumn+yyleng-
1; yycolumn+=yyleng;
    typedef union{
        int type_int;
        float type_float;
        char type_char;
        char type_id[32];
        struct node *ptr;
    }YYLVAL;
    #define YYSTYPE YYLVAL
%}
```

2.规则部分

规则部分是由正则表达式和相应的响应函数组成，其格式为

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，这些操作可以用一段 C 代码表示。Flex 将按照这部分给出饿内容依次尝试每一个规则，尽可能匹配最长的输入串。如果有些内容不匹配任何规则，Flex 默认之将其拷贝到标准输出，想要修改这个默认行为只需要在所有规则的最后加上一条“.” (即匹配任何输入)规则，然后在其对应的 action 部分书写想定义的行为即可。

规则部分的代码如下：

```
%%
 /*注释处理 单行+多行*/
\/\/[^\n]*  {;}//匹配注释的正则表达式
```

```
\/\*(\s|.)*?\*\/ {;}//匹配注释的正则表达式
{int} {yylval.type_int=atoi(yytext);return INT;}
{float} {yylval.type_float=atof(yytext); return FLOAT;}
"int" {strcpy(yylval.type_id,yytext); return TYPE;}
"float" {strcpy(yylval.type_id,yytext); return TYPE;}
"char" {strcpy(yylval.type_id,yytext); return TYPE;}
"return" {return RETURN;}
"if" {return IF;}
"else" {return ELSE;}
"while" {return WHILE;}
"for" {return FOR;}
{id} {strcpy(yylval.type_id,yytext); return ID;}

";" {return SEMI;}
"," {return COMMA;}
">"|"<"|">="|"<="|"=="|"!=" {strcpy(yylval.type_id,yytext); return RELOP;}
"=" {return ASSIGNOP;}
"+" {return PLUS;}
"-" {return MINUS;}
"+=" {return COMADD;}
"-=" {return COMSUB;}
"++" {return AUTOADD;}
"--" {return AUTOSUB;}
"*" {return STAR;}
"/" {return DIV;}
"&&" {return AND;}
"||" {return OR;}
"!" {return NOT;}
"(" {return LP;}
")" {return RP;}
"[" {return LB;}
"]" {return RB;}
"{" {return LC;}
"}" {return RC;}
[\n] {yycolumn=1;}
[ \r\t] {;}
.  {printf("Error type A: Mysterious character\"%s\" at line %d,column %d\n",yyte
xt,yylineno,yycolumn);}
```

其中的 yytext 的类型为 char*，它是 flex 提供的一个变量，里面保存了当前词法单元所对应的词素。函数 atoi（）作用是把一个字符串表示的整数转化为 int 类型。

3.用户自定义代码部分

这部分代码会被原封不动的拷贝到 lex.yy.c 中，以方便用户自定义所需要执行的函数（包括之前的 main 函数）。如果用户想要对这部分用到的变量、函数或者

头文件进行声明，可以前面的定义部分（即 Flex 源代码文件的第一部分）之前使用“%{“和”%}“符号将要声明的内容添加进去。被”%{“和”%}“所包围的内容也会被一并拷贝到 lex.yy.c 的最前面。

以下是用户子程序部分：

```
/* 复制到 lex.yy.c 中,main 冲突不能用了
void main(int argc,char *argv[]){
    yylex();
    return;
}
*/
int yywrap(){
    return 1;
}
```

3.2 语法分析器

同 Flex 源代码类似，Bison 源代码也分为三个部分，其作用与 Flex 源代码大致相同，其分为定义部分、规则部分、用户函数部分。

1.定义部分

所有的词法单元的定义都可以放到这部分。这段 bison 代码以“%{“和”%}“开头，被“%{“和”%}“包含的内容主要是对 stdio.h 的引用。接下来是一些以%token 开头的此法单元（终结符）定义，如果需要采用 Flex 生成的 yylex（），那么在这里定义的词法单元都可以作为 Flex 源代码里的返回值。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。

```
%{
#include "stdio.h"
#include "math.h"
#include "string.h"
#include "def.h"
extern int yylineno;
extern char *yytext;
extern FILE *yyin;
void yyerror(const char* fmt, ...);
void display(struct node *,int);
%}
```

此外，bison 文件中可能会有辅助定义部分，辅助定义部分如下：

```
%union {
  int type_int;
  float type_float;
  char type_char;
  char type_id[32];
  struct node *ptr;
};
```

辅助声明部分利用%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。

2.规则部分

这部分包括具体的语法和相应的语义动作。具体来讲是在书写产生式。第一个产生式左边的非终结符默认为初始符号。产生式里的箭头在这里用冒号":"表示，一组产生式与另一组之间以分号"；"隔开。产生式里无论是终结符还是非中介都都各自对应一个属性值，乘胜是左边的非终结符对应的属性值用￥￥表示，右边的几个符号的属性值按从左到右的顺序一次对应位$1、$2、$3 等。每条产生式的最后可以添加一组以花括号"{ "和"}"括起来的语义动作，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}。需要注意的是，在产生式中间添加语义动作在某些情况下有可能在原有语法中引入冲突，因此能使用时要特别谨慎。具体的代码实现部分已经在 2.2 中体现，此处不再赘述。

3.用户函数部分

这部分的代码会被原封不动的拷贝到 syntax.tab.c 中，以方便用户自定义所需要的函数。如果想要对这部分所用到的变量、函数或者头文件进行声明，可以在定义部分之前使用" %{ "和"%}"将要申明的内容添加进去。被" %{ "和"%}"所包围的内容也会被一并拷贝到 syntax.tab.c 的最前面。用户自定义函数如下：

```
%%
int main(int argc, char *argv[]){
  yyin=fopen(argv[1],"r");
  if (!yyin)
    return 0;
  yylineno=1;
  yyparse();
  return 0;
}

#include<stdarg.h>
void yyerror(const char* fmt, ...)
{
  va_list ap;
  va_start(ap, fmt);
  fprintf(stderr, "Grammar Error at Line %d Column %d: ", yylloc.first_line,yylloc.first_column);
  vfprintf(stderr, fmt, ap);
  fprintf(stderr, ".\n");
}
```

其中 yyerror 函数会在语法分析程序中每发现一个语法错误时被调用，其默认参数为"syntax error "。默认情况下 yyerror 智慧将传入的字符串参数打印到标准错误输出上，而自己也可以重新定义这个函数，从而使它打印一些别的内容。

4.抽象语法树节点的建立

在语法分析阶段，要生成建立抽象语法树 AST。抽象语法树将词法分析之后生成的单词元素都按照一定的规则组装起来，再利用树的的结构表示出文件中各语

法元素的关系。

首先是 AST 树结点的定义：

```
struct node {//以下对结点属性定义没有考虑存储效率,只是简单地列出要用到的一些属性
  enum node_kind kind;//结点类型
  union {
    char type_id[33]; //由标识符生成的叶结点
    int type_int; //由整常数生成的叶结点
    char type_char;//由字符型生成的叶节点
    float type_float; //由浮点常数生成的叶结点
  };
  struct node *ptr[3];//子树指针,由 kind 确定有多少棵子树
  int level;//层号
  int place; //表示结点对应的变量或运算结果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]; //该结点对应语句执行后的下一条语句位置标号
  struct codenode *code; //该结点中间代码链表头指针
  char op[10];
  int type;//结点对应值的类型
  int pos; //语法单位所在位置行号
  int offset; //偏移量
  int width; //各种数据占用的字节数
};
```

其次是实例化一个语法树节点：

```
struct node *mknode(int kind,struct node *first,struct node *second, struct node *third,int pos ){
  struct node *tempnode = (struct node*)malloc(sizeof(struct node));
  tempnode->kind = kind;
  tempnode->ptr[0] = first;
  tempnode->ptr[1] = second;
  tempnode->ptr[2] = third;
  tempnode->pos = pos;
  return tempnode;
}
```

5. 显示抽象语法树

抽象语法树的遍历是树的先序遍历，将遍历的结果输出，对不同的节点输出结果不一样。Display 的函数设计如下：

```
void display(struct node* T,int indent){
  if(T){
    switch (T->kind){
      case EXT_DEF_LIST:
        display(T->ptr[0],indent);
        display(T->ptr[1],indent);
        break;
```

```
case EXT_VAR_DEF:
   printf("%*c%s\n",indent,' ',"外部变量定义： ");
   display(T->ptr[0],indent+5);
   printf("%*c%s\n",indent+5,' ',"变量名： ");
   display(T->ptr[1],indent+5);
   break;
case FUNC_DEF:
   printf("%*c%s\n",indent,' ',"函数定义： ");
   display(T->ptr[0],indent+5);
   display(T->ptr[1],indent+5);
   display(T->ptr[2],indent+5);
   break;
case ARRAY_DEF:
   printf("%*c%s\n",indent,' ',"数组定义： ");
   display(T->ptr[0],indent+5);
   display(T->ptr[1],indent+5);
   break;
case FUNC_DEC:
   printf("%*c%s%s\n",indent,' ',"函数名： ",T->type_id);
   printf("%*c%s\n",indent,' ',"函数型参： ");
   display(T->ptr[0],indent+5);
   break;
case ARRAY_DEC:
   printf("%*c%s%s\n",indent,' ',"数组名： ",T->type_id);
   printf("%*c%s\n",indent,' ',"数组大小： ");
   display(T->ptr[0],indent+5);
   break;
case EXT_DEC_LIST:
   display(T->ptr[0],indent+5);
   if(T->ptr[1]->ptr[0]==NULL)
      display(T->ptr[1],indent+5);
   else
      display(T->ptr[1],indent);
   break;
case PARAM_LIST:
   display(T->ptr[0],indent);
   display(T->ptr[1],indent);
   break;
case PARAM_DEC:
   display(T->ptr[0],indent);
   display(T->ptr[1],indent);
   break;
case VAR_DEF:
   display(T->ptr[0],indent+5);
```

```
    display(T->ptr[1],indent+5);
    break;
case DEC_LIST:
    printf("%*c%s\n",indent,' ',"变量名： ");
    display(T->ptr[0],indent+5);
    display(T->ptr[1],indent);
    break;
case DEF_LIST:
    printf("%*c%s\n",indent+5,' ',"LOCAL VAR_NAME： ");
    display(T->ptr[0],indent+5);
    display(T->ptr[1],indent);
    break;
case COMP_STM:
    printf("%*c%s\n",indent,' ',"复合语句： ");
    printf("%*c%s\n",indent+5,' ',"复合语句的变量定义： ");
    display(T->ptr[0],indent+5);
    printf("%*c%s\n",indent+5,' ',"复合语句的语句部分： ");
    display(T->ptr[1],indent+5);
    break;
case STM_LIST:
    display(T->ptr[0],indent+5);
    display(T->ptr[1],indent);
    break;
case EXP_STMT:
    printf("%*c%s\n",indent,' ',"表达式语句： ");
    display(T->ptr[0],indent+5);
    break;
case IF_THEN:
    printf("%*c%s\n",indent,' ',"条件语句（if-else）： ");
    printf("%*c%s\n",indent,' ',"条件： ");
    display(T->ptr[0],indent+5);
    printf("%*c%s\n",indent,' ',"IF 语句： ");
    display(T->ptr[1],indent+5);
    break;
case IF_THEN_ELSE:
    printf("%*c%s\n",indent,' ',"条件语句（if-else-if）： ");
    display(T->ptr[0],indent+5);
    display(T->ptr[1],indent+5);
    break;
case WHILE:
    printf("%*c%s\n",indent,' ',"循环语句（while）： ");
    printf("%*c%s\n",indent+5,' ',"循环条件： ");
    display(T->ptr[0],indent+5);
    printf("%*c%s\n",indent+5,' ',"循环体： ");
```

```
            display(T->ptr[1],indent+5);
            break;
        case FOR:
            printf("%*c%s\n",indent,' ',"循环语句（for）：");
            printf("%*c%s\n",indent+5,' ',"循环条件：");
            display(T->ptr[0],indent+5);
            printf("%*c%s\n",indent+5,' ',"循环体：");
            display(T->ptr[1],indent+5);
            break;
        case FUNC_CALL:
            printf("%*c%s\n",indent,' ',"函数调用：");
            printf("%*c%s%s\n",indent+5,' ',"函数名：",T->type_id);
            printf("%*c%s\n",indent+5,' ',"第一个实际参数表达式：");
            display(T->ptr[0],indent+5);
            break;
        case ARGS:
            display(T->ptr[0],indent+5);
            display(T->ptr[1],indent+5);
            break;
        case ID:
            printf("%*cID： %s\n",indent,' ',T->type_id);//控制新的一行输出的空格
数，indent 代替%*c 中*
            break;
        case INT:
            printf("%*cINT： %d\n",indent,' ',T->type_int);
            break;
        case FLOAT:
            printf("%*cFLOAT： %f\n",indent,' ',T->type_float);
            break;
        case CHAR:
            printf("%*cCHAR： %c\n",indent,' ',T->type_char);
        case ARRAY:
            printf("%*c 数组名称： %s\n",indent,' ',T->type_id);
            break;
        case TYPE:
            if(T->type==INT)
                printf("%*c%s\n",indent,' ',"类型：int");
            else if(T->type==FLOAT)
                printf("%*c%s\n",indent,' ',"类型：float");
            else if(T->type==CHAR)
                printf("%*c%s\n",indent,' ',"类型：char");
            else if(T->type==ARRAY)
                printf("%*c%s\n",indent,' ',"类型：char 型数组");
            break;
```

```
            case ASSIGNOP:
            case OR:
            case AUTOADD_L:
            case AUTOSUB_L:
            case AUTOADD_R:
            case AUTOSUB_R:
            case AND:
            case RELOP:
            case PLUS:
            case MINUS:
            case STAR:
            case DIV:
            case COMADD:
            case COMSUB:
               printf("%*c%s\n",indent,' ',T->type_id);
               display(T->ptr[0],indent+5);
               display(T->ptr[1],indent+5);
               break;
            case RETURN:
               printf("%*c%s\n",indent,' ',"返回语句：");
               display(T->ptr[0],indent+5);
               break;
         }
      }
   }
```

3.3 符号表管理

在编译过程中，编译器使用符号表来记录源程序中各种名字的特性信息。在语法分析时构建语法分析树的同时构建符号表，符号表的建立主要是方便进行类型检查等分析，符号表记录的是源程序的符号信息。所谓“名字”包括：程序名、过程名、函数名、用户定义类型名、变量名、常量名、枚举值名、标号名等，所谓“特性信息”包括： 上述名字的种类、 具体类型、维数（如果语言支持数组）、函数参数个数、常量数值及目标地址（存储单元偏移地址）等。

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

3.4 语义检查

在语义分析中，我构造了函数 semantic_Analysis 来进行语义分析，其代码描述如下：

```
int i,j,counter=0,t;

int Semantic_Analysis(struct node* T,int type,int level,char flag,int command)
{
```

```
    int type1,type2;
    if(T){
      switch(T->kind){
        case EXT_DEF_LIST:
          Semantic_Analysis(T->ptr[0],type,level,flag,command);
          Semantic_Analysis(T->ptr[1],type,level,flag,command);
          break;
        case EXT_VAR_DEF://外部变量声明
          type=Semantic_Analysis(T->ptr[0],type,level,flag,command);
          Semantic_Analysis(T->ptr[1],type,level,flag,command);
          break;
        case ARRAY_DEF:
          type = Semantic_Analysis(T->ptr[0],type,level,flag,command);
          Semantic_Analysis(T->ptr[1],type,level,flag,command);
          break;
        case ARRAY_DEC:
          flag = 'A';//Array
          strcpy(new_table.symbols[new_table.index].name,T->type_id);
          new_table.symbols[new_table.index].level=level;
          new_table.symbols[new_table.index].type=type;
          new_table.symbols[new_table.index].flag=flag;
          new_table.index++;
          break;
        case TYPE:
          return T->type;
        case EXT_DEC_LIST:
          flag='V';
          Semantic_Analysis(T->ptr[0],type,level,flag,command);
          Semantic_Analysis(T->ptr[1],type,level,flag,command);
          break;
        case ID://检测新的变量名是否唯一
          i=0;
          while(new_table.symbols[i].level!=level&&i<new_table.index)// 转 到 相
同作用域
            i++;
          if(command==0){//定义变量
            while(i<new_table.index){
              if(strcmp(new_table.symbols[i].name,T->type_id)==0 && new_tabl
e.symbols[i].flag==flag){
                if(flag=='V')
                  printf("ERROR！ 第%d 行： 全 局 变 量 中 出 现 相 同 变 量
名%s\n",T->pos,T->type_id);
                else if(flag=='F')
                  printf("ERROR！ 第%d 行：函数定义中出现了相同的函数
```

```
名%s\n",T->pos,T->type_id);
                    else if(flag=='T')
                        printf("ERROR！第%d 行：局部变量中出现了相同的变量
名%s\n",T->pos,T->type_id);
                    else
                        printf("ERROR！第%d 行：函数参数中中出现了相同的变量
名%s\n",T->pos,T->type_id);
                    return 0;
                }
                i++;
            }
            strcpy(new_table.symbols[new_table.index].name,T->type_id);
            new_table.symbols[new_table.index].level=level;
            new_table.symbols[new_table.index].type=type;
            new_table.symbols[new_table.index].flag=flag;
            new_table.index++;
        }
        else{//使用变量
            i=new_table.index-1;
            while(i>=0){
                if(strcmp(new_table.symbols[i].name,T->type_id)==0&&(new_table
.symbols[i].flag=='V'||new_table.symbols[i].flag=='T')){
                    return new_table.symbols[i].type;
                }
                i--;
            }
            if(i<0){
                printf("ERROR!第%d 行:变量名%s 未定义\n",T->pos,T->type_id);
            }
        }
        break;
    case FUNC_DEF://函数声明
        type=Semantic_Analysis(T->ptr[0],type,level+1,flag,command);
        Semantic_Analysis(T->ptr[1],type,1,flag,command);
        Semantic_Analysis(T->ptr[2],type,1,flag,command);
        break;
    case FUNC_DEC:
        strcpy(new_table.symbols[new_table.index].name,T->type_id);
        new_table.symbols[new_table.index].level=0;
        new_table.symbols[new_table.index].type=type;
        new_table.symbols[new_table.index].flag='F';
        new_table.index++;
        counter=0;
        Semantic_Analysis(T->ptr[0],type,level,flag,command);//函数形参
```

```
            new_table.symbols[new_table.index - counter - 1].paramnum=counter;
            break;
        case PARAM_LIST:
            counter++;
            Semantic_Analysis(T->ptr[0],type,level,flag,command);
            Semantic_Analysis(T->ptr[1],type,level,flag,command);
            break;
        case PARAM_DEC:
            flag='P';
            type=Semantic_Analysis(T->ptr[0],type,level+1,flag,command);
            Semantic_Analysis(T->ptr[1],type,level,flag,command);
            break;
        case COMP_STM:
            flag='T';
            command=0;
            new_scope.TX[new_scope.top]=new_table.index;
            new_scope.top++;
            Semantic_Analysis(T->ptr[0],type,level,flag,command);//分析定义列表
            command=1;
            Semantic_Analysis(T->ptr[1],type,level+1,flag,command);//分析语句列
表
            new_table.index=new_scope.TX[new_scope.top-1];
            new_scope.top--;
            if (new_scope.top == 0)
              DisplaySymbolTable();
            break;
        case DEF_LIST:
            Semantic_Analysis(T->ptr[0],type,level,flag,command);
            Semantic_Analysis(T->ptr[1],type,level,flag,command);
            break;
        case VAR_DEF:
            type=Semantic_Analysis(T->ptr[0],type,level+1,flag,command);
            Semantic_Analysis(T->ptr[1],type,level,flag,command);
            break;
        case DEC_LIST:
            Semantic_Analysis(T->ptr[0],type,level,flag,command);
            Semantic_Analysis(T->ptr[1],type,level,flag,command);
            break;
        case STM_LIST:
            Semantic_Analysis(T->ptr[0],type,level,flag,command);//第一个语句
            Semantic_Analysis(T->ptr[1],type,level,flag,command);//其他语句
            break;
        case EXP_STMT:
            Semantic_Analysis(T->ptr[0],type,level,flag,command);
```

```
    break;
case RETURN:
    Semantic_Analysis(T->ptr[0],type,level,flag,command);
    break;
case IF_THEN:
case WHILE:
case FOR:
    Semantic_Analysis(T->ptr[0],type,level,flag,command);
    Semantic_Analysis(T->ptr[1],type,level,flag,command);
    break;
case IF_THEN_ELSE:
    Semantic_Analysis(T->ptr[0],type,level,flag,command);
    Semantic_Analysis(T->ptr[1],type,level,flag,command);
    Semantic_Analysis(T->ptr[2],type,level,flag,command);
    break;
case ASSIGNOP:
case OR:
case AND:
case RELOP:
case PLUS:
case MINUS:
case STAR:
case DIV:
case COMADD:
case COMSUB:
    type1=Semantic_Analysis(T->ptr[0],type,level,flag,command);
    type2=Semantic_Analysis(T->ptr[1],type,level,flag,command);
    if(type1==type2)
        return type1;
    break;
case AUTOADD_L:
case AUTOSUB_L:
case AUTOADD_R:
case AUTOSUB_R:
    Semantic_Analysis(T->ptr[0],type,level,flag,command);
    break;
case INT:
    return INT;
case FLOAT:
    return FLOAT;
case CHAR:
    return CHAR;
case FUNC_CALL:
    j=0;
```

```
            while(new_table.symbols[j].level==0&&j<new_table.index){
              if(strcmp(new_table.symbols[j].name,T->type_id)==0){
                if(new_table.symbols[j].flag!='F')
                  printf("ERROR！第%d 行：函数名%s 在符号表中定义为变量
\n",T->pos,T->type_id);
                break;
              }
              j++;
            }
            if(new_table.symbols[j].level==1||j==new_table.index){
              printf("ERROR！第%d 行：函数%s 未定义\n",T->pos,T->type_id);
              break;
            }
            type=new_table.symbols[j+1].type;
            counter=0;
            Semantic_Analysis(T->ptr[0],type,level,flag,command);//分析参数
            if(new_table.symbols[j].paramnum!=counter)
              printf("ERROR！第%d 行：函数调用%s 参数个数不匹配
\n",T->pos,T->type_id);
            break;
          case ARGS:
            counter++;
            t=Semantic_Analysis(T->ptr[0],type,level,flag,command);
            if(type!=t)
              printf("ERROR！第%d 行：函数调用的第%d 个参数类型不匹配
\n",T->pos,counter);
            type=new_table.symbols[j+counter+1].type;
            Semantic_Analysis(T->ptr[1],type,level,flag,command);
            break;
        }
    }
    return 0;
  }
```

其中，DisplaySymbolTable 函数是输出符号表的函数，实现如下：

```
  void DisplaySymbolTable()
  {
    int i;
    printf("\t\t***Symbol Table***\n");
    printf("-------------------------------------------------\n");
    printf("%s\t%s\t%s\t%s\t%s\t%s\n","Index","Name","Level","Type","Flag","Para
m_num");
    printf("-------------------------------------------------\n");
    for(i=0;i<new_table.index;i++){
      printf("%d\t",i);
```

```
        printf("%s\t",new_table.symbols[i].name);
        printf("%d\t",new_table.symbols[i].level);
        if(new_table.symbols[i].type==INT)
            printf("%s\t","int");
        else if(new_table.symbols[i].type==FLOAT)
            printf("%s\t","float");
        else
            printf("%s\t","char");
        printf("%c\t",new_table.symbols[i].flag);
        if(new_table.symbols[i].flag=='F')
            printf("%d\n",new_table.symbols[i].paramnum);
        else
            printf("\n");
    }
    printf("------------------------------------------------\n");
    printf("\n");
}
```

3.5 报错功能

见上

3.6 中间代码生成

1．定义 opn 结构体：

```
struct opn{
        int kind;
        int type;
        union {
                int const_int;
                float     const_float;
                char       const_char;
                char       id[33];
        };
        int level;
        int offset;
};
```

其包含类型、种类、层号、偏移量等信息。

2．定义 codenode 结构体：

```
struct codenode {
        int op;
struct opn opn1,opn2,result;
struct codenode *next,*prior;
};
```

其采用双向循环链表的方式存储中间代码。

3．定义 node 结构体：

```
struct node {
    enum node_kind kind;
    union
    {
        char type_id[33]; char type_char;int type_int; float type_float;
    };
    struct node *ptr[3];
    int level;int place;
    char Etrue[15],Efalse[15]; char Snext[15];
    struct codenode *code; char op[10];
    int type;
    int pos;
    int offset;
    int width;
};
```

其包含诸多含义：

place 记录该结点操作数在符号表中的位置序号；

type 记录该数据的类型，用于表达式计算；

offset 记录外部变量在静态数据区中的偏移量以及局部变量和临时变量 在活动记录中的偏移量；

width 记录个结点表示的语法单位中，定义的变量和临时单元所需要占 用的字节数；

code 记录中间代码序列的起始位置；

Etrue、Efalse 记录在完成布尔表达式翻译时，表达式值为‘真’（或为‘假’）时，要转移的程序位置；

Snext 记录该结点的语句序列执行完后，要转移到的程序位置。

4．newAlias 函数

代码描述：

```
char *newAlias()
{ //生成新别名
    static int k = 1;
    static char result[10];
    char s[10];
    snprintf(s, 10, "%d", k++);
    strcpy(result, "v");
    strcat(result, s);
    return result;
}
```

函数作用：生成一个新的别名。

5．newLabel 函数

代码描述：

```
char *newLabel()
{ //生成新标号
```

```
    static int k = 1;
    static char result[10];
    char s[10];
    snprintf(s, 10, "%d", k++);
    strcpy(result, "label");
    strcat(result, s);
    return result;
}
```
函数作用：生成一个新的标号。

6．newTemp 函数

代码描述：

```
char *newTemp()
{ //生成新的临时变量
    static int k = 1;
    static char result[10];
    char s[10];
    snprintf(s, 10, "%d", k++);
    strcpy(result, "temp");
    strcat(result, s);
    return result;
}
```
函数作用：生成一个新的临时变量。

7．makeopn 函数

代码描述：

```
struct opn makeopn(struct node *T,int kind)
{
    struct opn op;
    int judge=check_var(T);
    op.kind=kind;
    op.type=symbolTable.symbols[judge].type;
    if(kind==FUNCTION)
        strcpy(op.id,symbolTable.symbols[judge].name);
    else
        strcpy(op.id,symbolTable.symbols[judge].alias);
    return op;
}
```
函数作用：建立 opn 结构体用来存储各类信息。

8．genIR 函数

代码描述：

```
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;
}
```
函数作用：生成一条 TAC 代码的结点组成的双向循环链表，返回头指针

9. genLable 函数：

代码描述：

```
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;
}
```
函数作用：生成一条标号语句，返回头指针

10. genGoto 函数：

代码描述：

```
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;
}
```
函数作用：生成 GOTO 语句，返回头指针

11. merge 函数

代码描述：

```
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;
    }
```

函数作用：合并双向循环链表

12．prnIR 函数

代码描述：

```
    void prnIR(struct codenode *head)
    {
        char opnstr1[32], opnstr2[32], resultstr[32];
        struct codenode *h = head;
        do
        {
            if (h->opn1.kind == INT)
                sprintf(opnstr1, "#%d", h->opn1.const_int);
            if (h->opn1.kind == FLOAT)
                sprintf(opnstr1, "#%f", h->opn1.const_float);
            if (h->opn1.kind == ID)
                sprintf(opnstr1, "%s", h->opn1.id);
            if (h->opn2.kind == INT)
                sprintf(opnstr2, "#%d", h->opn2.const_int);
            if (h->opn2.kind == FLOAT)
                sprintf(opnstr2, "#%f", h->opn2.const_float);
            if (h->opn2.kind == ID)
                sprintf(opnstr2, "%s", h->opn2.id);
            sprintf(resultstr, "%s", h->result.id);
            switch (h->op)
            {
            case ASSIGNOP:
                printf("    %s := %s\n", resultstr, opnstr1);
                break;
            case PLUS:
            case MINUS:
            case STAR:
            case DIV:
                printf("    %s := %s %c %s\n", resultstr, opnstr1,
                        h->op == PLUS ? '+' : h->op == MINUS ? '-' : h->op ==
STAR ? '*' : '\\', opnstr2);
```

```
        break;
    case FUNCTION:
        printf("\nFUNCTION %s :\n", h->result.id);
        break;
    case PARAM:
        printf("    PARAM %s\n", h->result.id);
        break;
    case LABEL:
        printf("LABEL %s :\n", h->result.id);
        break;
    case GOTO:
        printf("    GOTO %s\n", h->result.id);
        break;
    case JLE:
        printf("    IF %s <= %s GOTO %s\n", opnstr1, opnstr2, resultstr);
        break;
    case JLT:
        printf("    IF %s < %s GOTO %s\n", opnstr1, opnstr2, resultstr);
        break;
    case JGE:
        printf("    IF %s >= %s GOTO %s\n", opnstr1, opnstr2, resultstr);
        break;
    case JGT:
        printf("    IF %s > %s GOTO %s\n", opnstr1, opnstr2, resultstr);
        break;
    case EQ:
        printf("    IF %s == %s GOTO %s\n", opnstr1, opnstr2, resultstr);
        break;
    case NEQ:
        printf("    IF %s != %s GOTO %s\n", opnstr1, opnstr2, resultstr);
        break;
    case ARG:
        printf("    ARG %s\n", h->result.id);
        break;
    case CALL:
        printf("    %s := CALL %s\n", resultstr, opnstr1);
        break;
    case RETURN:
        if (h->result.kind)
            printf("    RETURN %s\n", resultstr);
        else
            printf("    RETURN\n");
        break;
    }
```

```
            h = h->next;
        } while (h != head);
    }
```

函数作用：输出中间代码

13. **处理函数**

中间代码生成与语义分析是混合的，在遍历语法树时，遇到特定的符号，分别执行不同的操作。将这些不同的操作封装成为函数，定义的函数如下，具体细节因为篇幅有限，不在此具体展示。

```
void id_exp(struct node *T);
void int_exp(struct node *T);
void assignop_exp(struct node *T);
void relop_exp(struct node *T);
void args_exp(struct node *T);
void op_exp(struct node *T);
void func_call_exp(struct node *T);
void not_exp(struct node *T);
void ext_var_list(struct node *T);
void ext_def_list(struct node *T);
void ext_var_def(struct node *T);
void func_def(struct node *T);
void func_dec(struct node *T);
void param_list(struct node *T);
void param_dec(struct node *T);
void comp_stm(struct node *T);
void def_list(struct node *T);
void var_def(struct node *T);
void stmt_list(struct node *T);
void if_then(struct node *T);
void if_then_else(struct node *T);
void while_dec(struct node *T);
void exp_stmt(struct node *T);
void return_dec(struct node *T);
```

3.7 代码优化

未做

3.8 汇编代码生成

目标代码生成算法如表 5 所示。

表 5 目标代码生成算法

中间代码	MIPS32 指令
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 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 div $t1,$t2 mflo $t3 sw $t3, x 的偏移量($sp)
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
IF x>=y GOTO z	lw $t1, x 的偏移量($sp) lw $t2, y 的偏移量($sp) bge $t1,$t2,z
IF x<y GOTO z	lw $t1, x 的偏移量($sp) lw $t2, y 的偏移量($sp) ble $t1,$t2,z
IF x<=y GOTO z	lw $t1, x 的偏移量($sp) lw $t2, y 的偏移量($sp)

	blt $t1,$t2,z
X:=CALL f	

上表是中间代码对应的 MIPS 指令，此时我运用了一个核心函数 translate 对中间代码进行翻译。核心思想是将每一行中间代码读入，分别进行翻译处理。

```
string translate(string temp_str){
    //将每行 string 按空格存成数组
    vector<string> line;
    string temp_res;
    stringstream input(temp_str);
    while (input>>temp_res)
        line.emplace_back(temp_res);
    //核心处理
    string temp_return;
    if(line[0] == "LABEL")
        return line[1]+":";
    if (line[1] == ":=") {
        if (line.size() == 3)
            if (temp_str[temp_str.length()-2] == '#')
                return "\tli " + Get_R(line[0]) + ","+line.back().back();
            else{
                temp_return = "\tmove ";
                temp_return += Get_R(line[0])+',';
                temp_return += Get_R(line[2]);
                return temp_return;
            }
        if (line.size() == 5){
            if (line[3] == "+")
                if (temp_str[temp_str.length()-2] == '#'){
                    temp_return = "\taddi ";
                    temp_return += Get_R(line[0])+",";
                    temp_return += Get_R(line[2])+",";
                    temp_return += line.back().back();
                    return temp_return;
                }
                else{
                    temp_return = "\tadd ";
                    temp_return += Get_R(line[0])+",";
                    temp_return += Get_R(line[2])+",";
                    temp_return += Get_R(line.back());
                    return temp_return;
                }
            else if (line[3] == "-"){
                if (temp_str[temp_str.length()-2] == '#'){
                    temp_return = "\taddi ";
```

```
            temp_return += Get_R(line[0])+",";
            temp_return += Get_R(line[2])+",";
            temp_return += line.back().back();
            return temp_return;
        }
        else{
            temp_return = "\tsub ";
            temp_return += Get_R(line[0])+",";
            temp_return += Get_R(line[2])+",";
            temp_return += Get_R(line.back());
            return temp_return;
        }
    }
    else if (line[3] == "*"){
        temp_return = "\tmul ";
        temp_return += Get_R(line[0])+",";
        temp_return += Get_R(line[2])+",";
        temp_return += Get_R(line.back());
        return temp_return;
    }
    else if (line[3] == "/"){
        temp_return = "\tdiv ";
        temp_return += Get_R(line[2])+",";
        temp_return += Get_R(line.back()) + "\n\tmflo ";
        temp_return += Get_R(line[0]);
        return temp_return;
    }
    else if (line[3] == "<"){
        temp_return = "\tslt ";
        temp_return += Get_R(line[0])+",";
        temp_return += Get_R(line[2])+",";
        temp_return += Get_R(line.back());
        return temp_return;
    }
    else if (line[3] == ">"){
        temp_return = "\tslt ";
        temp_return += Get_R(line[0])+",";
        temp_return += Get_R(line.back())+",";
        temp_return += Get_R(line[2]);
        return temp_return;
    }
}

if (line[2] == "CALL")
```

```
            if (line[3] == "read" || line[3] == "print")
               return "\taddi $sp,$sp,-
4\n\tsw $ra,0($sp)\n\tjal " + line.back() + "\n\tlw $ra,0($sp)\n\tmove " +Get_R(line[0]
) + ",$v0\n\taddi $sp,$sp,4";
            else
               return "\taddi $sp,$sp,-
24\n\tsw $t0,0($sp)\n\tsw $ra,4($sp)\n\tsw $t1,8($sp)\n\tsw $t2,12($sp)\n\tsw $t3,16(
$sp)\n\tsw $t4,20($sp)\n\tjal "+line.back()+"\n\tlw $a0,0($sp)\n\tlw $ra,4($sp)\n\tlw $
t1,8($sp)\n\tlw $t2,12($sp)\n\tlw $t3,16($sp)\n\tlw $t4,20($sp)\n\taddi $sp,$sp,24\n\t
move "+Get_R(line[0])+" $v0";
         }
         if (line[0] == "GOTO")
            return "\tj "+line[1];
         if (line[0] == "RETURN")
            return "\tmove $v0,"+Get_R(line[1])+"\n\tjr $ra";
         if (line[0] == "IF") {
            if (line[2] == "=="){
               temp_return = "\tbeq ";
               temp_return += Get_R(line[1])+",";
               temp_return += Get_R(line[3])+",";
               temp_return += line.back();
               return temp_return;
            }
            if (line[2] == "!="){
               temp_return = "\tbne ";
               temp_return += Get_R(line[1])+",";
               temp_return += Get_R(line[3])+",";
               temp_return += line.back();
               return temp_return;
            }
            if (line[2] == ">"){
               temp_return = "\tbgt ";
               temp_return += Get_R(line[1])+",";
               temp_return += Get_R(line[3])+",";
               temp_return += line.back();
               return temp_return;
            }
            if (line[2] == "<"){
               temp_return = "\tblt ";
               temp_return += Get_R(line[1])+",";
               temp_return += Get_R(line[3])+",";
               temp_return += line.back();
               return temp_return;
            }
```

```
            if (line[2] == ">="){
               temp_return = "\tbge ";
               temp_return += Get_R(line[1])+",";
               temp_return += Get_R(line[3])+",";
               temp_return += line.back();
               return temp_return;
            }
            if (line[2] == "<="){
               temp_return = "\tble ";
               temp_return += Get_R(line[1])+",";
               temp_return += Get_R(line[3])+",";
               temp_return += line.back();
               return temp_return;
            }
         }
         if (line[0] == "FUNCTION")
            return line[1]+":";
         if (line[0] == "CALL")
            if (line.back() == "read" || line.back() == "print")
               return "\taddi $sp,$sp,-
4\n\tsw $ra,0($sp)\n\tjal "+line.back()+"\n\tlw $ra,0($sp)\n\taddi $sp,$sp,4";
            else
               return "\taddi $sp,$sp,-
24\n\tsw $t0,0($sp)\n\tsw $ra,4($sp)\n\tsw $t1,8($sp)\n\tsw $t2,12($sp)\n\tsw $t3,16(
$sp)\n\tsw $t4,20($sp)\n\tjal "+line.back()+"\n\tlw $a0,0($sp)\n\tlw $ra,4($sp)\n\tlw $
t1,8($sp)\n\tlw $t2,12($sp)\n\tlw $t3,16($sp)\n\tlw $t4,20($sp)\n\taddi $sp,$sp,24\n\t
move "+Get_R(line[0])+" $v0";
         if (line[0] == "ARG")
            return "\tmove $t0,$a0\n\tmove $a0,"+Get_R(line.back());
         if (line[0] == "PARAM")
            table[line.back()] = "a0";
         return " ";
      }
```

4 系统测试与评价

5.1 测试用例

1.

```
int a,b,c;
float m,n;
int fibo(int a)
{
    if (a == 1 || a == 2) return 1;
    return fibo(a-1)+fibo(a-2);
}

int main()
{
    int m,n,i;
    m = read();
    i=1;
    while(i<=m)
    {
        n = fibo(i);
        write(n);
        i=i+1;
    }
    return 1;
}
```

图 6. 测试用例 1

2.

```c
int a,b,c;
float m,n;
char c1,c2;
char h[10];
float a,b;//全局变量中出现相同变量名

int fibo(int a)
{
        int i;
        int haha;
        if(a == 1 || a == 2){
          return 1;
        }
        for(i<15){
          i++;
        }

        j = i+1;//无定义错误
        haha(c);//未定义的函数
        a = fibo(1,2);//参数个数不匹配
        b = fibo(m);//参数类型不匹配

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

int h1(int a, int a){}//出现了相同函数参数

int h2(){int hah; float hah;}//局部变量名出现了相同的变量名

float h1(){}//重复的函数名
```

图 7. 测试用例 2

5.2 正确性测试

实验一测试结果如图：

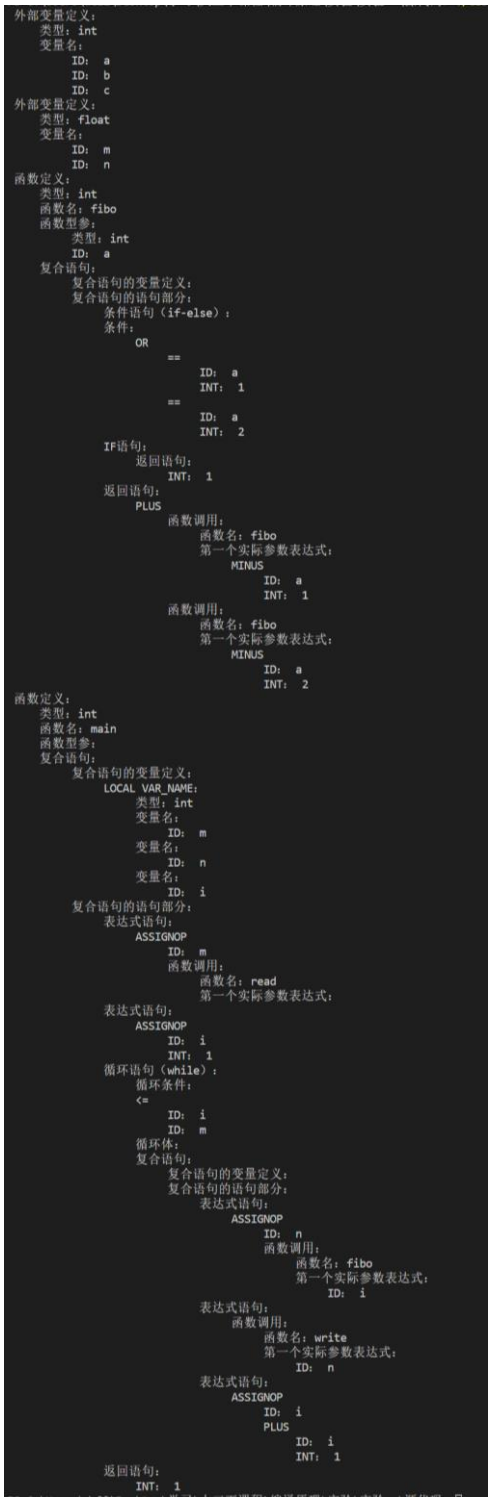

图 7.测试结果 1

实验二测试结果如图：

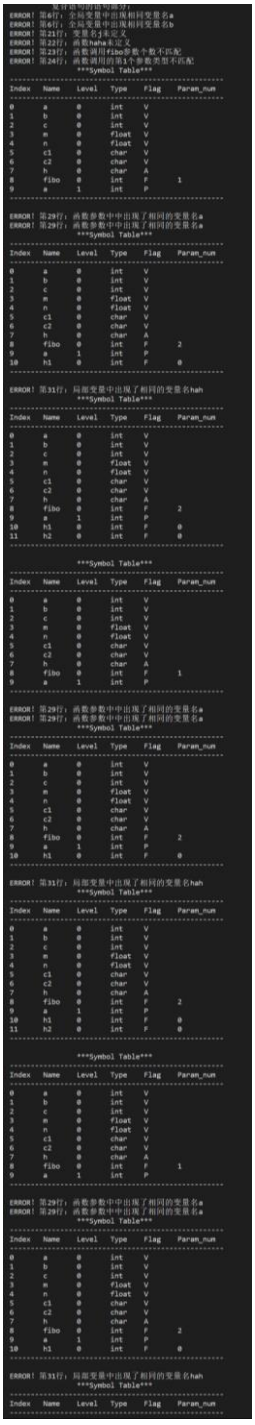

图 8.测试结果 2

实验三生成中间代码如图：

```
FUNCTION fibo :
  PARAM var7
  temp1 := #1
  IF var7 == temp1 GOTO label3
  GOTO label4
LABEL label4 :
  temp2 := #2
  IF var7 == temp2 GOTO label3
  GOTO label2
LABEL label3 :
  temp3 := #1
  RETURN temp3
LABEL label2 :
  temp4 := #1
  temp5 := var7 - temp4
  ARG temp5
  temp6 := CALL fibo
  temp7 := #2
  temp8 := var7 - temp7
  ARG temp8
  temp9 := CALL fibo
  temp10 := temp6 + temp9
  RETURN temp10
LABEL label1 :

FUNCTION main :
  var9 := var1
  temp11 := #1
  var11 := temp11
LABEL label10 :
  IF var11 <= var9 GOTO label9
  GOTO label8
LABEL label9 :
  ARG var11
  temp12 := CALL fibo
  var10 := temp12
  temp13 := #1
  temp14 := var11 + temp13
  GOTO label10
LABEL label8 :
  temp15 := #1
  RETURN temp15
LABEL label5 :
```

图 9.测试结果 3

实验四生成目标代码如下：

```
.data
_Prompt: .asciiz "Enter an integer:  "
_ret: .asciiz "\n"
.globl main
.text
```

```
read:
    li $v0,4
    la $a0,_Prompt
    syscall
    li $v0,5
    syscall
    jr $ra
write:
    li $v0,1
    syscall
    li $v0,4
    la $a0,_ret
    syscall
    move $v0,$0
    jr $ra

fibo:
    li $t3, 1
    sw $t3, 16($sp)
    lw $t1, 12($sp)
    lw $t2, 16($sp)
    beq $t1,$t2,label3
    j label4
label4:
    li $t3, 2
    sw $t3, 16($sp)
    lw $t1, 12($sp)
    lw $t2, 16($sp)
    beq $t1,$t2,label3
    j label2
label3:
    li $t3, 1
    sw $t3, 16($sp)
    lw $v0,16($sp)
    jr $ra
```

```
label2:
    li $t3, 1
    sw $t3, 16($sp)
    lw $t1, 12($sp)
    lw $t2, 16($sp)
    sub $t3,$t1,$t2
    sw $t3, 20($sp)
    move $t0,$sp
    addi $sp, $sp, -44
    sw $ra,0($sp)
    lw $t1, 20($t0)
    move $t3,$t1
    sw $t3,12($sp)
    jal fibo
    lw $ra,0($sp)
    addi $sp,$sp,44
    sw $v0,24($sp)
    li $t3, 2
    sw $t3, 28($sp)
    lw $t1, 12($sp)
    lw $t2, 28($sp)
    sub $t3,$t1,$t2
    sw $t3, 32($sp)
    move $t0,$sp
    addi $sp, $sp, -44
    sw $ra,0($sp)
    lw $t1, 32($t0)
    move $t3,$t1
    sw $t3,12($sp)
    jal fibo
    lw $ra,0($sp)
    addi $sp,$sp,44
    sw $v0,36($sp)
    lw $t1, 24($sp)
    lw $t2, 36($sp)
```

```
    add $t3,$t1,$t2
    sw $t3, 40($sp)
    lw $v0,40($sp)
    jr $ra
label1:

main:
    addi $sp, $sp, -32
    addi $sp, $sp, -4
    sw $ra,0($sp)
    jal read
    lw $ra,0($sp)
    addi $sp, $sp, 4
    sw $v0, 24($sp)
    lw $t1, 24($sp)
    move $t3, $t1
    sw $t3, 12($sp)
    li $t3, 1
    sw $t3, 24($sp)
    lw $t1, 24($sp)
    move $t3, $t1
    sw $t3, 20($sp)
label10:
    lw $t1, 20($sp)
    lw $t2, 12($sp)
    ble $t1,$t2,label9
    j label8
label9:
    move $t0,$sp
    addi $sp, $sp, -44
    sw $ra,0($sp)
    lw $t1, 20($t0)
    move $t3,$t1
    sw $t3,12($sp)
    jal fibo
```

```
    lw $ra,0($sp)
    addi $sp,$sp,44
    sw $v0,24($sp)
    lw $t1, 24($sp)
    move $t3, $t1
    sw $t3, 16($sp)
    lw $a0, 16($sp)
    addi $sp, $sp, -4
    sw $ra,0($sp)
    jal write
    lw $ra,0($sp)
    addi $sp, $sp, 4
    li $t3, 1
    sw $t3, 24($sp)
    lw $t1, 20($sp)
    lw $t2, 24($sp)
    add $t3,$t1,$t2
    sw $t3, 28($sp)
    lw $t1, 28($sp)
    move $t3, $t1
    sw $t3, 20($sp)
    j label10
label8:
    li $t3, 1
    sw $t3, 24($sp)
    lw $v0,24($sp)
    jr $ra
label5:
```

实验四运行结果如图：

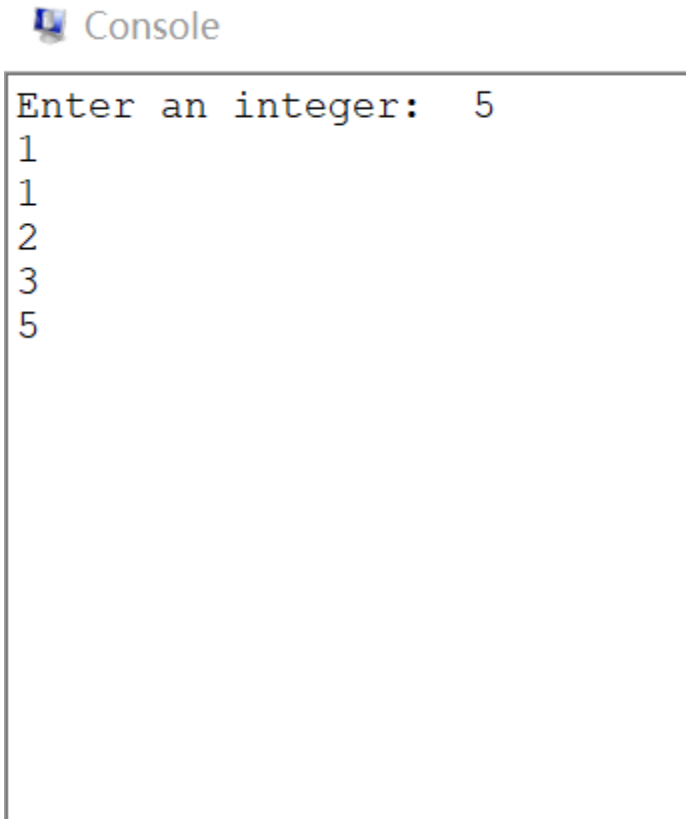

图 10.测试结果 4

5.3 报错功能测试

见上一节实验二测试结果图

5.4 系统的优点

功能正确，能正确编译运行 mini-c 程序

5.5 系统的缺点

没有任何新意，没做代码优化

5 实验小结或体会

四次编译原理实验任务最终完成了一个小型的编译器，虽然还有很多功能不支持，但是在这过程中对于词法分析、语法分析、语义分析、中间代码生成和目标代码生成的原理有了更加深入的理解。我完成的任务梳理如下：1.运用 lex 工具编写词法分析运用的正则表达式规则，实现程序的词法分析。2.运用 bison 编写语法分析部分，并将在其中加入了新的语法规则以及错误处理规则，实现了源代码的语法分析。3.设计 display 函数、mknode 函数在分析中建立抽象语法树 4.在语法分析和词法分析的基础上，设计 semantic_Analysis 函数，生成符号表，

利用符号表和语法树成功进行了语义分析，并能够辨别出不同类型的错误，最后输出符号表。5.在语义分析的时候就进行中间代码的生成，提高编译程序的效率6.利用生成的中间代码生成最终的 MIPS32 目标代码。在实验中，我有如下的心得体会：1.编译原理实验是一个具有挑战性的实验，整个过程不断在巩固课上没有重点讲的内容，让我在实践当中学习到了更多知识。同时，很锻炼我的编码能力，此次实验我学会了如何将复杂问题拆分为简单问题，从简单问题入手逐步解决困难问题。2.实验的核心部分是中间代码生成，但是实验指导书介绍的很有限，希望老师以后多给些资料阅读，其实看完指导书仅仅是算入门，对于真正上手写代码其实还有很长距离。第三次实验花费了很多精力，从从头开始翻阅理论书了解原理，到发现原理和实现的距离，再读各类编译器的代码，再不断思考之后才写出来。的确，前期的积累是很痛苦的，但是一旦理解开始，就非常快，几百行代码很快就出来了，因为逻辑是清晰的。

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

1.lex.l

```
%{
#include "parser.tab.h"
#include "string.h"
#include "def.h"
int yycolumn=1;
#define YY_USER_ACTION
yylloc.first_line=yylloc.last_line=yylineno; \
    yylloc.first_column=yycolumn;    yylloc.last_column=yycolumn+yyleng-1;
yycolumn+=yyleng;
typedef union {
    int type_int;
    float type_float;
    char type_char[3];
    char type_string[31];
    char type_id[32];
    struct node *ptr;
} YYLVAL;
#define YYSTYPE YYLVAL

%}
%option yylineno

id     [A-Za-z][A-Za-z0-9]*
int       [0-9]+
float    ([0-9]*\.[0-9]+)|([0-9]+\.)
char     ('[A-Za-z0-9]')
string (\"[A-Za-z0-9]*\")

%%
{int}            {yylval.type_int=atoi(yytext);return INT;}
{float}          {yylval.type_float=atof(yytext);return FLOAT;}
{char}             {strcpy(yylval.type_char,yytext);return CHAR;}
{string}
{strcpy(yylval.type_string,yytext);/*printf("(string,STRING)");*/return STRING;}

"int"            {strcpy(yylval.type_id,    yytext);/*printf("(TYPE,int)");*/return
TYPE;}
"float"          {strcpy(yylval.type_id,    yytext);/*printf("(TYPE,float)");*/return
```

```
TYPE;}
    "char"              {strcpy(yylval.type_id,
yytext);/*printf("(TYPE,char)");*/return TYPE;}
    "string"          {strcpy(yylval.type_id,
yytext);/*printf("(TYPE,string)");*/return TYPE;}
    "struct"         {/*printf("(STRUCT,struct)");*/return STRUCT;}


    "return"         {/*printf("(RETURN,return)");*/return RETURN;}
    "if"              {/*printf("(IF,if)");*/return IF;}
    "else"            {/*printf("(ELSE,else)");*/return ELSE;}
    "while"           {/*printf("(WHILE),while");*/return WHILE;}
    "for"             {/*printf("(FOR,for)");*/return FOR;}

    {id}            {strcpy(yylval.type_id,
yytext);/*printf("(ID,%s)",yylval.type_id);*/return ID;/*由于关键字的形式也符合表
示符的规则，所以把关键字的处理全部放在标识符的前面，优先识别*/}
    ";"             {/*printf("(SEMI,;)");*/return SEMI;}
    ","             {/*printf("(COMMA,,)");*/return COMMA;}
    ">"|"<"|">="|"<="|"=="|"!=" {strcpy(yylval.type_id,
yytext);/*printf("(RELOP,%s)",yylval.type_id);*/return RELOP;}
    "="             {/*printf("(ASSIGNOP,=)");*/return ASSIGNOP;}
    "+"             {/*printf("(PLUS,+)");*/return PLUS;}
    "-"             {/*printf("(MINUS,-)");*/return MINUS;}
    "*"             {/*printf("(STAR,*)");*/return STAR;}
    "/"             {/*printf("(DIV,/)");*/return DIV;}
    "&&"              {/*printf("(AND,&&)");*/return AND;}
    "||"            {/*printf("(OR,||)");*/return OR;}
    "."              {/*printf("(DOT,.)");*/return DOT;}
    "!"             {/*printf("(NOT,!)");*/return NOT;}
    "("             {/*printf("(LP,()");*/return LP;}
    ")"             {/*printf("(RP,))");*/return RP;}
    "["              {/*printf("(LB,[)");*/return LB;}
    "]"              {/*printf("(RB,])");*/return RB;}
    "{"             {/*printf("(LC,{)");*/return LC;}
    "}"             {/*printf("(RC,})\n");*/return RC;}
    [\n]             {yycolumn=1;}
    [ \r\t]        {}
    "//"[^\n]*    {/* 注释 */}
    "/*"([^\*]|(\*)*[^\*/])*(\*)*"*/" {/* 注释 */}
    .               {printf("\n==>ERROR:Mysterious character \"%s\" at
Line %d\n",yytext,yylineno);}
    %%
```

```
/* 和 bison 联用时，不需要这部分
void main()
{
yylex();
return 0;
}

*/
int yywrap()
{
return 1;
}
```

2.parser.y

```
%error-verbose
%locations
%{
#include "stdio.h"
#include "math.h"
#include "string.h"
#include "def.h"
extern int yylineno;
extern char *yytext;
extern FILE *yyin;
void yyerror(const char* fmt, ...);
void display(struct node *,int);
%}

%union {
    int       type_int;
    float    type_float;
    char type_char[3];
    char type_string[31];
    char      type_id[32];
    struct node *ptr;
};

//   %type 定义非终结符的语义值类型
%type    <ptr> program ExtDefList ExtDef Specifier StructSpecifier OptTag Tag
ExtDecList FuncDec CompSt VarList VarDec ParamDec Stmt ForDec StmList
DefList Def DecList Dec Exp Args

//% 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_float> FLOAT              //指定 ID 的语义值是 type_id，有词法分析得到的标识符字符串
%token <type_char> CHAR              //指定 ID 的语义值是 type_id，有词法分析得到的标识符字符串
%token <type_string> STRING            //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 STRUCT LP RP LB RB LC RC SEMI COMMA DOT   //用 bison 对该文件编译时，带参数-d，生成的 exp.tab.h 中给这些单词进行编码，可在 lex.l 中包含 parser.tab.h 使用这些单词种类码
%token PPLUS MMINUS PLUS MINUS STAR DIV ASSIGNOP MINUSASSIGNOP PLUSASSIGNOP DIVASSIGNOP STARASSIGNOP AND OR NOT IF ELSE WHILE FOR RETURN

%right ASSIGNOP MINUSASSIGNOP PLUSASSIGNOP DIVASSIGNOP STARASSIGNOP //  赋值
%left OR
%left AND
%left RELOP
%left PLUS MINUS
%left STAR DIV
%right UMINUS NOT PPLUS MMINUS //  负号和非
%right LB
%left RB
%left DOT
%nonassoc LOWER_THEN_ELSE
%nonassoc ELSE

%%

program: ExtDefList      {display($1,0);DisplaySymbolTable($1);}      /*显示语法树,语义分析  */;
                ;
ExtDefList: {$$=NULL;}
                | ExtDef ExtDefList {$$=mknode(EXT_DEF_LIST,$1,$2,NULL,yylineno);}    //每一个 EXTDEFLIST 的结点，其第 1 棵子树对应一个外部变量声明或函数
                ;
ExtDef:     Specifier ExtDecList SEMI {$$=mknode(EXT_VAR_DEF,$1,$2,NULL,yylineno);}    //该结点对应一个外部
```

```
变量声明
                | Specifier SEMI
                | Specifier FuncDec CompSt
{$$=mknode(FUNC_DEF,$1,$2,$3,yylineno);}            //该结点对应一个函数定义
                | error SEMI     {$$=NULL;}
                ;

    ExtDecList:   VarDec          {$$=$1;}          /*每一个 EXT_DECLIST 的结点，其第一棵子树对应一个变量名(ID 类型的结点),第二棵子树对应剩下的外部变量名*/
                  | VarDec COMMA ExtDecList
{$$=mknode(EXT_DEC_LIST,$1,$3,NULL,yylineno);}
                  ;
    /*
    Specifier:   TYPE
{$$=mknode(TYPE,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);$$->type=(!strcmp($1,"int"))?INT:(!strcmp($1,"float"))?FLOAT:(!strcmp($1,"char"))?CHAR:(!strcmp($1,"string"))?STRING:NULL;}
                  | StructSpecifier {}
                  ;
    */
    Specifier:   TYPE
{$$=mknode(TYPE,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);if(!strcmp($1, "int"))$$->type=INT;if(!strcmp($1, "float"))$$->type=FLOAT;if(!strcmp($1, "char"))$$->type=CHAR;if(!strcmp($1, "string"))$$->type=STRING;}
                  | StructSpecifier {$$=$1;}
                  ;

    StructSpecifier: STRUCT OptTag LC DefList RC
{$$=mknode(STRUCT_DEF,$2,$4,NULL,yylineno);}
                  | STRUCT Tag
{$$=mknode(STRUCT_DEC,$2,NULL,NULL,yylineno);}
                  ;

    OptTag: {$$=NULL;}
            | ID
{$$=mknode(STRUCT_TAG,NULL,NULL,NULL,yylineno);strcpy($$->struct_name,$1);}
            ;
    Tag: ID
{$$=mknode(STRUCT_TAG,NULL,NULL,NULL,yylineno);strcpy($$->struct_name,$1);}
        ;
```

```
    VarDec:    ID
{$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);}     //ID 结
点，标识符符号串存放结点的 type_id
                | VarDec LB INT RB {struct node
*temp=mknode(INT,NULL,NULL,NULL,yylineno);temp->type_int=$3;$$=mknode(
ARRAY_DEC, $1, temp,
NULL,yylineno);}//$$=mknode(ARRAY_DEC,$1,$3,NULL,yylineno);
                 ;

    FuncDec: ID LP VarList RP
{$$=mknode(FUNC_DEC,$3,NULL,NULL,yylineno);strcpy($$->type_id,$1);}//函
数名存放在$$->type_id
          |ID LP    RP
{$$=mknode(FUNC_DEC,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);}//
函数名存放在$$->type_id
                ;

    VarList: ParamDec    {$$=mknode(PARAM_LIST,$1,NULL,NULL,yylineno);}
                | ParamDec COMMA    VarList
{$$=mknode(PARAM_LIST,$1,$3,NULL,yylineno);}
                ;

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

    // 定义部分和执行部分
    CompSt: LC DefList StmList RC
{$$=mknode(COMP_STM,$2,$3,NULL,yylineno);}
              ;
    StmList: {$$=NULL; }
              | Stmt StmList    {$$=mknode(STM_LIST,$1,$2,NULL,yylineno);}
              ;
    Stmt:     Exp SEMI       {$$=mknode(EXP_STMT,$1,NULL,NULL,yylineno);}
            | CompSt         {$$=$1;}          //复合语句结点直接最为语句结点，
不再生成新的结点
            | RETURN Exp SEMI
{$$=mknode(RETURN,$2,NULL,NULL,yylineno);}
            | IF LP Exp RP Stmt %prec LOWER_THEN_ELSE
{$$=mknode(IF_THEN,$3,$5,NULL,yylineno);}
            | IF LP Exp RP Stmt ELSE Stmt
{$$=mknode(IF_THEN_ELSE,$3,$5,$7,yylineno);}
            | WHILE LP Exp RP Stmt
```

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

    ForDec: Exp SEMI Exp SEMI Exp
{$$=mknode(FOR_DEC,$1,$3,$5,yylineno);}
             | SEMI Exp SEMI
{$$=mknode(FOR_DEC,NULL,$2,NULL,yylineno);}
             ;

    DefList: {$$=NULL; }
             | Def DefList {$$=mknode(DEF_LIST,$1,$2,NULL,yylineno);}
             ;
    Def:     Specifier DecList SEMI
{$$=mknode(VAR_DEF,$1,$2,NULL,yylineno);}
             ;
    DecList: Dec   {$$=mknode(DEC_LIST,$1,NULL,NULL,yylineno);}
            | Dec COMMA DecList
{$$=mknode(DEC_LIST,$1,$3,NULL,yylineno);}
        ;
    Dec:       VarDec   {$$=$1;}
            | VarDec ASSIGNOP Exp
{$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type_id,"ASSIGNOP");
}
            ;
    Exp:     Exp ASSIGNOP Exp
{$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type_id,"ASSIGNOP");
}//$$结点 type_id 空置未用，正好存放运算符
            | Exp AND Exp
{$$=mknode(AND,$1,$3,NULL,yylineno);strcpy($$->type_id,"AND");}
            | Exp OR Exp
{$$=mknode(OR,$1,$3,NULL,yylineno);strcpy($$->type_id,"OR");}
            | Exp RELOP Exp
{$$=mknode(RELOP,$1,$3,NULL,yylineno);strcpy($$->type_id,$2);}  //词法分析
关系运算符号自身值保存在$2 中
            | Exp PLUS Exp
{$$=mknode(PLUS,$1,$3,NULL,yylineno);strcpy($$->type_id,"PLUS");}
            | Exp PLUS PLUS
{$$=mknode(PPLUS,$1,NULL,NULL,yylineno);strcpy($$->type_id,"PPLUS");}
            | Exp PLUS ASSIGNOP Exp
{$$=mknode(PLUSASSIGNOP,$1,$4,NULL,yylineno);strcpy($$->type_id,"PLUSA
SSIGNOP");}
            | Exp MINUS Exp
{$$=mknode(MINUS,$1,$3,NULL,yylineno);strcpy($$->type_id,"MINUS");}
```

```
        | Exp MINUS MINUS
{$$=mknode(MMINUS,$1,NULL,NULL,yylineno);strcpy($$->type_id,"MMINUS")
;}
        | Exp MINUS ASSIGNOP Exp
{$$=mknode(MINUSASSIGNOP,$1,$4,NULL,yylineno);strcpy($$->type_id,"MINU
SASSIGNOP");}
        | Exp STAR Exp
{$$=mknode(STAR,$1,$3,NULL,yylineno);strcpy($$->type_id,"STAR");}
        | Exp STAR ASSIGNOP Exp
{$$=mknode(STARASSIGNOP,$1,$4,NULL,yylineno);strcpy($$->type_id,"STARA
SSIGNOP");}
        | Exp DIV Exp
{$$=mknode(DIV,$1,$3,NULL,yylineno);strcpy($$->type_id,"DIV");}
        | Exp DIV ASSIGNOP Exp
{$$=mknode(DIVASSIGNOP,$1,$4,NULL,yylineno);strcpy($$->type_id,"DIVASSI
GNOP");}
        | LP Exp RP         {$$=$2;}
        | MINUS Exp %prec UMINUS
{$$=mknode(UMINUS,$2,NULL,NULL,yylineno);strcpy($$->type_id,"UMINUS");
}
        | NOT Exp
{$$=mknode(NOT,$2,NULL,NULL,yylineno);strcpy($$->type_id,"NOT");}
        | ID LP Args RP
{$$=mknode(FUNC_CALL,$3,NULL,NULL,yylineno);strcpy($$->type_id,$1);}
        | ID LP RP
{$$=mknode(FUNC_CALL,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);}
        | Exp LB Exp RB {$$=mknode(EXP_ARRAY,$1,$3,NULL,yylineno);}
        | Exp DOT ID {struct node
*temp=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$3);$$=mkno
de(EXP_ELE,$1,temp,NULL,yylineno);}//$$=mknode(EXP_ELE,$1,$3,NULL,yylin
eno);
        | ID
{$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);}
        | INT
{$$=mknode(INT,NULL,NULL,NULL,yylineno);$$->type_int=$1;$$->type=INT;}
        | FLOAT
{$$=mknode(FLOAT,NULL,NULL,NULL,yylineno);$$->type_float=$1;$$->type=F
LOAT;}
        | CHAR
{$$=mknode(CHAR,NULL,NULL,NULL,yylineno);strcpy(yylval.type_char,$1);$$-
>type=CHAR;}
        | STRING
{$$=mknode(STRING,NULL,NULL,NULL,yylineno);strcpy(yylval.type_string,$1);
$$->type=STRING;}
```

```
            ;

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

    %%

    int main(int argc, char *argv[]){
        yyin=fopen(argv[1],"r");
        if (!yyin) return 0;
        yylineno=1;
        yyparse();
        return 0;
        }

    #include<stdarg.h>
    void yyerror(const char* fmt, ...)
    {
          va_list ap;
          va_start(ap, fmt);
          fprintf(stderr, "===>ERROR:Grammar Error at Line %d Column %d: ",
yylloc.first_line,yylloc.first_column);
          vfprintf(stderr, fmt, ap);
          fprintf(stderr, ".\n");
    }
```

3.def.h

```
    #include "stdio.h"
    #include "stdlib.h"
    #include "string.h"
    #include "stdarg.h"
    #include "parser.tab.h"

    enum node_kind
    {
          EXT_DEF_LIST,
          EXT_VAR_DEF,
          FUNC_DEF,
          FUNC_DEC,
          EXT_STRUCT_DEF,
          STRUCT_DEF,
```

```
    STRUCT_DEC,
    STRUCT_TAG,
    EXP_ELE,
    EXP_ARRAY,
    ARRAY_DEC,
    EXT_DEC_LIST,
    PARAM_LIST,
    PARAM_DEC,
    VAR_DEF,
    DEC_LIST,
    DEF_LIST,
    COMP_STM,
    STM_LIST,
    EXP_STMT,
    FOR_DEC,
    IF_THEN,
    IF_THEN_ELSE,
    FUNC_CALL,
    ARGS,
    FUNCTION,
    PARAM,
    ARG,
    CALL,
    LABEL,
    GOTO,
    JLT,
    JLE,
    JGT,
    JGE,
    EQ,
    NEQ
};

#define MAXLENGTH 1000        //定义符号表的大小
#define DX 3 * sizeof(int) //活动记录控制信息需要的单元数

struct opn
{
    int kind; //标识操作的类型
    int type; //标识操作数的类型
    union {
        int const_int;        //整常数值，立即数
        float const_float; //浮点常数值，立即数
        char const_char;     //字符常数值，立即数
```

```
            char *const_string;
            char id[33]; //变量或临时变量的别名或标号字符串
            struct Array *type_array;
            struct Struct *type_struct;
        };
        int level;   //变量的层号，0 表示是全局变量，数据保存在静态数据区
        int offset; //变量单元偏移量，或函数在符号表的定义位置序号，目标代码生成时用
    };

    struct codenode
    {                                           //三地址 TAC 代码结点,采用双向循环链表存放中间语言代码
        int op;                                 //TAC 代码的运算符种类
        struct opn opn1, opn2, result; //2 个操作数和运算结果
        struct codenode *next, *prior;
    };

    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[31];
    };

    // 使用链表存储多个变量
    struct Array
    {
        int kind;
        union Value value;
        int index;
        struct Array *next;
    };

    // 使用链表存储多个变量
    struct Struct
    {
        int kind;
        char *name; // 字段名字
        union Value value;
        struct Struct *next;
    };
```

```
struct node
{                                   //以下对结点属性定义没有考虑存储效率，只是简单地列出要用到的一些属性
    enum node_kind kind; //结点类型
    char struct_name[33];
    union {
        char type_id[33]; //由标识符生成的叶结点
        int type_int;       //由整常数生成的叶结点
        float type_float; //由浮点常数生成的叶结点
        char type_char;
        char type_string[31];
        struct Array *type_array;
        struct Struct *type_struct;
    };
    struct node *ptr[3];            //子树指针，由 kind 确定有多少棵子树
    int level;                      //层号
    int place;                      //表示结点对应的变量或运算结果符号表的位置序号
    char Etrue[15], Efalse[15]; //对布尔表达式的翻译时，真假转移目标的标号
    char Snext[15];                 //该结点对饮语句执行后的下一条语句位置标号
    struct codenode *code;          //该结点中间代码链表头指针
    char op[10];
    int type;     //结点对应值的类型
    int pos;      //语法单位所在位置行号
    int offset; //偏移量
    int width;    //占数据字节数
    int num;
};

struct symbol
{                               //这里只列出了一个符号表项的部分属性，没考虑属性间的互斥
    char name[33];    //变量或函数名
    int level;          //层号，外部变量名或函数名层号为 0，形参名为 1，每到 1 个复合语句层号加 1，退出减 1
    int type;           //变量类型或函数返回值类型
    int paramnum;     //形式参数个数
    char alias[10]; //别名，为解决嵌套层次使用，使得每一个数据名称唯一
    char flag;          //符号标记，函数：'F'   变量：'V'    参数：'P'   临时变量：'T'
    int offset;       //外部变量和局部变量在其静态数据区或活动记录中的偏移量
```

```
                                    //或函数活动记录大小，目标代码生成时使用
        //其它...
    };
    //符号表，是一个顺序栈，index 初值为 0
    struct symboltable
    {
        struct symbol symbols[MAXLENGTH];
        int index;
    } symbolTable;

    struct symbol_scope_begin
    { /*当前作用域的符号在符号表的起始位置序号,这是一个栈结构，/每到达一个复合语句，将符号表的 index 值进栈，离开复合语句时，取其退栈值修改符号表的 index 值，完成删除该复合语句中的所有变量和临时变量*/
        int TX[30];
        int top;
    } symbol_scope_TX;


    /*generate AST*/
    struct node *mknode(int kind, struct node *first, struct node *second, struct node *third, int pos);

    /*semantic analysis*/
    void semantic_error(int line, char *msg1, char *msg2);
    int searchSymbolTable(char *name);
    int fillSymbolTable(char *name, char *alias, int level, int type, char flag, int offset);
    void Exp(struct node *T);
    void boolExp(struct node *T);
    void semantic_Analysis(struct node *T);
    void DisplaySymbolTable(struct node *T);

    /*inner code generation*/
    char *str_catch(char *s1, char *s2);
    char *newAlias();
    char *newLabel();
    char *newTemp();
    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 print_IR(struct codenode *head);
```

```
void id_exp(struct node *T);
void int_exp(struct node *T);
void assignop_exp(struct node *T);
void relop_exp(struct node *T);
void args_exp(struct node *T);
void op_exp(struct node *T);
void func_call_exp(struct node *T);
void not_exp(struct node *T);
void ext_var_list(struct node *T);
void ext_def_list(struct node *T);
void ext_var_def(struct node *T);
void func_def(struct node *T);
void func_dec(struct node *T);
void param_list(struct node *T);
void param_dec(struct node *T);
void comp_stm(struct node *T);
void def_list(struct node *T);
void var_def(struct node *T);
void stmt_list(struct node *T);
void if_then(struct node *T);
void if_then_else(struct node *T);
void while_dec(struct node *T);
void exp_stmt(struct node *T);
void return_dec(struct node *T);
```

4.display.c

```
#include "def.h"

struct node *mknode(int kind, struct node *first, struct node *second, struct node *third, int pos)
{
    struct node *T = (struct node *)malloc(sizeof(struct node));
    T->kind = kind;
    T->ptr[0] = first;
    T->ptr[1] = second;
    T->ptr[2] = third;
    T->pos = pos;
    return T;
}

//对抽象语法树的先根遍历
```

```
void display(struct node *T, int indent)
{
    int i = 1;
    struct node *T0;

    if (T)
    {
        switch (T->kind)
        {
        case EXT_DEF_LIST:
            display(T->ptr[0], indent); //显示该外部定义列表中的第一个
            display(T->ptr[1], indent); //显示该外部定义列表中的其它外部
定义
            break;
        case EXT_VAR_DEF:
            printf("%*c 外部变量定义：\n", indent, ' ');
            display(T->ptr[0], indent + 3); //显示外部变量类型
            printf("%*c 变量名：\n", indent + 3, ' ');
            display(T->ptr[1], indent + 6); //显示变量列表
            break;
        case TYPE:
            printf("%*c 类型：  %s\n", indent, ' ', T->type_id);
            break;
        case EXT_DEC_LIST:
            display(T->ptr[0], indent); //依次显示外部变量名，
            display(T->ptr[1], indent); //后续还有相同的，仅显示语法树此
处理代码可以和类似代码合并
            break;
        case FUNC_DEF:
            printf("%*c 函数定义：\n", indent, ' ');
            display(T->ptr[0], indent + 3); //显示函数返回类型
            display(T->ptr[1], indent + 3); //显示函数名和参数
            display(T->ptr[2], indent + 3); //显示函数体
            break;
        case FUNC_DEC:
            printf("%*c 函数名：%s\n", indent, ' ', T->type_id);
            if (T->ptr[0])
            {
                printf("%*c 函数形参：\n", indent, ' ');
                display(T->ptr[0], indent + 3); //显示函数参数列表
            }
            else
                printf("%*c 无参函数\n", indent + 3, ' ');
            break;
```

```
            case STRUCT_DEC:
            case STRUCT_DEF:
            case STRUCT_TAG:
            case EXP_ELE:
            case EXP_ARRAY:
                break;
            case PARAM_LIST:
                display(T->ptr[0], indent); //依次显示全部参数类型和名称，
                display(T->ptr[1], indent);
                break;
            case PARAM_DEC:
                if (T->ptr[0]->type == INT)
                    printf("%*c 类型：%s,参数名：%s\n", indent, ' ', "int",
T->ptr[1]->type_id);
                else if (T->ptr[0]->type == FLOAT)
                    printf("%*c 类型：%s,参数名：%s\n", indent, ' ', "float",
T->ptr[1]->type_id);
                else if (T->ptr[0]->type == CHAR)
                    printf("%*c 类型：%s,参数名：%s\n", indent, ' ', "char",
T->ptr[1]->type_id);
                else if (T->ptr[0]->type == STRING)
                    printf("%*c 类型：%s,参数名：%s\n", indent, ' ', "string",
T->ptr[1]->type_id);
                else
                    printf("%*c 类型：%s,参数名：%s\n", indent, ' ',
"unknown", T->ptr[1]->type_id);
                break;
            case EXP_STMT:
                printf("%*c 表达式语句：\n", indent, ' ');
                display(T->ptr[0], indent + 3);
                break;
            case RETURN:
                printf("%*c 返回语句：\n", indent, ' ');
                display(T->ptr[0], indent + 3);
                break;
            case COMP_STM:
                printf("%*c 复合语句：\n", indent, ' ');
                printf("%*c 复合语句的变量定义：\n", indent + 3, ' ');
                display(T->ptr[0], indent + 6); //显示定义部分
                printf("%*c 复合语句的语句部分：\n", indent + 3, ' ');
                display(T->ptr[1], indent + 6); //显示语句部分
                break;
            case STM_LIST:
                display(T->ptr[0], indent); //显示第一条语句
```

```
            display(T->ptr[1], indent); //显示剩下语句
            break;
        case WHILE:
            printf("%*cWHILE 循环语句：\n", indent, ' ');
            printf("%*cWHILE 循环条件：\n", indent + 3, ' ');
            display(T->ptr[0], indent + 6); //显示循环条件
            printf("%*cWHILE 循环体：\n", indent + 3, ' ');
            display(T->ptr[1], indent + 6); //显示循环体
            break;
        case FOR:
            printf("%*cFOR 循环语句：\n", indent, ' ');
            printf("%*cFOR 循环条件：\n", indent + 3, ' ');
            display(T->ptr[0], indent + 6); //显示循环条件
            printf("%*cFOR 循环体：\n", indent + 3, ' ');
            display(T->ptr[1], indent + 6); //显示循环体
            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)：\n", indent, ' ');
            printf("%*c 条件：\n", indent + 3, ' ');
            display(T->ptr[0], indent + 6); //显示条件
            printf("%*cIF 子句：\n", indent + 3, ' ');
            display(T->ptr[1], indent + 6); //显示 if 子句
            break;
        case IF_THEN_ELSE:
            printf("%*c 条件语句(IF_THEN_ELSE)：\n", indent, ' ');
            printf("%*c 条件：\n", indent + 3, ' ');
            display(T->ptr[0], indent + 6); //显示条件
            printf("%*cIF 子句：\n", indent + 3, ' ');
            display(T->ptr[1], indent + 6); //显示 if 子句
            printf("%*cELSE 子句：\n", indent + 3, ' ');
            display(T->ptr[2], indent + 6); //显示 else 子句
            break;
        case DEF_LIST:
            display(T->ptr[0], indent); //显示该局部变量定义列表中的第一
个
            display(T->ptr[1], indent); //显示其它局部变量定义
            break;
        case VAR_DEF:
            printf("%*cLOCAL VAR_NAME：\n", indent, ' ');
```

```
                display(T->ptr[0], indent + 3); //显示变量类型
                display(T->ptr[1], indent + 3); //显示该定义的全部变量名
                break;
            case DEC_LIST:
                printf("%*cVAR_NAME：\n", indent, ' ');
                T0 = T;
                while (T0)
                {
                    if (T0->ptr[0]->kind == ID)
                        printf("%*c %s\n", indent + 3, ' ', T0->ptr[0]->type_id);
                    else if (T0->ptr[0]->kind == ASSIGNOP)
                    {
                        printf("%*c %s ASSIGNOP\n ", indent + 3, ' ',
T0->ptr[0]->ptr[0]->type_id);
                        //显示初始化表达式
                        display(T0->ptr[0]->ptr[1], indent +
strlen(T0->ptr[0]->ptr[0]->type_id) + 4);
                    }
                    else if (T0->ptr[0]->kind == PLUSASSIGNOP)
                    {
                        printf("%*c %s PLUSASSIGNOP\n ", indent + 3, ' ',
T0->ptr[0]->ptr[0]->type_id);
                        //显示初始化表达式
                        display(T0->ptr[0]->ptr[1], indent +
strlen(T0->ptr[0]->ptr[0]->type_id) + 4);
                    }
                    else if (T0->ptr[0]->kind == MINUSASSIGNOP)
                    {
                        printf("%*c %s MINUSASSIGNOP\n ", indent + 3, ' ',
T0->ptr[0]->ptr[0]->type_id);
                        //显示初始化表达式
                        display(T0->ptr[0]->ptr[1], indent +
strlen(T0->ptr[0]->ptr[0]->type_id) + 4);
                    }
                    else if (T0->ptr[0]->kind == STARASSIGNOP)
                    {
                        printf("%*c %s STARASSIGNOP\n ", indent + 3, ' ',
T0->ptr[0]->ptr[0]->type_id);
                        //显示初始化表达式
                        display(T0->ptr[0]->ptr[1], indent +
strlen(T0->ptr[0]->ptr[0]->type_id) + 4);
                    }
                    else if (T0->ptr[0]->kind == DIVASSIGNOP)
                    {
```

```
                                printf("%*c %s DIVASSIGNOP\n ", indent + 3, ' ',
T0->ptr[0]->ptr[0]->type_id);
                                //显示初始化表达式
                                display(T0->ptr[0]->ptr[1], indent +
strlen(T0->ptr[0]->ptr[0]->type_id) + 4);
                            }
                            T0 = T0->ptr[1];
                        }
                        break;
                    case ID:
                        printf("%*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 STRING:
                        printf("%*cSTRING：%s\n", indent, ' ', T->type_string);
                        break;
                    case ASSIGNOP:
                    case MINUSASSIGNOP:
                    case PLUSASSIGNOP:
                    case STARASSIGNOP:
                    case DIVASSIGNOP:
                    case AND:
                    case OR:
                    case RELOP:
                    case PLUS:
                    case MINUS:
                    case STAR:
                    case DIV:
                        printf("%*c%s\n", indent, ' ', T->type_id);
                        display(T->ptr[0], indent + 3);
                        display(T->ptr[1], indent + 3);
                        break;
                    case MMINUS:
                    case PPLUS:
                        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：\n", indent, ' ');
                printf("%*c 函数名：%s\n", indent + 3, ' ', T->type_id);
                display(T->ptr[0], indent + 3);
                break;
            case ARGS:
                i = 1;
                while (T)
                { //ARGS 表示实际参数表达式序列结点，其第一棵子树为其一个实际参数表达式，第二棵子树为剩下的。
                    struct node *T0 = T->ptr[0];
                    printf("%*c 第%d 个实际参数表达式：\n", indent, ' ', i++);
                    display(T0, indent + 3);
                    T = T->ptr[1];
                }
                //                  printf("%*c 第%d 个实际参数表达式：\n",indent,' ',i);
                //                display(T,indent+3);
                printf("\n");
                break;
            }
        }
    }
```

5.semantic_analysis.c

```
#include "def.h"
int LEV = 0;     //层号
int func_size; //函数的活动记录大小


// 收集错误信息
void semantic_error(int line, char *msg1, char *msg2){
    printf("第%d 行,%s %s\n", line, msg1, msg2);
}

int searchSymbolTable(char *name)
{
```

```
        int i;
        for (i = symbolTable.index - 1; i >= 0; i--)
            if (!strcmp(symbolTable.symbols[i].name, name))
                return i;
        return -1;
    }

    // 首先根据 name 查符号表，不能重复定义 重复定义返回-1
    int fillSymbolTable(char *name, char *alias, int level, int type, char flag, int
offset)
    {
        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 temp_add(char *name, int level, int type, char flag, int offset)
    {
        strcpy(symbolTable.symbols[symbolTable.index].name, "");
        strcpy(symbolTable.symbols[symbolTable.index].alias, name);
        symbolTable.symbols[symbolTable.index].level = level;
        symbolTable.symbols[symbolTable.index].type = type;
        symbolTable.symbols[symbolTable.index].flag = flag;
        symbolTable.symbols[symbolTable.index].offset = offset;
        return symbolTable.index++; //返回的是临时变量在符号表中的位置序号
```

```
}

int match_param(int i, struct node *T)
{ //  匹配函数参数
    int j, num = symbolTable.symbols[i].paramnum;
    int type1, type2;
    if (num == 0 && T == NULL)
        return 1;
    for (j = 1; j < num; j++)
    {
/*
        if (!T)
        {
            semantic_error(T->pos, "", "函数调用参数太少");
            return 0;
        }
*/
        type1 = symbolTable.symbols[i + j].type; //形参类型
        type2 = T->ptr[0]->type;
        if (type1 != type2)
        {
            semantic_error(T->pos, "", "参数类型不匹配");
            return 0;
        }
        T = T->ptr[1];
    }
    if (T->ptr[1])
    { //num 个参数已经匹配完，还有实参表达式
        semantic_error(T->pos, "", "函数调用参数太多");
        return 0;
    }
    return 1;
}
//布尔表达式
void boolExp(struct node *T)
{
    struct opn opn1, opn2, result;
    int op;
    int rtn;
    if (T)
    {
        switch (T->kind)
        {
            case INT:
```

```
                if (T->type_int != 0)
                    T->code = genGoto(T->Etrue);
                else
                    T->code = genGoto(T->Efalse);
                T->width = 0;
                break;
            case FLOAT:
                if (T->type_float != 0.0)
                    T->code = genGoto(T->Etrue);
                else
                    T->code = genGoto(T->Efalse);
                T->width = 0;
                break;
            case ID: //查符号表，获得符号表中的位置，类型送 type
                rtn = searchSymbolTable(T->type_id);
                if (rtn == -1)
                    semantic_error(T->pos, T->type_id, "函数未定义，语
义错误");
                if (symbolTable.symbols[rtn].flag == 'F')
                    semantic_error(T->pos, T->type_id, "不是函数名，不
能进行函数调用，语义错误");
                else
                {
                    opn1.kind = ID;
                    strcpy(opn1.id, symbolTable.symbols[rtn].alias);
                    opn1.offset = symbolTable.symbols[rtn].offset;
                    opn2.kind = INT;
                    opn2.const_int = 0;
                    result.kind = ID;
                    strcpy(result.id, T->Etrue);
                    T->code = genIR(NEQ, opn1, opn2, result);
                    T->code = merge(2, T->code, genGoto(T->Efalse));
                }
                T->width = 0;
                break;
            case RELOP: //处理关系运算表达式,2 个操作数都按基本表达
式处理
                T->ptr[0]->offset = T->ptr[1]->offset = T->offset;
                Exp(T->ptr[0]);
                T->width = T->ptr[0]->width;
                Exp(T->ptr[1]);
                if (T->width < T->ptr[1]->width)
                    T->width = T->ptr[1]->width;
                opn1.kind = ID;
```

```
                    strcpy(opn1.id,
symbolTable.symbols[T->ptr[0]->place].alias);
                    opn1.offset = symbolTable.symbols[T->ptr[0]->place].offset;
                    opn2.kind = ID;
                    strcpy(opn2.id,
symbolTable.symbols[T->ptr[1]->place].alias);
                    opn2.offset = symbolTable.symbols[T->ptr[1]->place].offset;
                    result.kind = ID;
                    strcpy(result.id, T->Etrue);
                    if (strcmp(T->type_id, "<") == 0)
                        op = JLT;
                    else if (strcmp(T->type_id, "<=") == 0)
                        op = JLE;
                    else if (strcmp(T->type_id, ">") == 0)
                        op = JGT;
                    else if (strcmp(T->type_id, ">=") == 0)
                        op = JGE;
                    else if (strcmp(T->type_id, "==") == 0)
                        op = EQ;
                    else if (strcmp(T->type_id, "!=") == 0)
                        op = NEQ;
                    T->code = genIR(op, opn1, opn2, result);
                    T->code = merge(4, T->ptr[0]->code, T->ptr[1]->code,
T->code, genGoto(T->Efalse));
                    break;
                case AND:
                case OR:
                    if (T->kind == AND)
                    {
                        strcpy(T->ptr[0]->Etrue, newLabel());
                        strcpy(T->ptr[0]->Efalse, T->Efalse);
                    }
                    else
                    {
                        strcpy(T->ptr[0]->Etrue, T->Etrue);
                        strcpy(T->ptr[0]->Efalse, newLabel());
                    }
                    strcpy(T->ptr[1]->Etrue, T->Etrue);
                    strcpy(T->ptr[1]->Efalse, T->Efalse);
                    T->ptr[0]->offset = T->ptr[1]->offset = T->offset;
                    boolExp(T->ptr[0]);
                    T->width = T->ptr[0]->width;
                    boolExp(T->ptr[1]);
                    if (T->width < T->ptr[1]->width)
```

```
                    T->width = T->ptr[1]->width;
                if (T->kind == AND)
                    T->code = merge(3, T->ptr[0]->code,
genLabel(T->ptr[0]->Etrue), T->ptr[1]->code);
                else
                    T->code = merge(3, T->ptr[0]->code,
genLabel(T->ptr[0]->Efalse), T->ptr[1]->code);
                break;
            case NOT:
                strcpy(T->ptr[0]->Etrue, T->Efalse);
                strcpy(T->ptr[0]->Efalse, T->Etrue);
                boolExp(T->ptr[0]);
                T->code = T->ptr[0]->code;
                break;
            default:
                break;
        }
    }
}

//处理基本表达式
void Exp(struct node *T)
{
    if (T)
    {
        switch (T->kind)
        {
            case ID:
                id_exp(T);
                break;
            case INT:
                int_exp(T);
                break;
            case ASSIGNOP:
                assignop_exp(T);
                break;
            case PPLUS:
            case MMINUS:
            case PLUSASSIGNOP:
            case MINUSASSIGNOP:
            case STARASSIGNOP:
            case DIVASSIGNOP:
                break;
            case AND:
```

```
                    case OR:
                    case RELOP:
                        relop_exp(T);
                        break;
                    case PLUS:
                    case MINUS:
                    case STAR:
                    case DIV:
                        op_exp(T);
                        break;
                    case NOT: //未写完整
                        not_exp(T);
                        break;
                    case UMINUS: //未写完整
                        // uminus_exp(T);
                        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
                        func_call_exp(T);
                        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
                        args_exp(T);
                        break;
                }
            }
        }

    //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）
    void semantic_Analysis(struct node *T){
        if (T)
        {
            switch (T->kind)
            {
            case EXT_DEF_LIST:
                ext_def_list(T);
                break;
            case EXT_VAR_DEF: //处理外部说明,将第一个孩子(TYPE 结点)中的类型送到第二个孩子的类型域
                ext_var_def(T);
                break;
            case FUNC_DEF:
                func_def(T);
```

```
            break;
        case FUNC_DEC: //根据返回类型，函数名填写符号表
            func_dec(T);
            break;
        case PARAM_LIST: //处理函数形式参数列表
            param_list(T);
            break;
        case PARAM_DEC:
            param_dec(T);
            break;
        case COMP_STM:
            comp_stm(T);
            break;
        case DEF_LIST:
            def_list(T);
            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
            var_def(T);
            break;
        case STM_LIST:
            stmt_list(T);
            break;
        case IF_THEN:
            if_then(T);
            break; //控制语句都还没有处理 offset 和 width 属性
        case IF_THEN_ELSE:
            if_then_else(T);
            break;
        case WHILE:
            while_dec(T);
            break;
        case EXP_STMT:
            exp_stmt(T);
            break;
        case RETURN:
            return_dec(T);
            break;
        case ID:
        case STRUCT_TAG:
        case INT:
        case FLOAT:
        case CHAR:
        case STRING:
```

```
            case ASSIGNOP:
            case AND:
            case OR:
            case RELOP:
            case PLUS:
            case PPLUS:
            case PLUSASSIGNOP:
            case MINUS:
            case MMINUS:
            case MINUSASSIGNOP:
            case STAR:
            case STARASSIGNOP:
            case DIV:
            case DIVASSIGNOP:
            case NOT:
            case UMINUS:
            case FUNC_CALL:
            case EXP_ARRAY:
                Exp(T); //处理基本表达式
                break;
            }
        }
}

//打印函数
void DisplaySymbolTable(struct node *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);
    print_IR(T->code);
}
```

```
/* inner code generation */
char *str_catch(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 str_catch("var", s);
}

char *newLabel()
{
    static int no = 1;
    char s[10];
    snprintf(s, 10, "%d", no++);
    // itoa(no++, s, 10);
    return str_catch("label", s);
}

char *newTemp()
{
    static int no = 1;
    char s[10];
    snprintf(s, 10, "%d", no++);
    return str_catch("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, *t1, *t2;
    va_list ap;//指向参数的指针
    va_start(ap, num);//宏初始化 va_list 变量，使其指向第一个可变参数的
地址
    h1 = va_arg(ap, struct codenode *);//返回可变参数，va_arg 的第二个参
数是要返回的参数的类型,如果多个可变参数，依次调用 va_arg 获取各个参数
    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);//使用 va_end 宏结束可变参数的获取
        return h1;
    }

    void print_IR(struct codenode *head)
    {
        char opnstr1[32], opnstr2[32], resultstr[32];
        struct codenode *h = head;
        do
        {
            if (h->opn1.kind == INT)
                sprintf(opnstr1, "#%d", h->opn1.const_int);
            if (h->opn1.kind == FLOAT)
                sprintf(opnstr1, "#%f", h->opn1.const_float);
            if (h->opn1.kind == ID)
                sprintf(opnstr1, "%s", h->opn1.id);
            if (h->opn2.kind == INT)
                sprintf(opnstr2, "#%d", h->opn2.const_int);
            if (h->opn2.kind == FLOAT)
                sprintf(opnstr2, "#%f", h->opn2.const_float);
            if (h->opn2.kind == ID)
                sprintf(opnstr2, "%s", h->opn2.id);
            sprintf(resultstr, "%s", h->result.id);
            switch (h->op)
            {
            case ASSIGNOP:
                printf("    %s := %s\n", resultstr, opnstr1);
                break;
            case PLUS:
            case MINUS:
            case STAR:
            case DIV:
                printf("    %s := %s %c %s\n", resultstr, opnstr1,
                        h->op == PLUS ? '+' : h->op == MINUS ? '-' : h->op ==
STAR ? '*' : '\\', opnstr2);
                break;
            case FUNCTION:
                printf("\nFUNCTION %s :\n", h->result.id);
                break;
            case PARAM:
```

```
                printf("    PARAM %s\n", h->result.id);
                break;
            case LABEL:
                printf("LABEL %s :\n", h->result.id);
                break;
            case GOTO:
                printf("    GOTO %s\n", h->result.id);
                break;
            case JLE:
                printf("    IF %s <= %s GOTO %s\n", opnstr1, opnstr2, resultstr);
                break;
            case JLT:
                printf("    IF %s < %s GOTO %s\n", opnstr1, opnstr2, resultstr);
                break;
            case JGE:
                printf("    IF %s >= %s GOTO %s\n", opnstr1, opnstr2, resultstr);
                break;
            case JGT:
                printf("    IF %s > %s GOTO %s\n", opnstr1, opnstr2, resultstr);
                break;
            case EQ:
                printf("    IF %s == %s GOTO %s\n", opnstr1, opnstr2, resultstr);
                break;
            case NEQ:
                printf("    IF %s != %s GOTO %s\n", opnstr1, opnstr2, resultstr);
                break;
            case ARG:
                printf("    ARG %s\n", h->result.id);
                break;
            case CALL:
                printf("    %s := CALL %s\n", resultstr, opnstr1);
                break;
            case RETURN:
                if (h->result.kind)
                    printf("    RETURN %s\n", resultstr);
                else
                    printf("    RETURN\n");
                break;
            }
            h = h->next;
        } while (h != head);
}

void id_exp(struct node *T)
```

```
    {
        int rtn;

        rtn = searchSymbolTable(T->type_id);
        if (rtn == -1)
            semantic_error(T->pos, T->type_id, "变量未声明定义就引用，语义
错误");
        if (symbolTable.symbols[rtn].flag == 'F')
            semantic_error(T->pos, T->type_id, "是函数名，不是普通变量，语
义错误");
        else
        {
            T->place = rtn; //结点保存变量在符号表中的位置
            T->code = NULL; //标识符不需要生成 TAC
            T->type = symbolTable.symbols[rtn].type;
            T->offset = symbolTable.symbols[rtn].offset;
            T->width = 0; //未再使用新单元
        }
    }

    void int_exp(struct node *T){
        struct opn opn1, opn2, result;
        T->place = temp_add(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;
    }

    void assignop_exp(struct node *T)
    {
        struct opn opn1, opn2, result;
        if (T->ptr[0]->kind != ID)
        {
            semantic_error(T->pos, "", "赋值语句没有左值，语义错误");
        }
        else
        {
            Exp(T->ptr[0]); //处理左值，例中仅为变量
```

```
            T->ptr[1]->offset = T->offset;
            Exp(T->ptr[1]);
            T->type = T->ptr[0]->type;
            T->width = T->ptr[1]->width;
            T->code = merge(2, T->ptr[0]->code, T->ptr[1]->code);

            opn1.kind = ID;
            strcpy(opn1.id, symbolTable.symbols[T->ptr[1]->place].alias); //右值
一定是个变量或临时变量
            opn1.offset = symbolTable.symbols[T->ptr[1]->place].offset;
            result.kind = ID;
            strcpy(result.id, symbolTable.symbols[T->ptr[0]->place].alias);
            result.offset = symbolTable.symbols[T->ptr[0]->place].offset;
            T->code = merge(2, T->code, genIR(ASSIGNOP, opn1, opn2, result));
        }
    }

    void relop_exp(struct node *T)
    {
        T->type = INT;
        T->ptr[0]->offset = T->ptr[1]->offset = T->offset;
        Exp(T->ptr[0]);
        Exp(T->ptr[1]);
    }

    void args_exp(struct node *T)
    {
        T->ptr[0]->offset = T->offset;
        Exp(T->ptr[0]);
        T->width = T->ptr[0]->width;
        T->code = T->ptr[0]->code;
        if (T->ptr[1])
        {
            T->ptr[1]->offset = T->offset + T->ptr[0]->width;
            Exp(T->ptr[1]);
            T->width += T->ptr[1]->width;
            T->code = merge(2, T->code, T->ptr[1]->code);
        }
    }

    // 算数运算：加减乘除
    void op_exp(struct node *T)
    {
        struct opn opn1, opn2, result;
```

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

            T->place = temp_add(newTemp(), LEV, T->type, 'T', T->offset + T->ptr[0]->width + T->ptr[1]->width);
            opn1.kind = ID;
            strcpy(opn1.id, symbolTable.symbols[T->ptr[0]->place].alias);
            opn1.type = T->ptr[0]->type;
            opn1.offset = symbolTable.symbols[T->ptr[0]->place].offset;
            opn2.kind = ID;
            strcpy(opn2.id, symbolTable.symbols[T->ptr[1]->place].alias);
            opn2.type = T->ptr[1]->type;
            opn2.offset = symbolTable.symbols[T->ptr[1]->place].offset;
            result.kind = ID;
            strcpy(result.id, symbolTable.symbols[T->place].alias);
            result.type = T->type;
            result.offset = symbolTable.symbols[T->place].offset;
            T->code = merge(3, T->ptr[0]->code, T->ptr[1]->code, genIR(T->kind, opn1, opn2, result));
            T->width = T->ptr[0]->width + T->ptr[1]->width + 4;//INT
        }

        void func_call_exp(struct node *T)
        {
            int rtn,width;
            struct node *T0;
            struct opn opn1, opn2, result;
            rtn = searchSymbolTable(T->type_id);
            if (rtn == -1)
            {
                semantic_error(T->pos, T->type_id, "函数未定义，语义错误");
                return;
            }
            if (symbolTable.symbols[rtn].flag != 'F')
            {
                semantic_error(T->pos, T->type_id, "不是函数名，不能进行函数调用，语义错误");
                return;
```

```
        }
        T->type = symbolTable.symbols[rtn].type;
        width = T->type == INT ? 4 : 8; //存放函数返回值的单数字节数
        if (T->ptr[0])
        {
            T->ptr[0]->offset = T->offset;
            Exp(T->ptr[0]);                                    //处理所有实参表达式求
值，及类型
            T->width = T->ptr[0]->width + width; //累加上计算实参使用临时变
量的单元数
            T->code = T->ptr[0]->code;
        }
        else
        {
            T->width = width;
            T->code = NULL;
        }
        match_param(rtn, T->ptr[0]); //处理所以参数的匹配
            //处理参数列表的中间代码
        T0 = T->ptr[0];
        while (T0)
        {
            result.kind = ID;
            strcpy(result.id, symbolTable.symbols[T0->ptr[0]->place].alias);
            result.offset = symbolTable.symbols[T0->ptr[0]->place].offset;
            T->code = merge(2, T->code, genIR(ARG, opn1, opn2, result));
            T0 = T0->ptr[1];
        }
        T->place = temp_add(newTemp(), LEV, T->type, 'T', T->offset + T->width -
width);
        opn1.kind = ID;
        strcpy(opn1.id, T->type_id); //保存函数名
        opn1.offset = rtn;                    //这里 offset 用以保存函数定义入口,在目
标代码生成时，能获取相应信息
        result.kind = ID;
        strcpy(result.id, symbolTable.symbols[T->place].alias);
        result.offset = symbolTable.symbols[T->place].offset;
        T->code = merge(2, T->code, genIR(CALL, opn1, opn2, result)); //生成函
数调用中间代码
    }

    void not_exp(struct node *T)
    {
        T->type = INT;
```

```
        T->ptr[0]->offset = T->offset;
        Exp(T->ptr[0]);
    }

    //处理变量列表
    void ext_var_list(struct node *T)
    {
        int rtn, num = 1;
        switch (T->kind)
        {
        case EXT_DEC_LIST:
            T->ptr[0]->type = T->type;                    //将类型属性向下传递
变量结点
            T->ptr[0]->offset = T->offset;                //外部变量的偏移量向下
传递
            T->ptr[1]->type = T->type;                    //将类型属性向下传递
变量结点
            T->ptr[1]->offset = T->offset + T->width; //外部变量的偏移量向下传
递
            T->ptr[1]->width = T->width;
            ext_var_list(T->ptr[0]);
            ext_var_list(T->ptr[1]);
            T->num = T->ptr[1]->num + 1;
            break;
        case ID:
            rtn = fillSymbolTable(T->type_id, newAlias(), LEV, T->type, 'V',
T->offset); //最后一个变量名
            if (rtn == -1)
                semantic_error(T->pos, T->type_id, "变量重复定义，语义错误
");
            else
                T->place = rtn;
            T->num = 1;
            break;
        default:
            break;
        }
    }

    void ext_def_list(struct node *T)
    {
        if (!T->ptr[0])
            return;
```

```
        // 语义分析之前先设置偏移地址
        T->ptr[0]->offset = T->offset;
        semantic_Analysis(T->ptr[0]); //访问外部定义列表中的第一个
        // 之后合并 code
        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);
        }
    }

    void ext_var_def(struct node *T)
    {
        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->type = T->ptr[1]->type = !strcmp(T->ptr[0]->type_id, "int") ? INT :
FLOAT;
        T->ptr[1]->offset = T->offset; //这个外部变量的偏移量向下传递
        // T->ptr[1]->width = T->type == INT ? 4 : 8;                    //将一个变量
的宽度向下传递
        ext_var_list(T->ptr[1]);                                        //处理外部变量说明
中的标识符序列
        T->width = (T->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
```

```
}

void func_def(struct node *T)
{
    if (!strcmp(T->ptr[0]->type_id, "int"))
    {
        T->ptr[1]->type = INT;
    }
    if (!strcmp(T->ptr[0]->type_id, "float"))
    {
        T->ptr[1]->type = FLOAT;
    }
    if (!strcmp(T->ptr[0]->type_id, "char"))
    {
        T->ptr[1]->type = CHAR;
    }
    //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
    T->width = 0;                       //函数的宽度设置为 0，不会对外部变量的地址分配产生影响
    T->offset = DX;                     //设置局部变量在活动记录中的偏移量初值
    semantic_Analysis(T->ptr[1]);   //处理函数名和参数结点部分，这里不考虑用寄存器传递参数
    T->offset += T->ptr[1]->width; //用形参单元宽度修改函数局部变量的起始偏移量
    T->ptr[2]->offset = T->offset;
    strcpy(T->ptr[2]->Snext, newLabel()); //函数体语句执行结束后的位置属性
    semantic_Analysis(T->ptr[2]);           //处理函数体结点
    //计算活动记录大小,这里 offset 属性存放的是活动记录大小，不是偏移
    symbolTable.symbols[T->ptr[1]->place].offset = T->offset + T->ptr[2]->width;
    T->code = merge(3, T->ptr[1]->code, T->ptr[2]->code, genLabel(T->ptr[2]->Snext)); //函数体的代码作为函数的代码
}

void func_dec(struct node *T)
{
    int rtn;
    struct opn opn1, opn2, result;
    rtn = fillSymbolTable(T->type_id, newAlias(), LEV, T->type, 'F', 0); //函数不在数据区中分配单元，偏移量为 0
```

```
            if (rtn == -1)
            {
                semantic_error(T->pos, T->type_id, "函数名重复使用，可能是函数
重复定义，语义错误");
                return;
            }
            else
                T->place = rtn;
            result.kind = ID;
            strcpy(result.id, T->type_id);
            result.offset = rtn;
            T->code = genIR(FUNCTION, opn1, opn2, result); //生成中间代码：
FUNCTION  函数名
            T->offset = DX;                                    //设置形式参数在
活动记录中的偏移量初值
            if (T->ptr[0])
            { //判断是否有参数
                T->ptr[0]->offset = T->offset;
                semantic_Analysis(T->ptr[0]); //处理函数参数列表
                T->width = T->ptr[0]->width;
                symbolTable.symbols[rtn].paramnum = T->ptr[0]->num;
                T->code = merge(2, T->code, T->ptr[0]->code); //连接函数名和参数
代码序列
            }
            else
                symbolTable.symbols[rtn].paramnum = 0, T->width = 0;
        }

        void param_list(struct node *T)
        {
            T->ptr[0]->offset = T->offset;
            semantic_Analysis(T->ptr[0]);
            if (T->ptr[1])
            {
                T->ptr[1]->offset = T->offset + T->ptr[0]->width;
                semantic_Analysis(T->ptr[1]);
                T->num = T->ptr[0]->num + T->ptr[1]->num;                  //统计参
数个数
                T->width = T->ptr[0]->width + T->ptr[1]->width;          //累加参数
单元宽度
                T->code = merge(2, T->ptr[0]->code, T->ptr[1]->code); //连接参数代
码
            }
            else
```

```
        {
            T->num = T->ptr[0]->num;
            T->width = T->ptr[0]->width;
            T->code = T->ptr[0]->code;
        }
    }

    void param_dec(struct node *T)
    {
        int rtn;
        struct opn opn1, opn2, result;
        rtn = fillSymbolTable(T->ptr[1]->type_id, newAlias(), 1, T->ptr[0]->type,
'P', T->offset);
        if (rtn == -1)
            semantic_error(T->ptr[1]->pos, T->ptr[1]->type_id, "参数名重复定义,
语义错误");
        else
            T->ptr[1]->place = rtn;
        T->num = 1;                                          //参数个数计算的初始
值
        T->width = T->ptr[0]->type == INT ? 4 : 8; //参数宽度
        result.kind = ID;
        strcpy(result.id, symbolTable.symbols[rtn].alias);
        result.offset = T->offset;
        T->code = genIR(PARAM, opn1, opn2, result); //生成：FUNCTION 函数
名
    }

    void comp_stm(struct node *T)
    {
        LEV++;
        //设置层号加 1，并且保存该层局部变量在符号表中的起始位置在
symbol_scope_TX
        symbol_scope_TX.TX[symbol_scope_TX.top++] = symbolTable.index;
        T->width = 0;
        T->code = NULL;
        if (T->ptr[0])
        {
            T->ptr[0]->offset = T->offset;
            semantic_Analysis(T->ptr[0]); //处理该层的局部变量 DEF_LIST
            T->width += T->ptr[0]->width;
            T->code = T->ptr[0]->code;
        }
        if (T->ptr[1])
```

```
        {
            T->ptr[1]->offset = T->offset + T->width;
            strcpy(T->ptr[1]->Snext, T->Snext); //S.next 属性向下传递
            semantic_Analysis(T->ptr[1]);           //处理复合语句的语句序列
            T->width += T->ptr[1]->width;
            T->code = merge(2, T->code, T->ptr[1]->code);
        }
        //print_symbol();
//c 在退出一个符合语句前显示的符号表
        LEV--;
//出复合语句，层号减 1
        symbolTable.index = symbol_scope_TX.TX[--symbol_scope_TX.top]; //删
除该作用域中的符号
    }

    void def_list(struct node *T)
    {
        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;
        }
    }

    void var_def(struct node *T)
    {
        int rtn, num, width;
        struct node *T0;
        struct opn opn1, opn2, result;
        T->code = NULL;
        if (!strcmp(T->ptr[0]->type_id, "int"))
        {
            T->ptr[1]->type = INT;
            width = 4;
```

```
        }
        if (!strcmp(T->ptr[0]->type_id, "float"))
        {
            T->ptr[1]->type = FLOAT;
            width = 8;
        }
        if (!strcmp(T->ptr[0]->type_id, "char"))
        {
            T->ptr[1]->type = CHAR;
            width = 1;
        }
        if (!strcmp(T->ptr[0]->type_id, "string"))
        {
            T->ptr[1]->type = STRING;
        }
        // T->ptr[1]->type = !strcmp(T->ptr[0]->type_id, "int") ? INT : FLOAT; //确
定变量序列各变量类型
        T0 = T->ptr[1]; //T0 为变量名列表子树根指针，对 ID、ASSIGNOP 类
结点在登记到符号表，作为局部变量
        num = 0;
        T0->offset = T->offset;
        T->width = 0;
        // width = T->ptr[1]->type == INT ? 4 : 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;
            if (T0->ptr[0]->kind == ID)
            {
                rtn = fillSymbolTable(T0->ptr[0]->type_id, newAlias(), LEV,
T0->ptr[0]->type, 'V', T->offset + T->width); //此处偏移量未计算，暂时为 0
                if (rtn == -1)
                    semantic_error(T0->ptr[0]->pos, T0->ptr[0]->type_id, "变量
重复定义");
                else
                    T0->ptr[0]->place = rtn;
                T->width += width;
            }
```

```
            else if (T0->ptr[0]->kind == ASSIGNOP)
            {
                rtn = fillSymbolTable(T0->ptr[0]->ptr[0]->type_id, newAlias(),
LEV, T0->ptr[0]->type, 'V', T->offset + T->width); //此处偏移量未计算，暂时为 0
                if (rtn == -1)
                    semantic_error(T0->ptr[0]->ptr[0]->pos,
T0->ptr[0]->ptr[0]->type_id, "变量重复定义");
                else
                {
                    T0->ptr[0]->place = rtn;
                    T0->ptr[0]->ptr[1]->offset = T->offset + T->width + width;
                    Exp(T0->ptr[0]->ptr[1]);
                    opn1.kind = ID;
                    strcpy(opn1.id,
symbolTable.symbols[T0->ptr[0]->ptr[1]->place].alias);
                    result.kind = ID;
                    strcpy(result.id,
symbolTable.symbols[T0->ptr[0]->place].alias);
                    T->code = merge(3, T->code, T0->ptr[0]->ptr[1]->code,
genIR(ASSIGNOP, opn1, opn2, result));
                }
                T->width += width + T0->ptr[0]->ptr[1]->width;
            }
            T0 = T0->ptr[1];
        }
    }

    void stmt_list(struct node *T)
    {
        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]);
        T->code = T->ptr[0]->code;
        T->width = T->ptr[0]->width;
```

```
        if (T->ptr[1])
        { //2 条以上语句连接,S.next 属性向下传递
            strcpy(T->ptr[1]->Snext, T->Snext);
            T->ptr[1]->offset = T->offset; //顺序结构共享单元方式
                                                    //
T->ptr[1]->offset=T->offset+T->ptr[0]->width; //顺序结构顺序分配单元方式
            semantic_Analysis(T->ptr[1]);
            //序列中第 1 条为表达式语句，返回语句，复合语句时，第 2 条前
不需要标号
            if (T->ptr[0]->kind == RETURN || T->ptr[0]->kind == EXP_STMT ||
T->ptr[0]->kind == COMP_STM)
                T->code = merge(2, T->code, T->ptr[1]->code);
            else
                T->code = merge(3, T->code, genLabel(T->ptr[0]->Snext),
T->ptr[1]->code);
            if (T->ptr[1]->width > T->width)
                T->width = T->ptr[1]->width; //顺序结构共享单元方式
                                                    //
T->width+=T->ptr[1]->width;//顺序结构顺序分配单元方式
        }
    }

    void if_then(struct node *T)
    {
        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);
    }

    void if_then_else(struct node *T)
    {
        strcpy(T->ptr[0]->Etrue, newLabel()); //设置条件语句真假转移位置
        strcpy(T->ptr[0]->Efalse, newLabel());
        T->ptr[0]->offset = T->ptr[1]->offset = T->ptr[2]->offset = T->offset;
        boolExp(T->ptr[0]); //条件，要单独按短路代码处理
        T->width = T->ptr[0]->width;
```

```
        strcpy(T->ptr[1]->Snext, T->Snext);
        semantic_Analysis(T->ptr[1]); //if 子句
        if (T->width < T->ptr[1]->width)
            T->width = T->ptr[1]->width;
        strcpy(T->ptr[2]->Snext, T->Snext);
        semantic_Analysis(T->ptr[2]); //else 子句
        if (T->width < T->ptr[2]->width)
            T->width = T->ptr[2]->width;
        T->code = merge(6, T->ptr[0]->code, genLabel(T->ptr[0]->Etrue),
T->ptr[1]->code,
                        genGoto(T->Snext), genLabel(T->ptr[0]->Efalse),
T->ptr[2]->code);
    }

    void while_dec(struct node *T)
    {
        strcpy(T->ptr[0]->Etrue, newLabel()); //子结点继承属性的计算
        strcpy(T->ptr[0]->Efalse, T->Snext);
        T->ptr[0]->offset = T->ptr[1]->offset = T->offset;
        boolExp(T->ptr[0]); //循环条件，要单独按短路代码处理
        T->width = T->ptr[0]->width;
        strcpy(T->ptr[1]->Snext, newLabel());
        semantic_Analysis(T->ptr[1]); //循环体
        if (T->width < T->ptr[1]->width)
            T->width = T->ptr[1]->width;
        T->code = merge(5, genLabel(T->ptr[1]->Snext), T->ptr[0]->code,
                        genLabel(T->ptr[0]->Etrue), T->ptr[1]->code,
genGoto(T->ptr[1]->Snext));
    }

    void exp_stmt(struct node *T)
    {
        T->ptr[0]->offset = T->offset;
        semantic_Analysis(T->ptr[0]);
        T->code = T->ptr[0]->code;
        T->width = T->ptr[0]->width;
    }

    void return_dec(struct node *T)
    {
        int num;
        struct opn opn1, opn2, result;
        if (T->ptr[0])
        {
```

```
                T->ptr[0]->offset = T->offset;
                Exp(T->ptr[0]);
                num = symbolTable.index;
                do
                    num--;
                while (symbolTable.symbols[num].flag != 'F');
                if (T->ptr[0]->type != symbolTable.symbols[num].type)
                {
                    semantic_error(T->pos, "返回值类型错误，语义错误", "");
                    T->width = 0;
                    T->code = NULL;
                    return;
                }
                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);
            }
        }
```

6.main.cpp

```
    #include <iostream>
    #include <vector>
    #include <regex>
    #include <fstream>

    using namespace std;

    string regs[] =
{"t1","t2","t3","t4","t5","t6","t7","t8","t9","s0","s1","s2","s3","s4","s5","s6","s7"};
    vector<string> variables;
    map<string,string> table;
    map<string,int> reg_ok;
```

```
void Load_Var(string Inter){
regex temp_re("temp\\d+");
smatch result;
string temp_line;
string::const_iterator iter = Inter.begin();
string::const_iterator iterEnd = Inter.end();
while (regex_search(iter,iterEnd,result,temp_re)){
temp_line = result[0];
//cout<<temp_line<<endl;
variables.emplace_back(temp_line);
iter = result[0].second;
}
//      for(auto it = variables.begin();it!=variables.end();it++)
//              cout<<*it<<endl;
}

string Load_Inter(const string& filename){
string lines;
string temp_line;
ifstream in(filename);
if(in){
while (getline(in,temp_line)){
if (temp_line == " ")
continue;
lines.append(temp_line);
lines.append("\n");
}
}
in.close();
return lines;
}

string Get_R(const string& temp_str){
for (auto it = variables.begin();it!=variables.end();++it)
if (*it == temp_str){
it = variables.erase(it);
break;
}

if (table.find(temp_str) != table.end())//如果已经存在寄存器分配，那么直接返回寄存器
return "$"+table[temp_str];
else{//没找到
vector<string> keys;
```

```
for (auto & it : table)//已经分配寄存器的变量 key
keys.emplace_back(it.first);

for (auto & key : keys)//当遇到未分配寄存器的变量时，清空之前所有分配的临时变量的映射关系
if (key.find("temp")!=string::npos && find(variables.begin(),variables.end(),key) == variables.end()){
reg_ok[table[key]] = 1;
table.erase(key);
}

for (const auto & reg : regs) //对于所有寄存器
if(reg_ok[reg] == 1){ //如果寄存器可用
table[temp_str] = reg;//将可用寄存器分配给该变量，映射关系存到 table 中
reg_ok[reg] = 0;//寄存器 reg 设置为已用
return "$"+reg;
}
}
}

string translate(string temp_str){
//将每行 string 按空格存成数组
vector<string> line;
string temp_res;
stringstream input(temp_str);
while (input>>temp_res)
line.emplace_back(temp_res);
//核心处理
string temp_return;
if(line[0] == "LABEL")
return line[1]+":";
if (line[1] == ":=") {
if (line.size() == 3)
if (temp_str[temp_str.length()-2] == '#')
return "\tli " + Get_R(line[0]) + ","+line.back().back();
else{
temp_return = "\tmove ";
temp_return += Get_R(line[0])+',';
temp_return += Get_R(line[2]);
return temp_return;
}
if (line.size() == 5){
if (line[3] == "+")
if (temp_str[temp_str.length()-2] == '#'){
```

```
temp_return = "\taddi ";
temp_return += Get_R(line[0])+",";
temp_return += Get_R(line[2])+",";
temp_return += line.back().back();
return temp_return;
}
else{
temp_return = "\tadd ";
temp_return += Get_R(line[0])+",";
temp_return += Get_R(line[2])+",";
temp_return += Get_R(line.back());
return temp_return;
}
else if (line[3] == "-"){
if (temp_str[temp_str.length()-2] == '#'){
temp_return = "\taddi ";
temp_return += Get_R(line[0])+",";
temp_return += Get_R(line[2])+",";
temp_return += line.back().back();
return temp_return;
}
else{
temp_return = "\tsub ";
temp_return += Get_R(line[0])+",";
temp_return += Get_R(line[2])+",";
temp_return += Get_R(line.back());
return temp_return;
}
}
else if (line[3] == "*"){
temp_return = "\tmul ";
temp_return += Get_R(line[0])+",";
temp_return += Get_R(line[2])+",";
temp_return += Get_R(line.back());
return temp_return;
}
else if (line[3] == "/"){
temp_return = "\tdiv ";
temp_return += Get_R(line[2])+",";
temp_return += Get_R(line.back()) + "\n\tmflo ";
temp_return += Get_R(line[0]);
return temp_return;
}
else if (line[3] == "<"){
```

```
        temp_return = "\tslt ";
        temp_return += Get_R(line[0])+",";
        temp_return += Get_R(line[2])+",";
        temp_return += Get_R(line.back());
        return temp_return;
        }
        else if (line[3] == ">"){
        temp_return = "\tslt ";
        temp_return += Get_R(line[0])+",";
        temp_return += Get_R(line.back())+",";
        temp_return += Get_R(line[2]);
        return temp_return;
        }
        }

        if (line[2] == "CALL")
        if (line[3] == "read" || line[3] == "print")
        return "\taddi $sp,$sp,-4\n\tsw $ra,0($sp)\n\tjal " + line.back() + "\n\tlw
$ra,0($sp)\n\tmove " +Get_R(line[0]) + ",$v0\n\taddi $sp,$sp,4";
        else
        return "\taddi $sp,$sp,-24\n\tsw $t0,0($sp)\n\tsw $ra,4($sp)\n\tsw
$t1,8($sp)\n\tsw $t2,12($sp)\n\tsw $t3,16($sp)\n\tsw $t4,20($sp)\n\tjal
"+line.back()+"\n\tlw $a0,0($sp)\n\tlw $ra,4($sp)\n\tlw $t1,8($sp)\n\tlw
$t2,12($sp)\n\tlw $t3,16($sp)\n\tlw $t4,20($sp)\n\taddi $sp,$sp,24\n\tmove
"+Get_R(line[0])+" $v0";
        }
        if (line[0] == "GOTO")
        return "\tj "+line[1];
        if (line[0] == "RETURN")
        return "\tmove $v0,"+Get_R(line[1])+"\n\tjr $ra";
        if (line[0] == "IF") {
        if (line[2] == "=="){
        temp_return = "\tbeq ";
        temp_return += Get_R(line[1])+",";
        temp_return += Get_R(line[3])+",";
        temp_return += line.back();
        return temp_return;
        }
        if (line[2] == "!="){
        temp_return = "\tbne ";
        temp_return += Get_R(line[1])+",";
        temp_return += Get_R(line[3])+",";
        temp_return += line.back();
        return temp_return;
```

```
        }
        if (line[2] == ">"){
        temp_return = "\tbgt ";
        temp_return += Get_R(line[1])+",";
        temp_return += Get_R(line[3])+",";
        temp_return += line.back();
        return temp_return;
        }
        if (line[2] == "<"){
        temp_return = "\tblt ";
        temp_return += Get_R(line[1])+",";
        temp_return += Get_R(line[3])+",";
        temp_return += line.back();
        return temp_return;
        }
        if (line[2] == ">="){
        temp_return = "\tbge ";
        temp_return += Get_R(line[1])+",";
        temp_return += Get_R(line[3])+",";
        temp_return += line.back();
        return temp_return;
        }
        if (line[2] == "<="){
        temp_return = "\tble ";
        temp_return += Get_R(line[1])+",";
        temp_return += Get_R(line[3])+",";
        temp_return += line.back();
        return temp_return;
        }
        }
        if (line[0] == "FUNCTION")
        return line[1]+":";
        if (line[0] == "CALL")
        if (line.back() == "read" || line.back() == "print")
        return "\taddi $sp,$sp,-4\n\tsw $ra,0($sp)\n\tjal "+line.back()+"\n\tlw
$ra,0($sp)\n\taddi $sp,$sp,4";
        else
        return "\taddi $sp,$sp,-24\n\tsw $t0,0($sp)\n\tsw $ra,4($sp)\n\tsw
$t1,8($sp)\n\tsw $t2,12($sp)\n\tsw $t3,16($sp)\n\tsw $t4,20($sp)\n\tjal
"+line.back()+"\n\tlw $a0,0($sp)\n\tlw $ra,4($sp)\n\tlw $t1,8($sp)\n\tlw
$t2,12($sp)\n\tlw $t3,16($sp)\n\tlw $t4,20($sp)\n\taddi $sp,$sp,24\n\tmove
"+Get_R(line[0])+" $v0";
        if (line[0] == "ARG")
        return "\tmove $t0,$a0\n\tmove $a0,"+Get_R(line.back());
```

```
if (line[0] == "PARAM")
table[line.back()] = "a0";
return " ";
}

void write_to_txt(const vector<string>& obj){
ofstream out("demo.asm");
string temp =".data\n"
"_prompt: .asciiz \"Enter an integer:\"\n"
"_ret: .asciiz \"\\n\"\n"
".globl main\n"
".text\n"
"read:\n"
"       li $v0,4\n"
"       la $a0,_prompt\n"
"       syscall\n"
"       li $v0,5\n"
"       syscall\n"
"       jr $ra\n"
"\n"
"print:\n"
"       li $v0,1\n"
"       syscall\n"
"       li $v0,4\n"
"       la $a0,_ret\n"
"       syscall\n"
"       move $v0,$0\n"
"       jr $ra\n";
out<<temp;
for (auto & it:obj)
out<<it<<endl;
out.close();
}

int main(){
for (const auto & reg : regs) //初始化，所有寄存器都可用
reg_ok[reg] = 1;

string filename = "inter.txt";
string Inter = Load_Inter(filename);//读取中间代码

Load_Var(Inter);//第一遍扫描，记录所有变量
//翻译
vector<string> obj;
```

```
ifstream in(filename);
string temp_line,obj_line;
if(in) {
while (getline(in, temp_line)) {
obj_line = translate(temp_line);
if (obj_line == " ")
continue;
obj.emplace_back(obj_line);
}
} else{
cout<<"file open falied.\n";
}
in.close();
write_to_txt(obj);
return 0;
}
```