

华中科技大学

实验报告

题目: miniC - Compiler

课程名称: 编译原理

专业班级: 软件工程 1804

学 号: U201817065

姓 名: 赵胜豪

指导教师: 胡雯蕾

报告日期: 2020/12/01

软件学院

目录

1 概述	1
2 系统描述.....	2
2.1 自定义语言描述.....	2
2.2 单词文法与语言文法.....	2
2.3 符号表结构定义.....	6
2.4 错误类型码定义.....	6
2.5 中间代码结构定义.....	7
2.6 目标代码指令集则.....	8
3 系统设计与实现.....	12
3.1 词法分析器.....	12
3.2 语法分析器.....	13
3.3 符号表管理.....	16
3.4 语义检查.....	17
3.5 报错功能.....	18
3.6 中间代码生成.....	19
3.7 代码优化.....	20
3.6 汇编代码生成.....	20
4 系统测试与评价.....	20
4.1 测试用例.....	20
4.2 正确性测试.....	23
4.3 报错功能测试.....	26
4.4 系统的优点.....	27
4.5 系统的缺点.....	27
5 实验小结或体会.....	28
参考文献	29

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 自定义语言描述

- 本实验定义的语言是 miniC 语言，语法与 C 语言相近。所定义的源语言包含的语言成分如下：

数据类型包括 char 类型、int 类型和 float 类型

基本运算包括算术运算、比较运算、逻辑运算、自增自减运算和复合赋值运算

控制语句包括 if 语句和 while 语句

其它补充选项实现的有：数组、for 循环

2.2 单词文法与语言文法

mini C 中的单词可以分为 6 类：标识符、关键字、运算符、界符、常量以及注释，具体每种单词的文法定义如表 2.1 所示：

表 2.1：单词的文法定义

单词符号说明	单词种类码	正则表达式
{char}	CHAR	(\[^\n\])
{int}	INT	[0-9]+
{float}	FLOAT	([0-9]*\.[0-9]+) ([0-9]+\.)
“char”	TYPE	
“int”	TYPE	
“float”	TYPE	
"string"	TYPE	
return	RETURN	
if	IF	
else	ELSE	
while	WHILE	
for	FOR	
id	ID	[A-Za-z][A-Za-z0-9]*
“,”	SEMI	
“,”	COMMA	
">" "<" ">=" "<=" "==" "!="	RELOP	
“=”	ASSIGNOP	
“+”	PLUS	

“-“	MINUS	
“++”	DPLUS	
“--”	DMINUS	
“*”	STAR	
“/”	DIV	
“&&”	AND	
“ ”	OR	
“!”	NOT	
“(”	LP	
)”	RP	
“[”	LB	
“]”	RB	
“{”	LC	
“}”	RC	
[n]	换行，无种类码	
[r\t]	换行，无种类码	

对语言的文法定义如下代码所示：

```

program: ExtDefList      {printf("程序体: \n"); display($1,0); }      //显示语法树,语义分析
魔改 semantic_Analysis($1);
      ;
ExtDefList: {$$=NULL;}
      | ExtDef ExtDefList {$$=mknode(2,EXT_DEF_LIST,yylineno,$1,$2);}      //每一
个 EXTDEFLIST 的结点，其第 1 棵子树对应一个外部变量声明或函数
      ;
ExtDef:   Specifier ExtDecList SEMI      {$$=mknode(2,EXT_VAR_DEF,yylineno,$1,$2);}
//该结点对应一个外部变量声明
      | Specifier FuncDec CompSt      {$$=mknode(3,FUNC_DEF,yylineno,$1,$2,$3);}
//该结点对应一个函数定义
      | error SEMI      {$$=NULL;}
      ;
Specifier:                                     TYPE
{$$=mknode(0,TYPE,yylineno);strcpy($$->type_id,$1);if(!strcmp($1,"int"))
$$->type=INT;if(!strcmp($1,"float"))      $$->type=FLOAT;if(!strcmp($1,"char"))
$$->type=CHAR;}
      ;
ExtDecList:  VarDec      {$$=$1;}      /*每一个 EXT_DECLIST 的结点，其第一棵子
树对应一个变量名(ID 类型的结点),第二棵子树对应剩下的外部变量名*/
      |      VarDec      COMMA      ExtDecList
{$$=mknode(2,EXT_DEC_LIST,yylineno,$1,$3);}
      ;

```

```

VarDec: ID          {$$=mknode(0,ID,yylineno);strcpy($$->type_id,$1);} //ID 结点,
标识符号串存放结点的 type_id
      |
      ID          DimensionList
      {$$=mknode(1,DIMENSION,yylineno,$2);strcpy($$->type_id,$1);} //数组声名
      ;
DimensionList: LB          INT          RB
      {$$=mknode(1,DIMENSION_LIST,yylineno,$2);$$->type_int=$2;} //匹配数组的 [?]
      |
      LB          INT          RB          DimensionList
      {$$=mknode(2,DIMENSION_LIST,yylineno,$2,$4);$$->type_int=$2;}
      ;
FuncDec: ID          LP          VarList          RP
{$$=mknode(1,FUNC_DEC,yylineno,$3);strcpy($$->type_id,$1);} // 函数名存放在
$$->type_id
      |ID          LP          RP
      {$$=mknode(0,FUNC_DEC,yylineno);strcpy($$->type_id,$1);$$->ptr[0]=NULL;} //函数名存
      放在$$->type_id
      ;
VarList: ParamDec  {$$=mknode(1,PARAM_LIST,yylineno,$1);}
      | ParamDec COMMA  VarList  {$$=mknode(2,PARAM_LIST,yylineno,$1,$3);}
      ;
ParamDec: Specifier VarDec          {$$=mknode(2,PARAM_DEC,yylineno,$1,$2);}
      ;

CompSt: LC DefList StmList RC      {$$=mknode(2,COMP_STM,yylineno,$2,$3);}
      ;
StmList: {$$=NULL; }
      | Stmt StmList  {$$=mknode(2,STM_LIST,yylineno,$1,$2);}
      ;
Stmt:   Exp SEMI      {$$=mknode(1,EXP_STMT,yylineno,$1);}
      | CompSt        {$$=$1;} //复合语句结点直接最为语句结点, 不再生成新的
      结点
      | RETURN Exp SEMI  {$$=mknode(1,RETURN,yylineno,$2);}
      | IF LP Exp RP Stmt %prec LOWER_THEN_ELSE
      {$$=mknode(2,IF_THEN,yylineno,$3,$5);}
      | IF LP Exp RP Stmt ELSE Stmt
      {$$=mknode(3,IF_THEN_ELSE,yylineno,$3,$5,$7);}
      | WHILE LP Exp RP Stmt {$$=mknode(2,WHILE,yylineno,$3,$5);}
      | FOR LP Def Exp SEMI Exp RP Stmt {$$=mknode(4,FOR,yylineno,$3,$4,$6,$8);}
      ;
DefList: %prec ID {$$=NULL; }
      | Def DefList {$$=mknode(2,DEF_LIST,yylineno,$1,$2);}
      | error SEMI  {$$=NULL;}
      ;

```

```

Def:    Specifier DecList SEMI {$$=mknode(2,VAR_DEF,yylineno,$1,$2);}
        ;
DecList: Dec    {$$=mknode(1,DEC_LIST,yylineno,$1);}
        | Dec COMMA DecList {$$=mknode(2,DEC_LIST,yylineno,$1,$3);} //局部变量声
名
        ;
Dec:    VarDec    {$$=$1;}
        |
          VarDec          ASSIGNOP          Exp
{$$=mknode(2,ASSIGNOP,yylineno,$1,$3);strcpy($$->type_id,"ASSIGNOP");}
        ;
Exp:    Exp          ASSIGNOP          Exp
{$$=mknode(2,ASSIGNOP,yylineno,$1,$3);strcpy($$->type_id,"ASSIGNOP");} // $ 结 点
type_id 空置未用，正好存放运算符
        | Exp AND Exp    {$$=mknode(2,AND,yylineno,$1,$3);strcpy($$->type_id,"AND");}
        | Exp OR Exp     {$$=mknode(2,OR,yylineno,$1,$3);strcpy($$->type_id,"OR");}
        | Exp RELOP Exp {$$=mknode(2,RELOP,yylineno,$1,$3);strcpy($$->type_id,$2);} //
词法分析关系运算符自身值保存在$2 中
        | Exp PLUS Exp   {$$=mknode(2,PLUS,yylineno,$1,$3);strcpy($$->type_id,"PLUS");}
        |
          Exp          MINUS          Exp
{$$=mknode(2,MINUS,yylineno,$1,$3);strcpy($$->type_id,"MINUS");}
        | Exp STAR Exp   {$$=mknode(2,STAR,yylineno,$1,$3);strcpy($$->type_id,"STAR");}
        | Exp DIV Exp    {$$=mknode(2,DIV,yylineno,$1,$3);strcpy($$->type_id,"DIV");}
        | LP Exp RP      {$$=$2;}
        |
          MINUS          Exp          %prec          UMINUS
{$$=mknode(1,UMINUS,yylineno,$2);strcpy($$->type_id,"UMINUS");}
        | NOT Exp        {$$=mknode(1,NOT,yylineno,$2);strcpy($$->type_id,"NOT");}
        |
          DPLUS          Exp
{$$=mknode(1,DPLUS_AFTER,yylineno,$2);strcpy($$->type_id,"DPLUS_AFTER");} // 后 自
增
        |
          Exp          DPLUS
{$$=mknode(1,DPLUS_BEFORE,yylineno,$1);strcpy($$->type_id,"DPLUS_BEFORE");} // 前
自增
        |
          DMINUS          Exp
{$$=mknode(1,DMINUS_AFTER,yylineno,$2);strcpy($$->type_id,"DMINUS_AFTER");} //
后自减
        |
          Exp          DMINUS
{$$=mknode(1,DMINUS_BEFORE,yylineno,$1);strcpy($$->type_id,"DMINUS_BEFORE");}
//前自减
        | ID LP Args RP {$$=mknode(1,FUNC_CALL,yylineno,$3);strcpy($$->type_id,$1);}
        | ID LP RP      {$$=mknode(0,FUNC_CALL,yylineno);strcpy($$->type_id,$1);}
        | ID            {$$=mknode(0,ID,yylineno);strcpy($$->type_id,$1);}
        | INT           {$$=mknode(0,INT,yylineno);$$->type_int=$1;$$->type=INT;}
        |
          FLOAT
{$$=mknode(0,FLOAT,yylineno);$$->type_float=$1;$$->type=FLOAT;}

```

			CHAR
{	\$\$=mknode(0,CHAR,yylineno);	\$\$->type_char=\$1;	\$\$->type=CHAR;}
	BREAK	{	\$\$=mknode(0,BREAK,yylineno);
	CONTINUE	{	\$\$=mknode(0,CONTINUE,yylineno);
	LC ArrayListN RC	{	\$\$=mknode(1,ARRAY,yylineno,\$2);
;			//多维数组赋值
ArrayListN:	ArrayListN	COMMA	ArrayList1
{	\$\$=mknode(2,ARRAY_N,yylineno,\$1,\$3);		
	ArrayList1	{	\$\$=mknode(2,ARRAY_1,yylineno,\$1);}
	ArrayList	%prec	LCOMMA
{	\$\$=mknode(1,ARRAY_1,yylineno,\$1);		//一维数组
;			
ArrayList1:	LC ArrayList RC	{	\$\$=mknode(2,ARRAY,yylineno,\$2);}
;			
ArrayList:	ArrayList COMMA Exp	{	\$\$=mknode(2,ARRAY_LIST,yylineno,\$1,\$3);
			//匹配数组的赋值
	Exp	{	\$\$=mknode(1,ARRAY_LIST,yylineno,\$1);
			//用 type_id 来存储, 因为不
			确定元素是哪种类型的
			// { \$\$=NULL; }
;			
Args:	Exp COMMA Args	{	\$\$=mknode(2,ARGS,yylineno,\$1,\$3);}
	Exp	{	\$\$=mknode(1,ARGS,yylineno,\$1);}
;			

2.3 符号表结构定义

这里采用的是单表组织形式来实现符号表, 用一个符号栈来表示当前在作用域内的符号, 每当有一个新的符号出现, 则将其属性压栈到符号栈中, 当符号出作用域则立即将这个符号弹出符号栈。

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

2.4 错误类型码定义

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

语义分析阶段负责检查各种语义错误，主要包括：

- (1) 全局变量重复定义
- (2) 函数调用参数太少
- (3) 函数调用参数类型不匹配
- (4) 函数调用参数太多
- (5) 变量未定义
- (6) ID 是函数名，类型不匹配
- (7) 赋值语句需要左值
- (8) 函数未定义
- (9) 不是一个函数
- (10) 函数重复定义
- (11) 参数名重复定义
- (12) 局部变量重复定义
- (13) 局部变量重复定义（赋值语句中）
- (14) char 类型不支持算术运算
- (15) char 类型不支持逻辑运算
- (16) char 类型不支持比较大小
- (17) char 类型不可以加负号
- (18) 赋值语句类型不匹配
- (19) 函数返回值类型不匹配

2.5 中间代码结构定义

选用四元式作为中间代码的形式，各种定义如表 2.2 所示：

表 2.2：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

2.6 目标代码指令集则

选用 MIPS 作为对应的目标代码,在生成目标代码时,要完成寄存器的分配,为了降低实现的难度,选择朴素的寄存器分配算法。中间代码与目标代码的对应关系如表 2.3 所示:

表 2.3: 中间代码与目标代码的对应关系

中间代码节点类型	目标代码形式
LABEL	Label_name: #直接打印标号名
Return_op	lw \$t0,4(\$sp) #从栈中取出返回值 sw \$t0,12(\$fp) #将返回值存到活动记录指定位置 add \$t0,\$ra,\$zero #从 ra 寄存器取出本函数返回地址 lw \$ra,8(\$fp) #恢复调用者函数的返回值 addi \$sp,\$fp,8 #恢复 sp 的值 lw \$fp,4(\$fp) #恢复帧指针的值 jr \$t0 返回
GOTO_BREAK	J end_label_num #跳到语句块结束标记处
JUDGE	lw \$t0,4(\$sp) #将布尔值从栈中取出到 t0 寄存器 addi \$sp,\$sp,4 #将布尔值出栈 beqz \$t0 end_label #布尔值为假跳转
GOTO_JUDGE	J judge_label #跳转到语句块的判断标号处
ASSIGN_OP	lw \$t0,8(\$sp) #从栈中取出要赋的值 addi \$sp,\$sp,8 #将临时值出栈 sw \$t0, 变量偏移(\$fp) #将值存到要赋的变量
AND_NO	lw \$t1,8(\$sp) #取出操作数 1 lw \$t2,4(\$sp) #取出操作数 2 addi \$sp,\$sp,8 #将操作数出栈 and \$t0,\$t1,\$t2 #计算与的结果 sw \$t0,(\$sp) #结果压栈 addi \$sp,\$sp,-4
OR_NO	lw \$t1,8(\$sp) #取出操作数 1 lw \$t2,4(\$sp) #取出操作数 2 addi \$sp,\$sp,8 #将操作数出栈 or \$t0,\$t1,\$t2 #计算或的结果

	sw \$t0,(\$sp) #结果压栈 addi \$sp,\$sp,-4
GR_NO	lw \$t1,8(\$sp) #取出操作数 1 lw \$t2,4(\$sp) #取出操作数 2 addi \$sp,\$sp,8 #将操作数出栈 addi \$t3,\$zero,0 #将 t3 置零 addi \$t4,\$zero,1 #将 t4 置 1 sub \$t0,\$t1,\$t2 #操作数 1-操作数 2 的结果存 t0 bgtz \$t0, label_x #结果大于 0 跳转到 label_x sw \$t3,(\$sp) #将 0 压栈，表示布尔值为 false j label_y #跳到结束 label 处。 Label_x: sw \$t4,(\$sp) #将 1 压栈，表示布尔值为真 label_y: addi \$sp,\$sp,-4 #栈向低地址空间移动
LS_NO	lw \$t1,8(\$sp) #取出操作数 1 lw \$t2,4(\$sp) #取出操作数 2 addi \$sp,\$sp,8 #将操作数出栈 addi \$t3,\$zero,0 #将 t3 置零 addi \$t4,\$zero,1 #将 t4 置 1 sub \$t0,\$t1,\$t2 #操作数 1-操作数 2 的结果存 t0 bltz \$t0, label_x #结果小于 0 跳转到 label_x sw \$t3,(\$sp) #将 0 压栈，表示布尔值为 false j label_y #跳到结束 label 处。 Label_x: sw \$t4,(\$sp) #将 1 压栈，表示布尔值为真 label_y: addi \$sp,\$sp,-4 #栈向低地址空间移动
GE	lw \$t1,8(\$sp) #取出操作数 1 lw \$t2,4(\$sp) #取出操作数 2 addi \$sp,\$sp,8 #将操作数出栈 addi \$t3,\$zero,0 #将 t3 置零 addi \$t4,\$zero,1 #将 t4 置 1 sub \$t0,\$t1,\$t2 #操作数 1-操作数 2 的结果存 t0 bgez \$t0, label_x #结果大于等于 0 跳转到 label_x sw \$t3,(\$sp) #将 0 压栈，表示布尔值为 false j label_y #跳到结束 label 处。 Label_x: sw \$t4,(\$sp) #将 1 压栈，表示布尔值为真 label_y: addi \$sp,\$sp,-4 #栈向低地址空间移动
LE	lw \$t1,8(\$sp) #取出操作数 1 lw \$t2,4(\$sp) #取出操作数 2 addi \$sp,\$sp,8 #将操作数出栈

	addi \$t3,\$zero,0 #将 t3 置零 addi \$t4,\$zero,1 #将 t4 置 1 sub \$t0,\$t1,\$t2 #操作数 1-操作数 2 的结果存 t0 blez \$t0, label_x #结果大于 0 跳转到 label_x sw \$t3,(\$sp) #将 0 压栈，表示布尔值为 false j label_y #跳到结束 label 处。 Label_x: sw \$t4,(\$sp) #将 1 压栈，表示布尔值为真 label_y: addi \$sp,\$sp,-4 #栈向低地址空间移动
EQ_NO	lw \$t1,8(\$sp) #取出操作数 1 lw \$t2,4(\$sp) #取出操作数 2 addi \$sp,\$sp,8 #将操作数出栈 addi \$t3,\$zero,0 #将 t3 置零 addi \$t4,\$zero,1 #将 t4 置 1 beq \$t1, \$t2,label_x #结果大于 0 跳转到 label_x sw \$t3,(\$sp) #将 0 压栈，表示布尔值为 false j label_y #跳到结束 label 处。 Label_x: sw \$t4,(\$sp) #将 1 压栈，表示布尔值为真 label_y: addi \$sp,\$sp,-4 #栈向低地址空间移动
NE_NO	lw \$t1,8(\$sp) #取出操作数 1 lw \$t2,4(\$sp) #取出操作数 2 addi \$sp,\$sp,8 #将操作数出栈 addi \$t3,\$zero,0 #将 t3 置零 addi \$t4,\$zero,1 #将 t4 置 1 bne \$t1, \$t2,label_x #结果大于 0 跳转到 label_x sw \$t3,(\$sp) #将 0 压栈，表示布尔值为 false j label_y #跳到结束 label 处。 Label_x: sw \$t4,(\$sp) #将 1 压栈，表示布尔值为真 label_y: addi \$sp,\$sp,-4 #栈向低地址空间移动
ADD	lw \$t1,8(\$sp) #取操作数 1 lw \$t2,4(\$sp) #取操作数 2 addi \$sp,\$sp,8 #操作数出栈 add \$t0,\$t1,\$t2 #操作数相加存到 t0 中 sw \$t0,(\$sp) #计算结果压栈 addi \$sp,\$sp,-4
MINUS	lw \$t1,8(\$sp) #取操作数 1 lw \$t2,4(\$sp) #取操作数 2 addi \$sp,\$sp,8 #操作数出栈 sub \$t0,\$t1,\$t2 #操作数相减存到 t0 中

	sw \$t0,(\$sp) #计算结果压栈 addi \$sp,\$sp,-4
MUL	lw \$t1,8(\$sp) #取操作数 1 lw \$t2,4(\$sp) #取操作数 2 addi \$sp,\$sp,8 #操作数出栈 mul \$t0,\$t1,\$t2 #操作数相乘存到 t0 中 sw \$t0,(\$sp) #计算结果压栈 addi \$sp,\$sp,-4
DIV	lw \$t1,8(\$sp) #取操作数 1 lw \$t2,4(\$sp) #取操作数 2 addi \$sp,\$sp,8 #操作数出栈 div \$t1,\$t2 #操作数相除 mflo \$t0 #商存到 t0 sw \$t0,(\$sp) #计算结果压栈 addi \$sp,\$sp,-4
UMINUS_NO	lw \$t1,4(\$sp) #取操作数 addi \$sp,\$sp,4 #操作数出栈 sub \$t0,\$zero,\$t1 #计算负值 sw \$t0,(\$sp) #结果如栈 addi \$sp,\$sp,-4
NOT	lw \$t1,4(\$sp) #取操作数 addi \$sp,\$sp,4 #操作数出栈 beq \$t1,\$zero,label_x sw \$zero,(\$sp) #操作数是 1，取非后将 0 入栈 j label-y label_x: addi \$t1,\$zero,1 sw \$t1,(\$sp) #操作数是 0，将取非后的 1 入栈 label_y: addi \$sp,\$sp,-4
INT_NO	addi \$t0,\$zero,num #立即数存到 t0 sw \$t0,(\$sp) \$t0 压栈 addi \$sp,\$sp,-4
ID_NO	lw \$t0,-变量偏移(\$fp) #取出变量的值 sw \$t0,(\$sp) #变量值压栈 addi \$sp,\$sp,-4
FUN_PUBLIC	addi \$sp,\$sp,-12 #预留控制区空间 sw \$ra,8(\$sp) #保存当前函数返回地址 sw \$fp,4(\$sp) #保存当前帧地址
FUN_CALL	addi \$fp,\$sp,实参所占空间大小 #调整帧指针 addi \$sp,\$fp,-活动记录大小 #调整栈指针 jal x #跳转到函数 x

3 系统设计与实现

3.1 词法分析器

miniC 中的单词分为 6 类：关键字、标识符、运算符、常量、界符、注释。利用 flex 设计此法分析器，只需要根据每一类单词的特征写出其对应的正规式，flex 即可自动生成对应的词法分析程序。

要注意的是，关键字的规则要写在标识符的前面，因为 flex 是前面的规则先匹配，如果标识符的规则写在了前面，那么所有的关键字都会被识别为标识符。

(1) 关键字

关键字是需要完全匹配，书写关键字的正则表达式只需要列举即可

"int"	{strcpy(yyval.type_id, yytext);return TYPE;}
"float"	{strcpy(yyval.type_id, yytext);return TYPE;}
"char"	{strcpy(yyval.type_id, yytext);return TYPE;}
"return"	{return RETURN;}
"if"	{return IF;}
"else"	{return ELSE;}
"while"	{return WHILE;}
"for"	{return FOR;}

(2) 标识符

标识符是以字母开头的，字母数字。

id	[A-Za-z][A-Za-z0-9]*
{id}	{strcpy(yyval.type_id, yytext); return ID;}

(3) 常量

常量包括整形常量、浮点常量、字符常量以及字符串常量。

int	[0-9]+
float	([0-9]*\.[0-9]+) ([0-9]+\.)
char	(\[A-Za-z0-9\])
{int}	{yyval.type_int=atoi(yytext);return INT;}
{float}	{yyval.type_float=atof(yytext);return FLOAT;}
{char}	{strcpy(yyval.type_char,yytext);return CHAR;}
{string}	{strcpy(yyval.type_string,yytext);return STRING;}

(4) 界符

","	{return SEMI;}
","	{return COMMA;}
"("	{return LP;}

```

")"      {return RP;}
"["      {return LB;}
"]"      {return RB;}
"{"      {return LC;}
"}"      {return RC;}
[\n]     {yycolumn=1;}
[ \r\t]  {}

```

(5) 运算符

运算符包括算数运算符以及逻辑运算符等。

```

">|"<|">="|"<="|"=="|"!=" {strcpy(yyval.type_id, yytext);return RELOP;}
"="      {return ASSIGNOP;}
"+"      {return PLUS;}
"_"      {return MINUS;}
"++"     { return DPLUS;}
"--"     { return DMINUS;}
"*"      {return STAR;}
"/"      {return DIV;}
"&&"     {return AND;}
"||"     {return OR;}

```

(6) 注释

注释部分我没有完成

3.2 语法分析器

使用 bison 只需要写好文法规则并设计好规约动作就可以自动生成语法分析器。

```

program: ExtDefList      {printf("程序体: \n"); display($1,0); }      //显示语法树,语义分析
魔改 semantic_Analysis($1);
      ;
ExtDefList: {$$=NULL;}
          | ExtDef ExtDefList {$$=mknode(2,EXT_DEF_LIST,yylineno,$1,$2);} //每一个
          个 EXTDEFLIST 的结点, 其第 1 棵子树对应一个外部变量声明或函数
      ;
ExtDef:   Specifier ExtDecList SEMI      {$$=mknode(2,EXT_VAR_DEF,yylineno,$1,$2);}
//该结点对应一个外部变量声明
          |Specifier FuncDec CompSt      {$$=mknode(3,FUNC_DEF,yylineno,$1,$2,$3);}
//该结点对应一个函数定义
          | error SEMI      {$$=NULL;}
      ;
Specifier:                                     TYPE
{$$=mknode(0,TYPE,yylineno);strcpy($$->type_id,$1);if(!strcmp($1,"int"))
$$->type=INT;if(!strcmp($1,"float"))          $$->type=FLOAT;if(!strcmp($1,"char"))

```

```

$$->type=CHAR;}

;
ExtDecList: VarDec      {$$=$1;}      /*每一个 EXT_DECLIST 的结点,其第一棵子
树对应一个变量名(ID 类型的结点),第二棵子树对应剩下的外部变量名*/
          |              VarDec      COMMA      ExtDecList
{$$=mknode(2,EXT_DEC_LIST,yylineno,$1,$3);}

;
VarDec: ID      {$$=mknode(0,ID,yylineno);strcpy($$->type_id,$1);} //ID 结点,
标识符字符串存放结点的 type_id
      |              ID      DimensionList
      {$$=mknode(1,DIMENSION,yylineno,$2);strcpy($$->type_id,$1);} //数组声名
;
DimensionList: LB      INT      RB
      {$$=mknode(1,DIMENSION_LIST,yylineno,$2);$$->type_int=$2;} //匹配数组的 [?]
      |              LB      INT      RB      DimensionList
{$$=mknode(2,DIMENSION_LIST,yylineno,$2,$4);$$->type_int=$2;}

;
FuncDec: ID      LP      VarList      RP
{$$=mknode(1,FUNC_DEC,yylineno,$3);strcpy($$->type_id,$1);} // 函数名存放在
$$->type_id
      |ID      LP      RP
{$$=mknode(0,FUNC_DEC,yylineno);strcpy($$->type_id,$1);$$->ptr[0]=NULL;} //函数名存
放在$$->type_id

;
VarList: ParamDec  {$$=mknode(1,PARAM_LIST,yylineno,$1);}
      | ParamDec COMMA  VarList  {$$=mknode(2,PARAM_LIST,yylineno,$1,$3);}
;
ParamDec: Specifier VarDec      {$$=mknode(2,PARAM_DEC,yylineno,$1,$2);}
;

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

```



```

    | WHILE LP Exp RP Stmt {$$=mknode(2,WHILE,yylineno,$3,$5);}
    | FOR LP Def Exp SEMI Exp RP Stmt {$$=mknode(4,FOR,yylineno,$3,$4,$6,$8);}
    ;
DefList: %prec ID {$$=NULL; }
    | Def DefList {$$=mknode(2,DEF_LIST,yylineno,$1,$2);}
    | error SEMI {$$=NULL;}
    ;
Def: Specifier DecList SEMI {$$=mknode(2,VAR_DEF,yylineno,$1,$2);}
    ;
DecList: Dec {$$=mknode(1,DEC_LIST,yylineno,$1);}
    | Dec COMMA DecList {$$=mknode(2,DEC_LIST,yylineno,$1,$3);} //局部变量声
名
    ;
Dec: VarDec {$$=$1;}
    | VarDec ASSIGNOP Exp
    {$$=mknode(2,ASSIGNOP,yylineno,$1,$3);strcpy($$->type_id,"ASSIGNOP");}
    ;
Exp: Exp ASSIGNOP Exp
    {$$=mknode(2,ASSIGNOP,yylineno,$1,$3);strcpy($$->type_id,"ASSIGNOP");} // $ 结 点
type_id 空置未用，正好存放运算符
    | Exp AND Exp {$$=mknode(2,AND,yylineno,$1,$3);strcpy($$->type_id,"AND");}
    | Exp OR Exp {$$=mknode(2,OR,yylineno,$1,$3);strcpy($$->type_id,"OR");}
    | Exp RELOP Exp {$$=mknode(2,RELOP,yylineno,$1,$3);strcpy($$->type_id,$2);} //
词法分析关系运算符自身值保存在$2 中
    | Exp PLUS Exp {$$=mknode(2,PLUS,yylineno,$1,$3);strcpy($$->type_id,"PLUS");}
    | Exp MINUS Exp
    {$$=mknode(2,MINUS,yylineno,$1,$3);strcpy($$->type_id,"MINUS");}
    | Exp STAR Exp {$$=mknode(2,STAR,yylineno,$1,$3);strcpy($$->type_id,"STAR");}
    | Exp DIV Exp {$$=mknode(2,DIV,yylineno,$1,$3);strcpy($$->type_id,"DIV");}
    | LP Exp RP {$$=$2;}
    | MINUS Exp %prec UMINUS
    {$$=mknode(1,UMINUS,yylineno,$2);strcpy($$->type_id,"UMINUS");}
    | NOT Exp {$$=mknode(1,NOT,yylineno,$2);strcpy($$->type_id,"NOT");}
    | DPLUS Exp
    {$$=mknode(1,DPLUS_AFTER,yylineno,$2);strcpy($$->type_id,"DPLUS_AFTER");} // 后 自
增
    | Exp DPLUS
    {$$=mknode(1,DPLUS_BEFORE,yylineno,$1);strcpy($$->type_id,"DPLUS_BEFORE");} // 前
自增
    | DMINUS Exp
    {$$=mknode(1,DMINUS_AFTER,yylineno,$2);strcpy($$->type_id,"DMINUS_AFTER");} //
后自减
    | Exp DMINUS
    {$$=mknode(1,DMINUS_BEFORE,yylineno,$1);strcpy($$->type_id,"DMINUS_BEFORE");}

```

```

//前自减
    | ID LP Args RP { $$=mknode(1,FUNC_CALL,yylineno,$3);strcpy($$->type_id,$1);}
    | ID LP RP      { $$=mknode(0,FUNC_CALL,yylineno);strcpy($$->type_id,$1);}
    | ID            { $$=mknode(0,ID,yylineno);strcpy($$->type_id,$1);}
    | INT           { $$=mknode(0,INT,yylineno);$$->type_int=$1;$$->type=INT;}
    |                                                       FLOAT
{ $$=mknode(0,FLOAT,yylineno);$$->type_float=$1;$$->type=FLOAT;}
    |                                                       CHAR
{ $$=mknode(0,CHAR,yylineno);$$->type_char=$1;$$->type=CHAR;}
    | BREAK        { $$=mknode(0,BREAK,yylineno);$$->type=BREAK;}
    | CONTINUE      { $$=mknode(0,CONTINUE,yylineno);$$->type=CONTINUE;}
    | LC ArrayListN RC { $$=mknode(1,ARRAY,yylineno,$2);} //多维数组赋值
;
ArrayListN:   ArrayListN          COMMA          ArrayList1
{ $$=mknode(2,ARRAY_N,yylineno,$1,$3);}
    | ArrayList1 { $$=mknode(2,ARRAY_1,yylineno,$1);}
    | ArrayList  %prec              LCOMMA
{ $$=mknode(1,ARRAY_1,yylineno,$1);} //一维数组
;
ArrayList1:   LC ArrayList RC { $$=mknode(2,ARRAY,yylineno,$2);}
;
ArrayList: ArrayList COMMA Exp { $$=mknode(2,ARRAY_LIST,yylineno,$1,$3);} //匹配数组的赋值
    | Exp { $$=mknode(1,ARRAY_LIST,yylineno,$1);} //用 type_id 来存储，因为不确定元素是哪种类型的
    // { $$=NULL; }
;
Args:   Exp COMMA Args { $$=mknode(2,ARGS,yylineno,$1,$3);}
    | Exp { $$=mknode(1,ARGS,yylineno,$1);}
;

```

3.3 符号表管理

定义一个符号表中一个符号的结构体如下：

```

struct symbol
{
    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...
};
```

其中各个字段的含义如注释所示。

整个符号表的定义如下所示：

```
struct symboltable
{
    struct symbol symbols[MAXLENGTH];
    int index;
} symbolTable;
```

这里的符号表采用数组的形式来实现。每当插入一个新的结点时，就将 `index` 加一。

3.4 语义检查

语义分析这部分的一个非常重要的工作就是符号表的管理，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。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。

语义分析这部分完成的是静态语义分析，主要包括：

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

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

（3）名字的上下文相关性检查。名字的出现遵循作用域与可见性的前提下应该满足一定的上下文的相关性。如变量在使用前必须经过声明，如果是面向对象的语言，在外部不能访问私有变量等等。

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

变量的作用域：

在语义分析过程中，各个变量名有其对应的作用域，一个作用域内不允许名字重复，为此，通过一个全局变量 `LEV` 来管理，`LEV` 的初始值为 0。这样在处理

外部变量名，以及函数名时，对应符号的层号值都是 1；处理函数形式参数时，固定形参名在填写符号表时，层号为 1。由于 mini_C 中允许有复合语句，复合语句中可定义局部变量，函数体本身也是一个复合语句，这样在 AST 的遍历中，通过 LEV 的修改来管理不同的作用域。

(1) 每次遇到一个复合语句的结点 COM_STM，首先对 LEV 加 1，表示准备进入一个新的作用域，为了管理这个作用域中的变量，使用栈 symbol_scope_TX，记录该作用域变量在符号表中的起点位置，即将符号表 symbolTable 的栈顶位置 symbolTable.index 保存在栈 symbol_scope_TX 中。

(2) 每次要登记一个新的符号到符号表中时，首先在 symbolTable 中，从栈顶向栈底方向查层号为 LEV 的符号是否和当前待登记的符号重名，是则报重复定义错误，否则使用 LEV 作为层号将新的符号登记到符号表中。

(3) 每次遍历完一个复合语句的结点 COM_STM 的子树，准备回到其父结点时，这时该复合语句语义分析完成，需要从符号表中删除该复合语句的变量，方法是首先 symbol_scope_TX 退栈，取出该复合语句作用域的起点，再根据这个值修改 symbolTable.index，很简单地完成了符号表的符号删除操作。

(4) 符号表的查找操作，在 AST 的遍历过程中，当分析各种表达式，遇到变量的访问时，在 symbolTable 中，从栈顶向栈底方向查询是否有相同的符号定义，如果全部查询完后没有找到，就是该符号没有定义；如果相同符号在符号表中有多处定义，按查找的方向可知，符合就近优先的原则。如果查找到符号后，就进一步进行语义分析，如：(1) 函数调用时，根据函数名在符号表找到的是一个变量，不是函数，需要报错；(2) 函数调用时，根据函数名找到这个函数，需要判断参数个数、类型是否匹配；(3) 根据变量名找的是一个函数。等等，需要做出各种检查。

3.5 报错功能

编译程序的报错大体分为以下四个方面：

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

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

(3) 名字的上下文相关性检查。名字的出现遵循作用域与可见性的前提

下应该满足一定的上下文的相关性。如变量在使用前必须经过声明，如果是面向对象的语言，在外部不能访问私有变量等等。

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

3.6 中间代码生成

中间代码生成的关键人物就是通过遍历 AST，将各子树的中间代码链进行拼接。最终在 AST 的根节点得到能完整描述程序结构的 code 链表。基于该链表即可完成中间代码的输出以及目标代码的生成。

为了完成中间代码的生成，对于 AST 中的结点，需要考虑设置以下属性，在遍历过程中，根据翻译模式给出的计算方法完成属性的计算。

.place 记录该结点操作数在符号表中的位置序号，这里包括变量在符号表中的位置，以及每次完成了计算后，中间结果需要用一个临时变量保存，临时变量也需要登记到符号表中。另外由于使用复合语句，可以使作用域嵌套，不同的作用域中的变量可以同名，这是在 mini-c 中，和 C 语言一样采用就近优先的原则，但在中间语言中，没有复合语句区分层次，所以每次登记一个变量到符号表中时，会多增加一个别名 (alias) 的表项，通过别名实现数据的唯一性。翻译时，对变量的操作替换成对别名的操作，别名命名形式为 v+序号。生成临时变量时，命名形式为 temp+序号，在填符号表时，可以在符号名称这栏填写一个空串，临时变量名直接填写到别名这栏。

.type 一个结点表示数据时，记录该数据的类型，用于表达式的计算中。该属性也可用于语句，表示语句语义分析的正确性 (OK 或 ERROR)。

.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。另外对函数，利用这项保存活动记录的大小。

.width 记录一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，方便计算变量、临时变量在活动记录中偏移量，以及最后计算函数活动记录的大小。

.code 记录中间代码序列的起始位置，如采用链表表示中间代码序列，该属性就是一个链表的头指针。

.Etrue 和 **.Efalse** 在完成布尔表达式翻译时，表达式值为真假时要转移的程序位置 (标号的字符串形式)。

.Snext 该结点的语句序列执行完后，要转移或到的的程序位置 (标号的字符串形式)。

3.7 代码优化

没有实现代码优化功能模块。

3.6 汇编代码生成

这部分要完成将 TAC 指令序列转换成目标代码，目标代码选择为 MIPS 汇编指令。最终生成的 MIPS 汇编指令可以在 Mars4_5.jar 上运行。

4 系统测试与评价

4.1 测试用例

实验一测试代码如下所示：

```
int a,b,c;
float m,n;
char p,q;
int d[2];
int e[2][3];
int f[2][3][4];
int fibo(int a) {
    if (a==1 || a==2)
        return 1;
    return fibo(a-1)+fibo(a-2);
}
int main(){
    int g[5][6];
    int h[3]={1,2,3};
    int m[2][3]={{1,2,3},{4,5,6}};
    char pp,qq;
    int m,n,i;
    int z=0;
    pp='\';
    qq=".";
    m=read();
    i=1;

    i++;
```

```

i--;
++i;
--i;

i=++i;

for(int z=0,x=6;z<=10;z++) {
    write(z);
    break;
}
while(i<=m)
{
    n=fibo(i);
    write(n);
    i=i+1;
    continue;
}

d={2,3};
e={{2,3,4},{5,6,7}};

return 1;
}

```

实验二测试代码

```

int a;
float a;
int test1(int a,int b)
{
    return 1;
}
int test1()
{
    return 0;
}
int test2(int a,int a){
    return 0;
}
int main(){
    int i,j;
    float i,i=1;
    char k;
    test1(1);
    test1('a',1);
    test1(1,2,3);

    m;
}

```

```

    test1;

    3=2;

    test213123();

    j0;

    k=k+1;

    !k;

    k<2;

    -k;

    return 1.0;
}

```

实验三测试代码

```

int a,b,c;

int fibo(int a) {
    if (a==1 || a==2)
        return 1;
    return fibo(a-1)+fibo(a-2);
}

int main(){
    int m,n,i;

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

```


4.2 正确性测试

实验一的测试结果如图 4.1 所示：

输出了二元组，二元组之后是程序体

```
管理员: C:\Windows\System32\cmd.exe
Active code page: 65001

C:\Users\HUAWEI\Desktop\lab_1>parser test.c
1 [TYPE, int]
1 [ID, a]
1 [COMMA, ,]
1 [ID, b]
1 [COMMA, ,]
1 [ID, c]
1 [SEMI, ;]
2 [TYPE, float]
2 [ID, m]
2 [COMMA, ,]
2 [ID, n]
2 [SEMI, ;]
3 [TYPE, char]
3 [ID, p]
3 [COMMA, ,]
3 [ID, q]
3 [SEMI, ;]
4 [TYPE, int]
4 [ID, d]
4 [LB, []
4 [INT, 2]
4 [RB, ]]
4 [SEMI, ;]
5 [TYPE, int]
5 [ID, e]
5 [LB, []
5 [INT, 2]
5 [RB, ]]
5 [LB, []
5 [INT, 3]
5 [RB, ]]
5 [SEMI, ;]
6 [TYPE, int]
6 [ID, f]
6 [LB, []
6 [INT, 2]
6 [RB, ]]
6 [LB, []
6 [INT, 3]
6 [RB, ]]
6 [LB, []
6 [INT, 4]
6 [RB, ]]
6 [SEMI, ;]
```

二元组

```
管理员: C:\Windows\System32\cmd.exe
44 [RC,}]
44 [RC,}]
44 [SEMI,;]
46 [RETURN,return]
46 [INT,1]
46 [SEMI,;]
47 [RC,}]
程序体:
  外部变量定义: (1)
    类型: int
    变量名:
      ID: a
      ID: b
      ID: c
  外部变量定义: (2)
    类型: float
    变量名:
      ID: m
      ID: n
  外部变量定义: (3)
    类型: char
    变量名:
      ID: p
      ID: q
  外部变量定义: (4)
    类型: int
    变量名:
      ID: d
      维大小: 2
  外部变量定义: (5)
    类型: int
    变量名:
      ID: e
      维大小: 2
      维大小: 3
  外部变量定义: (6)
    类型: int
    变量名:
      ID: f
      维大小: 2
      维大小: 3
      维大小: 4
  函数定义: (11)
    类型: int
    函数名: fibo
    函数形参:
      类型: int, 参数名: a
```

图 4.1: 实验一测试结果

实验二的测试结果如图 4.2 所示:

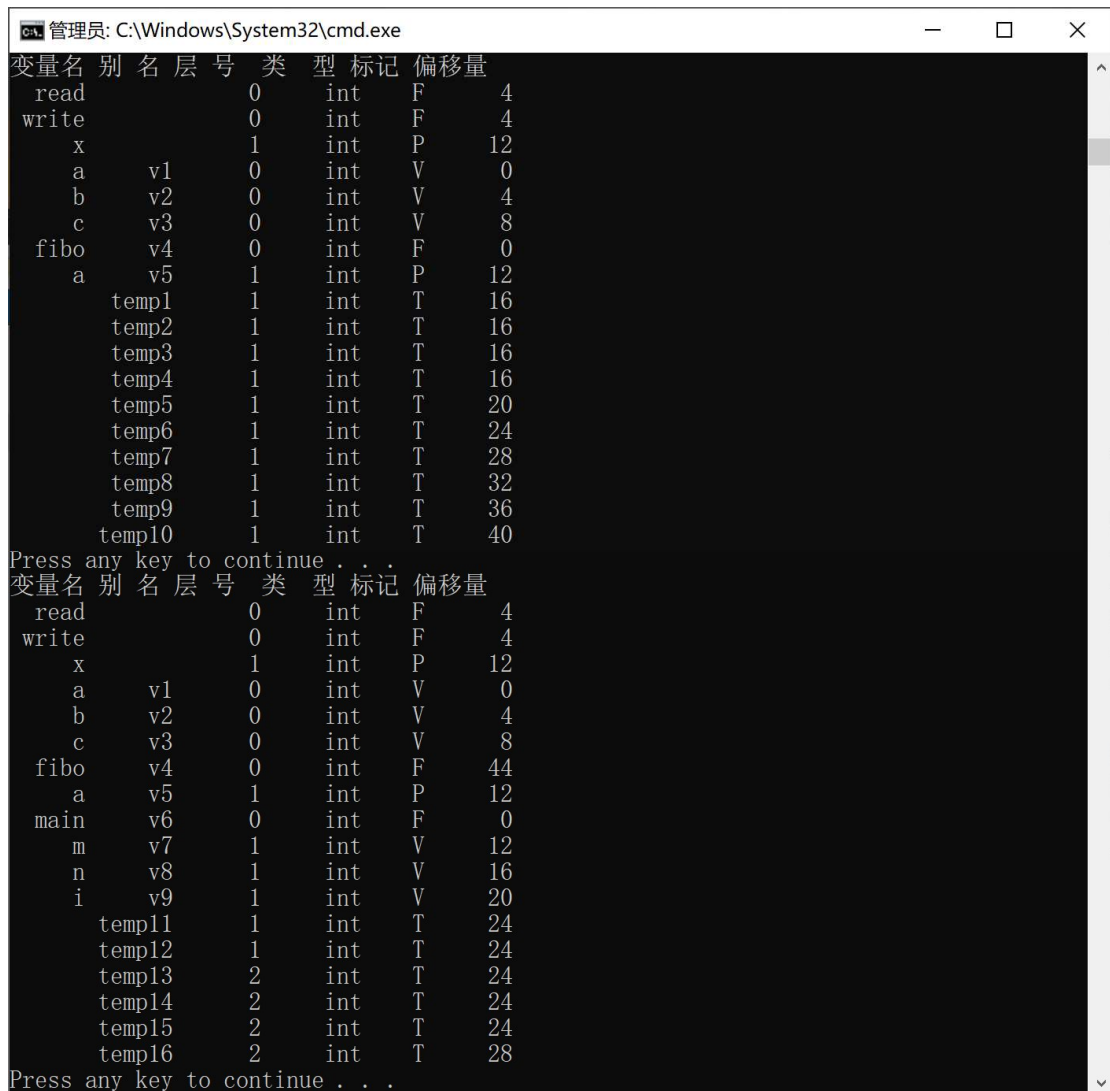
```

管理员: C:\Windows\System32\cmd.exe
FLA0T: 1.000000
在2行, a 全局变量重复定义
变量名 别名 层号 类型 标记 偏移量
  read          0    int    F      4
  write         0    int    F      4
    x           1    int    P     12
    a      v1    0    int    V      0
  test1      v3    0    int    F      0
    a      v4    1    int    P     12
    b      v5    1    int    P     16
      temp1     1    int    T     20
Press any key to continue . . .
在6行, test1 函数重复定义
变量名 别名 层号 类型 标记 偏移量
  read          0    int    F      4
  write         0    int    F      4
    x           1    int    P     12
    a      v1    0    int    V      0
  test1      v3    0    int    F     24
    a      v4    1    int    P     12
    b      v5    1    int    P     16
      temp2     1    int    T     12
Press any key to continue . . .
在9行, a 参数名重复定义
变量名 别名 层号 类型 标记 偏移量
  read          0    int    F     16
  write         0    int    F      4
    x           1    int    P     12
    a      v1    0    int    V      0
  test1      v3    0    int    F     24
    a      v4    1    int    P     12
    b      v5    1    int    P     16
  test2      v7    0    int    F      0
    a      v8    1    int    P     12
      temp3     1    int    T     20
Press any key to continue . . .
在14行, i 局部变量重复定义
在14行, i 局部变量重复定义(赋值)
在16行, 函数调用参数太少!
在17行, 参数类型不匹配
在18行, 函数调用参数太多!
在20行, m 变量未定义
在22行, test1 是函数名, 类型不匹配
在24行, 赋值语句需要左值
在26行, test213123 函数未定义
在28行, j 不是一个函数
在30行, k 是char类型, 不可以进行算术运算
在30行, 赋值语句左右类型不匹配
在32行, k 是char类型, 不可以进行逻辑运算
在34行, k 是char类型, 不可以比较大小
在36行, k 是char类型, 前面不可以加负号(取负操作)
在39行, 函数返回类型不匹配
变量名 别名 层号 类型 标记 偏移量
  read          0    int    F     16
  write         0    int    F      4
    x           1    int    P     12

```

图 4.2: 实验二测试结果

实验三测试结果如图 4.3 所示:

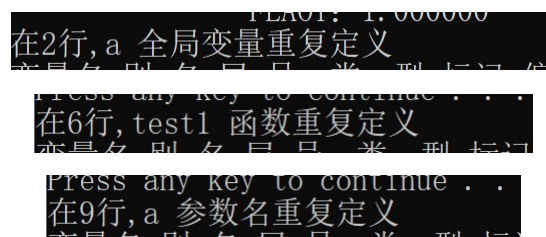


```
管理员: C:\Windows\System32\cmd.exe
变量名 别名 层号 类型 标记 偏移量
read      0    int  F    4
write     0    int  F    4
x         1    int  P   12
a        v1    0    int  V    0
b        v2    0    int  V    4
c        v3    0    int  V    8
fibo     v4    0    int  F    0
a        v5    1    int  P   12
temp1    1    int  T   16
temp2    1    int  T   16
temp3    1    int  T   16
temp4    1    int  T   16
temp5    1    int  T   20
temp6    1    int  T   24
temp7    1    int  T   28
temp8    1    int  T   32
temp9    1    int  T   36
temp10   1    int  T   40
Press any key to continue . . .
变量名 别名 层号 类型 标记 偏移量
read      0    int  F    4
write     0    int  F    4
x         1    int  P   12
a        v1    0    int  V    0
b        v2    0    int  V    4
c        v3    0    int  V    8
fibo     v4    0    int  F   44
a        v5    1    int  P   12
main     v6    0    int  F    0
m        v7    1    int  V   12
n        v8    1    int  V   16
i        v9    1    int  V   20
temp11   1    int  T   24
temp12   1    int  T   24
temp13   2    int  T   24
temp14   2    int  T   24
temp15   2    int  T   24
temp16   2    int  T   28
Press any key to continue . . .
```

图 4.3: 实验三测试结果

4.3 报错功能测试

实验二的测试代码中添加了几个错误，在测试代码中已经用注释给出。错误的报错情况如图 4.4 所示:



```
ERROR: 1.000000
在2行, a 全局变量重复定义
Press any key to continue . . .
在6行, test1 函数重复定义
Press any key to continue . . .
在9行, a 参数名重复定义
```

```
Press any key to continue . . .
在14行, i 局部变量重复定义
在14行, i 局部变量重复定义(赋值)
在16行, 函数调用参数太少!
在17行, 参数类型不匹配
在18行, 函数调用参数太多!
在20行, m 变量未定义
在22行, test1 是函数名, 类型不匹配
在24行, 赋值语句需要左值
在26行, test213123 函数未定义
在28行, j 不是一个函数
在30行, k 是char类型, 不可以进行算术运算
在30行, 赋值语句左右类型不匹配
在32行, k 是char类型, 不可以进行逻辑运算
在34行, k 是char类型, 不可以比较大小
在36行, k 是char类型, 前面不可以加负号(取负操作)
在39行, 函数返回类型不匹配
```

图 4.4: 报错

4.4 系统的优点

系统在 miniC 的基础上, 还增加了 `char` 类型、自增自减、多维数组、`for` 循环等多种功能。能够检测多种错误并给出正确的提示信息。能够清晰并且详细的展示编译过程中每一个步骤的结果。

4.5 系统的缺点

没有实现 `string` 类型、结构体、联合等高级的数据结构和操作。

5 实验小结或体会

编译原理算是这个学期最具有挑战性的一门学科了吧，在四次实验任务的层层递进下，最终实现了一个简易的编译器，虽然它支持的功能还很少，但是在这实验操作的过程中我对词法分析、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。

在这个过程中，会不断的遇到大大小小的问题，通过不断的查阅资料，复习课本上的知识，一个个解决问题，我学到了很多知识，。同时也锻炼了我的编码能力。编译原理的知识是整个学科中的核心之一，通过学习这门学科我对高级代码语言有了更深刻的了解，甚至看到后可以说出这是怎么实现的，我感觉很有用。

最后，感谢指导老师徐丽萍老师，感谢《编译原理》课程组，在这次实验中给予我很大帮助。今后我会带着收获的知识与教诲，更加刻苦努力的学习。

参考文献

- [1] 吕映芝等. 编译原理(第二版). 北京: 清华大学出版社, 2005
- [2] 胡伦俊等. 编译原理(第二版). 北京: 电子工业出版社, 2005
- [3] 王元珍等. 80X86 汇编语言程序设计. 武汉: 华中科技大学出版社, 2005
- [4] 王雷等. 编译原理课程设计. 北京: 机械工业出版社, 2005
- [5] 曹计昌等. C 语言程序设计. 北京: 科学出版社, 2008