華中科技大學

编译技术课程设计

院　　系　软 件 学 院

专业班级　软工 1803

姓　　名　王粟鹏

学　　号　U201817044

指导教师　胡雯蔷、徐丽萍、祝建华

2020 年　12 月　3 日

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

该编译程序目的是将 MINI-C 程序编译为 MIPS 汇编指令。

在语言设计部分，首先将整个程序划分为全局变量部分和函数部分。在全局变量部分可以识别 int、char、string 和数组，函数声明部分又分为形参部分和函数体部分，在函数体内部又分为变量申明和操作语句部分。

2.2 单词文法与语言文法

2.2.1 单词文法

注释：
comment (\/\/.*\n)|(\/*.**\/)
标识符：
id [A-Za-z][A-Za-z0-9]*
常量：
int [0-9]+
char '[^']'
float ([0-9]*\.[0-9]+)|([0-9]+\.)
string \"(\\.|[^"\\])*\"
关键字：
"int" "float" "char" "string" "return" "if""else""while" "for" "break" "continue"
界符：
";"
运算符：
"," ">"|"<"|">="|"<="|"=="|"!=""=" "++" "+" "--" "-" "*" "/" "&&" "||"
"!" "(" ")" "{" "}" "[" "]" [\n] [\r\t]

2.2.2 语言文法

```
program: ExtDefList;                          //整个程序的开始
ExtDefList: {$$=NULL;}                        //全局定义部分
        | ExtDef   ExtDefList
        ;
ExtDef:   Specifier ExtDecList SEMI           //变量声明
        |Specifier FuncDec CompSt             //函数声明
        | error SEMI   {$$=NULL;}
        ;                                     //对应一个变量或者函数的定义
Specifier:  TYPE
```

```
        ;                                   //变量的类型
ExtDecList:  VarDec
        | VarDec COMMA ExtDecList
        ;                                   //标识符
VarDec:     ID
         |Exp ArraySize
         ;                                  //一个标识符或者数组
ArraySize:  LB Exp RB ArraySize
         |LB Exp RB
         ;                                  //对应数组的大小
FuncDec: ID LP VarList RP
         |ID LP  RP
         ;                                  //函数名称及形参部分
VarList: ParamDec
         | ParamDec COMMA  VarList
         ;                                  //函数的形参
ParamDec: Specifier VarDec
        ;                                   //一个函数形参
CompSt: LC DefList StmList RC
        ;                                   //函数体
StmList: {$$=NULL; }
       | Stmt StmList
       ;                                    //函数操作语句序列
Stmt:   Exp SEMI
     | CompSt
     | RETURN Exp SEMI
     | IF LP Exp RP Stmt %prec LOWER_THEN_ELSE
     | IF LP Exp RP Stmt ELSE Stmt
     | WHILE LP Exp RP Stmt
     | FOR LP Exp SEMI Exp SEMI Exp RP Stmt
     ;                                      //操作部分
DefList: {$$=NULL; }
       | Def DefList
       | error SEMI
       ;                                    //函数内部变量声明
Def:    Specifier DecList SEMI
       ;                                    //一个变量的声明
DecList: Dec
      | Dec COMMA DecList
       ;                                    //符号序列
Dec:    VarDec
      | VarDec ASSIGNOP Exp
      ;
Exp:    Exp ASSIGNOP Exp
```

```
    | Exp AND Exp
    | Exp OR Exp
    | Exp RELOP Exp
    | Exp PLUS Exp
    | Exp MINUS Exp
    | Exp STAR Exp
    | Exp DIV Exp
    | LP Exp RP
    | MINUS Exp %prec UMINUS
    | NOT Exp
    | DPLUS  Exp
    | Exp DPLUS
    | DMINUS  Exp
    | Exp DMINUS
    | ID LP Args RP
    | ID LP RP
    | ID
    | INT
    | FLOAT
    | CHAR
    | STRING
    | BREAK
    | CONTINUE
    ;                                   //操作语句
Args:   Exp COMMA Args
     | Exp
     ;
```

2.3 符号表结构定义

index	name	level	type	flag	param_num

index：该符号的序号
name：该符号的名称
level：该符号的作用域
type：该符号的数据类型或者函数返回值类型
flag：符号的类型　‘F’:函数 ‘V’:变量 ‘P’:参数 ‘T’:临时变量 ‘A’:数组
param_num：函数形参的个数

2.4 错误类型码定义

1、函数传入参数个数少于规定数量
2、函数传入参数个数多于规定数量
3、参数类型不匹配
4、函数未定义
5、变量误用，非函数变量用作函数形式
6、引用未声明的变量
7、把函数用作普通变量
8、赋值语句没有左值
9、变量重复定义
10、函数名重复定义
11、参数重复定义
12、变量重复定义
13、返回类型错误
14、变量命名格式不符合规定

2.5 中间代码结构定义

中间代码格式采用三地址 TAC 方式，具体的中间代码结构形式定义如下：

语法	描述	OP	OPN1	OPN2	RESULT
LEBAL X	定义标号 X	LABEL			
FUNCTION f	定义函数 f	FUNCITION			
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	除法操作	DIV	y	z	x
x:=y*z	乘法操作	STAR	y	z	x
GOTO x	无条件转移操作	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	传入参数	ARG			x
x:=CALL f	调用函数	CALL	f		x
PARAM x	函数形参	PARAM			x
READ x	读入	READ			x
WRITE x	打印	WRITE			x

2.6 目标代码指令集选择

目标语言选择 MIPS32 指令序列，可以在 QtSpim 上运行指令集。在将 TAC 中间代码指令生成为目标代码 MIPS 指令时，其对应关系如下图所示，其中 reg(x)表示为变量 x 分配的寄存器。

中间代码	MIPS32 指令
LEBAL X	x;
x:=#k	li reg(x),k
x:=y	mov 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	div reg(y),reg(z) mflo reg(x)
x:=y*z	mul reg(x),reg(y),reg(z)
GOTO x	j x
x:=CALL f	jal f mov reg(x),$s0
RETURN x	mov $s0,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

3 系统设计与实现

3.1 词法分析器

在 lex.l 文件中实现词法分析，当识别出一个单词时，输出该单词的内容，并输出该电磁所属于的单词种类。单词文法定义已经在上一部分给出。使用 yylineno 存储该单词的行号，使用 yytext 来存储当前进行词法分析的单词，对于某一类的单词，当词法分析器识别出该单词时，即输出该单词所处的行号、单词的类型、单词的内容，识别之后词法分析器还会向语法分析器中传递每个单词的类型以及内容信息。

```
{int}        {printf("columm=%d   int 型常量 %s \n",yylineno,yytext);
             yylval.type_int=atoi(yytext); return INT;}
{float}      {printf("columm=%d float 常量%s \n",yylineno,yytext);
             yylval.type_float = atof(yytext) ; return FLOAT;}
{char}       {printf("columm=%d   char 常量 %s \n",yylineno,yytext);
             strcpy (yylval.type_char , yytext ); return CHAR;}
{string}     {printf("columm=%d   string 常量 %s \n",yylineno,yytext);
             strcpy(yylval.type_string,  yytext); return STRING;}
{comment}    {
             printf("this is comment.\n");
             for(int i=0;i<yyleng;i++){
                  if(yytext[i]!="\n")
                       printf("%c",yytext[i]);
                  else
                       break;
             }
             printf("\n");
             }
"int"        {printf("columm=%d   标示符    %s \n",yylineno,yytext);
             strcpy(yylval.type_id,  yytext);return TYPE;}
"float"      {printf("columm=%d   标示符    %s \n",yylineno,yytext);
             strcpy(yylval.type_id,  yytext);return TYPE;}
"char"       {printf("columm=%d   标示符    %s \n",yylineno,yytext);
             strcpy(yylval.type_id,  yytext);return TYPE;}
"string"     {printf("columm=%d   标示符    %s \n",yylineno,yytext);
             strcpy(yylval.type_id,  yytext);return TYPE;}
"return"     {printf("columm=%d   关键字    %s \n",yylineno,yytext);return RETURN;}
"if"         {printf("columm=%d   关键字    %s \n",yylineno,yytext);return IF;}
"else"       {printf("columm=%d   关键字    %s \n",yylineno,yytext);return ELSE;}
"while"      {printf("columm=%d   关键字    %s \n",yylineno,yytext);return WHILE;}
"for"        {printf("columm=%d   关键字    %s \n",yylineno,yytext);return FOR;}
"break"      {printf("columm=%d   关键字    %s \n",yylineno,yytext);return BREAK;}
"continue"   {printf("columm=%d   关键字    %s \n",yylineno,yytext);return CONTINUE;}
```

```
{id}            {printf("columm=%d  变量或函数名称   %s \n",yylineno,yytext);
                strcpy(yylval.type_id,  yytext); return ID;
{int}{id}       {printf("columm=%d  变量命名错误   %s \n",yylineno,yytext);
                strcpy(yylval.type_id,  yytext);}
";"             {printf("columm=%d  分号%s \n",yylineno,yytext);return SEMI;}
","             {printf("columm=%d  逗号%s \n",yylineno,yytext);return COMMA;}
">"|"<"|">="|"<="|"=="|"!="
                {printf("columm=%d  逻辑运算符%s \n",yylineno,yytext);
                strcpy(yylval.type_id, yytext);;return RELOP;}
"="             {printf("columm=%d  运算符   %s \n",yylineno,yytext);return ASSIGNOP;}
"++"            {printf("columm=%d  运算符   %s \n",yylineno,yytext);return DPLUS;}
"+"             {printf("columm=%d  运算符   %s \n",yylineno,yytext);return PLUS;}
"--"            {printf("columm=%d  运算符   %s \n",yylineno,yytext);return DMINUS;}
"-"             {printf("columm=%d  运算符   %s \n",yylineno,yytext);return MINUS;}
"*"             {printf("columm=%d  运算符   %s \n",yylineno,yytext);return STAR;}
"/"             {printf("columm=%d  运算符   %s \n",yylineno,yytext);return DIV;}
"&&"            {printf("columm=%d  运算符   %s \n",yylineno,yytext);return AND;}
"||"            {printf("columm=%d  运算符   %s \n",yylineno,yytext);return OR;}
"!"             {printf("columm=%d  运算符   %s \n",yylineno,yytext);return NOT;}
"("             {printf("columm=%d  LP %s \n",yylineno,yytext);return LP;}
")"             {printf("columm=%d  RP %s \n",yylineno,yytext);return RP;}
"{"             {printf("columm=%d  LC %s \n",yylineno,yytext);return LC;}
"}"             {printf("columm=%d  RC %s \n",yylineno,yytext);return RC;}
"["             {printf("columm=%d  LB %s \n",yylineno,yytext);return LB;}
"]"             {printf("columm=%d  RB %s \n",yylineno,yytext);return RB;}
[\n]            {yycolumn=1;}
[ \r\t]         {}
.           {printf("Error type A :Mysterious character \"%s\"\n\t at Line %d\n",yytext,yylineno);}
```

3.2 语法分析器

该部分的主要功能是生成语法分析树，语法分析树的结构定义如下：

```
struct ASTNode {
   int kind;
   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;                    //由某个字符生成的叶节点
       char type_string[100];             //由字符串生成的叶结点
       struct Array *type_array;
       struct Struct *type_struct;
```

```
    };
  struct ASTNode *ptr[4];              //由 kind 确定有多少棵子树，存储该节点的所有子树
  int place;                           //存放（临时）变量在符号表的位置序号
  char Etrue[15],Efalse[15];           //对布尔表达式的翻译时，真假转移目标的标号
  char Snext[15];                      //结点对应语句 S 执行后的下一条语句位置标号
  struct codenode *code;               //该结点中间代码链表头指针
  int type;                            //用以标识表达式结点的类型
  int pos;                             //语法单位所在位置行号
  int offset;                          //偏移量
  int width;                           //占数据字节数
  int num;                             //计数器，可以用来统计形参个数
  };
```

语法分析树中存储着程序各个语法单元之间的关系，每一个节点存储着每个语法单元的内容、类型等信息。通过 mknode 函数来创建一个节点，mknode 函数定义如下：

```
struct ASTNode * mknode(int num,int kind,int pos,...){
  struct ASTNode *T=(struct ASTNode *)calloc(sizeof(struct ASTNode),1);  //分配空间
  int i=0;
  T->kind=kind;
  T->pos=pos;
  va_list pArgs;                       //用于获取不确定个数的参数
  va_start(pArgs, pos);                //pArgs 用来存储额外的参数信息
  for(i=0;i<num;i++)
    T->ptr[i]= va_arg(pArgs, struct ASTNode *);
  while (i<4) T->ptr[i++]=NULL;
  va_end(pArgs);
  return T;
}
```

mknode 函数接收四个参数。num 为该节点子树的数量，kind 为该节点的类型，pos 为该节点的行号，剩下的参数为子树的序列，类型为 ASTNode。mknode 函数返回值为一个存储着当前节点信息的 ASTNode 节点。

mknode()函数通过在语法分析过程中被调用，通过语法分析过程中传入的参数信息，创建一个个节点，把他们通过父子树的形式连接在一起。每一个语法推导之后{}中的内容即为语义操作，为每个语义单位创建一个一个节点，并将节点之间的关系传到其他节点中。语义分析中的各个部分如下所示

```
program: ExtDefList   { printf("程序开始： \n");display($1,0);printf("\n \n 中间代码：\n\n");
          DisplaySymbolTable($1);}
          ;
ExtDefList: {$$=NULL;}
      | ExtDef ExtDefList {$$=mknode(2,EXT_DEF_LIST,yylineno,$1,$2);}
      ;
ExtDef:   Specifier ExtDecList SEMI   {$$=mknode(2,EXT_VAR_DEF,yylineno,$1,$2);}
      |Specifier FuncDec CompSt        {$$=mknode(3,FUNC_DEF,yylineno,$1,$2,$3);}
      | error SEMI   {$$=NULL;}
```

```
        ;
Specifier:  TYPE    {$$=mknode(0,TYPE,yylineno);strcpy($$->type_id,$1);$$->type=(!strcmp($1
,"int")?INT:(!strcmp($1,"float")?FLOAT:CHAR));}
        ;
ExtDecList:  VarDec      {$$=$1;}
        | VarDec COMMA ExtDecList {$$=mknode(2,EXT_DEC_LIST,yylineno,$1,$3);}
        ;
VarDec:     ID          {$$=mknode(0,ID,yylineno);strcpy($$->type_id,$1);}
         |Exp ArraySize   {$$=mknode(2,ARRAY,yylineno,$1,$2);}
         ;
ArraySize:  LB Exp RB ArraySize   {$$=mknode(2,ARRAYSIZE,yylineno,$2,$4);}
         |LB Exp RB            {$$=mknode(1,ARRAYSIZE,yylineno,$2);}
         ;
FuncDec: ID LP VarList RP   {$$=mknode(1,FUNC_DEC,yylineno,$3);strcpy($$->type_id,$1);}
         |ID LP  RP   {$$=mknode(0,FUNC_DEC,yylineno);strcpy($$->type_id,$1);$$->ptr[0]=N
         ULL;}
         ;
VarList: ParamDec  {$$=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      {$$=$1;}
    | RETURN Exp SEMI   {$$=mknode(1,RETURN,yylineno,$2);}
    | IF LP Exp RP Stmt %prec LOWER_THEN_ELSE   {$$=mknode(2,IF_THEN,yylineno,$3,$
5);}
    | IF LP Exp RP Stmt ELSE Stmt   {$$=mknode(3,IF_THEN_ELSE,yylineno,$3,$5,$7);}
    | WHILE LP Exp RP Stmt {$$=mknode(2,WHILE,yylineno,$3,$5);}
    | FOR LP Exp SEMI Exp SEMI Exp RP Stmt {$$=mknode(4,FOR,yylineno,$3,$5,$7,$9);}
    ;
DefList: {$$=NULL; }
      | Def DefList {$$=mknode(2,DEF_LIST,yylineno,$1,$2);}
      | error SEMI   {$$=NULL;}
      ;
Def:    Specifier DecList SEMI {$$=mknode(2,VAR_DEF,yylineno,$1,$2);}
      ;
DecList: Dec  {$$=mknode(1,DEC_LIST,yylineno,$1);}
     | Dec COMMA DecList  {$$=mknode(2,DEC_LIST,yylineno,$1,$3);}
```

```
    ;
Dec:    VarDec   {$$=$1;}
    | VarDec ASSIGNOP Exp  {$$=mknode(2,ASSIGNOP,yylineno,$1,$3);strcpy($$->type_id,"
ASSIGNOP");}
    ;
Exp:    Exp ASSIGNOP Exp {$$=mknode(2,ASSIGNOP,yylineno,$1,$3);strcpy($$->type_id,"AS
SIGNOP");}
    | Exp AND Exp   {$$=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($$->type_id,$2);}
    | Exp PLUS Exp  {$$=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    {$$=mknode(1,FIRSTDPLUS,yylineno,$2);strcpy($$->type_id,"FIRSTDPL
US");}
    | Exp DPLUS     {$$=mknode(1,LASTDPLUS,yylineno,$1);strcpy($$->type_id,"LASTDPLU
S");}
    | Exp PLUS ASSIGNOP Exp {$$=mknode(2,PLUSASSIGNOP,$1,$4,yylineno);strcpy($$->ty
pe_id,"PLUSASSIGNOP");}
    | Exp MINUS ASSIGNOP Exp {$$=mknode(2,MINUSASSIGNOP,$1,$4,yylineno);strcpy($$
->type_id,"MINUSASSIGNOP");}
    | Exp STAR ASSIGNOP Exp {$$=mknode(2,STARASSIGNOP,$1,$4,yylineno);strcpy($$->ty
pe_id,"STARASSIGNOP");}
    | Exp DIV ASSIGNOP Exp {$$=mknode(2,DIVASSIGNOP,$1,$4,yylineno);strcpy($$->type_
id,"DIVASSIGNOP");}
    | DMINUS  Exp   {$$=mknode(1,FIRSTDMINUS,yylineno,$2);strcpy($$->type_id,"FIRSTD
MINUS");}
    | Exp DMINUS    {$$=mknode(1,LASTDMINUS,yylineno,$1);strcpy($$->type_id,"LASTD
MINUS");}
    | 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);}
    | Exp LB Exp RB {$$=mknode(2,EXP_ARRAY,$1,$3,yylineno);}
    | ID            {$$=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);$$->type_char=$1;}
    | STRING        {$$=mknode(0,STRING,yylineno);strcpy($$->type_string,$1);}
    | BREAK         {$$=mknode(0,BREAK,yylineno);}
    | CONTINUE      {$$=mknode(0,CONTINUE,yylineno);}
```

```
    ;

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

3.3 符号表管理

符号表项结构定义如下：

```
struct symbol {
   char name[33];          //变量或函数名
   int level;              //层号
   int type;               //变量类型或函数返回值类型
   int  paramnum;          //对函数适用，记录形式参数个数
   char alias[10];         //别名，为解决嵌套层次使用
   char flag;              //符号标记，函数：'F'  变量：'V'   参数：'P'  临时变量：'T'  数组：’A’
   char offset;            //外部变量和局部变量在其静态数据区或活动记录中的偏移量，
                           //或记录函数活动记录大小，目标代码生成时使用
   };
```

在头文件的定义中，定义了一个存储了符号表的数据结构：

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

在语义分析中进行的符号表操作，符号表项都会存储到该表中。

在语法分析创建语法树之后，通过调用函数 Semantic_Analysis()函数进行符号表管理，该函数通过传入的参数，对符号表进行添加、删除、语义检查等操作。该函数的定义部分如下：

该函数接收一个 ASTNode 类型的参数，该 T 节点即为要进行符号表操作的对象，type 为该节点的类型，level 为当前的作用域，flag 为该节点的数据性质。函数在执行过程中，根据 T->kind 的类型，进行不同的符号表操作，对于有子树的节点，再递归的调用该函数进行子树的符号表分析。

```
int Semantic_Analysis(struct ASTNode* T,int type,int level,char flag,int command){
   int type1,type2;
   int indexX=0;
   if(T){
      switch(T->kind){
            case EXT_DEF_LIST:     //外部声明列表
                  Semantic_Analysis(T->ptr[0],type,level,flag,command);
                  Semantic_Analysis(T->ptr[1],type,level,flag,command);
                  break;
            case EXT_VAR_DEF:      //外部定义声明
```

```
            flag='V';
            type=Semantic_Analysis(T->ptr[0],type,level,flag,command);
            Semantic_Analysis(T->ptr[1],type,level,flag,command);
            break;
        case ARRAY:
            Semantic_Analysis(T->ptr[0],type,level,'A',command);
            break;
        case TYPE:
            return T->type;
            break;
        case EXT_DEC_LIST:

            Semantic_Analysis(T->ptr[0],type,level,flag,command);
            Semantic_Analysis(T->ptr[1],type,level,flag,command);
            break;
        case ID:
            i=0;
            while(new_table.symbols[i].level!=level&&i<new_table.index)//转到相同作用域
                i++;
            if(command==0)
            {
              while(i<new_table.index)
              {
              if(strcmp(new_table.symbols[i].name,T->type_id)==0&&new_table.symbols[i].f
lag==flag)
              {
                  if(flag=='V')
                      printf("ERROR! 第 %d 行 : 相 同 全 局 变 量 名 称 %s\n", T->pos,
T->type_id);
                  else if(flag=='T')
                      printf("ERROR! 第 %d 行 : 相 同 局 部 变 量 名 称 %s\n", T->pos,
T->type_id);
                  else if(flag=='F')
                      printf("ERROR! 第%d 行:相同函数名称%s\n",T->pos,T->type_id);
                  else
                      printf("ERROR! Same function parameter %s at line %d\n",T->type_i
d,T->pos);
                  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;
                    printf("%d\t",new_table.index);
                    printf("%s\t",new_table.symbols[new_table.index].name);
                    printf("%d\t",new_table.symbols[new_table.index].level);
                    if(new_table.symbols[new_table.index].type==INT)
                        printf("int\t");
                    else if(new_table.symbols[new_table.index].type==FLOAT)
                        printf("float\t");
                    else
                        printf("char\t");
                    printf("%c\t",new_table.symbols[new_table.index].flag);

                    printf("\n");
                    new_table.index++;
                }
                else{
                    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%s 没有定义 \n",T->pos,T->type_id);
                    }
                }
                break;
            case FUNC_DEF:
                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);
                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:
            printf("\t\t***%s***\n",new_table.symbols[new_table.index-1-counter].name);
            printf("-------------------------------------------------\n");
            printf("%s\t%s\t%s\t%s\t%s\t%s\n","Index","Name","Level","Type","Flag","Param
_num");
            printf("-------------------------------------------------\n");
            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);   //分析局部定义部分
            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--;
            printf("\n\n");
            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);
```

```
        Semantic_Analysis(T->ptr[1],type,level,flag,command);
        break;
case EXP_STMT:
        Semantic_Analysis(T->ptr[0],type,level,flag,command);
        break;
case IF_THEN:
        Semantic_Analysis(T->ptr[0],type,level,flag,command);
        Semantic_Analysis(T->ptr[1],type,level,flag,command);
        break;
case WHILE:
        Semantic_Analysis(T->ptr[0],type,level,flag,command);
        Semantic_Analysis(T->ptr[1],type,level,flag,command);
        break;
case FOR:
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 DPLUS:
case BREAK:
case DMINUS:
    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 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("ERRO! 第%d 行:函数名%s 在符号表中的定义为变量
\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);
              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;
}
```

3.4 语义检查

语义检查的工作在符号表操作过程中进行，即在 Semantic_Analysis()函数中进行。

在程序创建一个新变量时，语义检查程序会遍历符号表，进行唯一性检查，查询该变量是否已经定义过，如果定义过则会报告错误，重复定义，如果没有定义过则会向符号表中添加该项；如果是调用一个变量，则会进行上下文相关性检查，查询是否已经定义过；在调用函数或者赋值等操作时，需要进行数据类型检测，会比较该当前数据的类型 type 和等式右边整体的数据类型 t。

```
case ID:
i=0;
while(new_table.symbols[i].level!=level&&i<new_table.index)//转到相同作用域
```

```
        i++;
if(command==0){
        while(i<new_table.index){
        if(strcmp(new_table.symbols[i].name,T->type_id)==0&&new_table.symbols[i].flag==flag){
                if(flag=='V')
                        printf("ERROR! 第%d 行:相同全局变量名称%s\n", T->pos, T->type_id);
                else if(flag=='T')
                        printf("ERROR! 第%d 行:相同局部变量名称%s\n", T->pos, T->type_id);
                else if(flag=='F')
                        printf("ERROR! 第%d 行:相同函数名称%s\n",T->pos,T->type_id);
                 else
                        printf("ERROR! Same function parameter %s at line %d\n",T->type_id,T->pos);
                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{
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'||n
ew_table.symbols[i].flag=='T')){
                return new_table.symbols[i].type;
                }
                i--;
        }
        if(i<0){
        printf("ERROR! 第%d 行:变量%s 没有定义 \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;

3.5 报错功能

报错功能同样在 Semantic_Analysisi()函数中执行，在进行语义检测时，如果遇到错误，会立刻输出错误信息。

例如：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);

3.6 中间代码生成

3.6.1 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）。我们需要通过查询符号表找到 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回 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]

3.6.2 语句翻译模式

mini-c 中的语句博爱阔表达式语句、复合语句、返回语句和循环语句。

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]

3.6.3 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]

3.6.4 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，并构造这些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]

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 $Args_1$	t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) arg_list = t1 + arg_list code2 = translate_Args($Args_1$, sym_table, arg_list) return code1 + code2

3.6.5 中间代码生成过程

1、定义 opn 结构体：（其包含类型、种类、层号、偏移量等信息）

```
struct opn{
   int kind;                    //标识联合成员的属性
   int type;                    //标识操作数的数据类型
   union {
      int    const_int;         //整常数值，立即数
      float  const_float;       //浮点常数值，立即数
      char   const_char;        //字符常数值，立即数
      char   id[33];            //变量或临时变量的别名或标号字符串
      char   const_string[100];
      };
   int level;                   //变量的层号，0 表示是全局变量，数据保存在静态数据区
   int offset;                  //偏移量，目标代码生成时用
   };
```

2、定义 codenode 结构体：（三地址 TAC 代码结点,采用单链表存放中间语言代码）

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

3、定义 ASTNode 结构体：

```
struct ASTNode {
   int kind;
   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;              //由某个字符生成的叶节点
       char type_string[100];       //由字符串生成的叶结点
       struct Array *type_array;
       struct Struct *type_struct;
       };
   struct ASTNode *ptr[4];          //由 kind 确定有多少棵子树
   int place;                       //存放（临时）变量在符号表的位置序号
   char Etrue[15],Efalse[15];       //对布尔表达式的翻译时，真假转移目标的标号
   char Snext[15];                  //结点对应语句 S 执行后的下一条语句位置标号
   struct codenode *code;           //该结点中间代码链表头指针
   int type;                        //用以标识表达式结点的类型
   int pos;                         //语法单位所在位置行号
   int offset;                      //偏移量
   int width;                       //占数据字节数
```

```
int num;                    //计数器，可以用来统计形参个数
};
```

其包含了许多信息：

place 记录了该节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 记录该节点表示的语法单元中，定义变量和临时变量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 记录在布尔表达式翻译时，表达式为真或假时，要转移的位置；Snext 记录该节点语句序列执行完后，要转移到的程序位置。

4、newAlias 函数：生成一个新的别名

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

5、newLabel 函数：生成一个新的标号

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

6、newTemp 函数：生成一个临时变量

```
char *newTemp()
{
   static int no = 1;
   char s[10];
   snprintf(s, 10, "%d", no++);
   return str_catch("temp", s);
}
```

8、genIR 函数：生成一条 TAC 代码的节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;
}
```

9、genLabel 函数：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;
}
```

10、genGoto 函数：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;
}
```

11、merge 函数：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;
}
```

12、prnIR 函数：打印中间代码

```
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 ASTNode *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
  }
}
```

13、处理函数

在中间代码生成过程中，会遇到一些特定的符号，特定的表达式运算，在这种情况下可以分别执行不同的操作。将这些分别处理的操作封装成为函数，在遇到相应的符号时调用相应的函数执行操作，在此不具体展示其中的内容。

```
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

无

3.8 汇编代码生成

3.8.1 寄存器分配算法

寄存器分配采用的是朴素寄存器分配算法。朴素寄存器分配算法的思想最简单，但也最低效，他将所有的变量或临时变量都放入内存中。如此一来，每翻译一条中间代码之前都需要吧要用到的变量先加载到寄存器中，得到该代码的计算结果之后又需要将结果写回内存。这种方法的确能够将中间代码翻译成可正常运行的目标代码，而且实现和调试都特别容易，不过它最大的问题是寄存器的利用率实在太低。它不尽闲置了 MIPS 提供的大部分通用寄存器，那些未被闲置的寄存器也没有对减少目标代码的方寸次数做出任何贡献。由于工程较小，暂时采用此类最简易的方式进行寄存器分配，具体实现代码
如下：
这是可分配的寄存集合：

```
string regs[] = {"t1","t2","t3","t4","t5","t6","t7","t8","t9","s0","s1","s2","s3","s4","s5","s6","s7"};
```

这是具体的分配算法：

```
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 "$"+table[temp_str];
else{
    vector<string> keys;
    for (auto & it : table)              //已经分配寄存器的变量 key
    keys.emplace_back(it.first);
    for (auto & key : keys)              //当遇到未分配寄存器的变量时，清空之前所有分配的
临时变量的映射关系
        if (key.find("temp")!=string::npos && find(variables.begin(),variables.end() ,key) ==
    variables.end()){
            reg_ok[table[key]] = 1;
            table.erase(key);
        }
for (const auto & reg : regs)        //对于所有寄存器
    if(reg_ok[reg] == 1){                //如果寄存器可用
        table[temp_str] = reg;           //将可用寄存器分配给该变量，映射关系存到 table 中
        reg_ok[reg] = 0;                     //寄存器 reg 设置为已用
        return "$"+reg;
    }
```

}
}

3.8.2 目标代码与 MIPS 指令对应关系

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) 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

对于以上中间代码对应的 MIPS 指令，使用 translate 函数进行翻译。核心思想是读取中

间代码程序，分割得到每一行，然后对每一行中的每一部分进行分别处理：

```
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\t
addi $sp,$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\t
lw $t1,8($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 " ";
}
```

4 系统测试与评价

5.1 测试用例

5.1.1、实验一二测试用例：

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

int main(){
int m,n,i;
k=2;
m=read();
i=1;
m=i+2-n;
s='a';      //it's ok
--i;
for(i=0;i<=2;i++)
{
if(i==1)
   m=2;
else
   break;}
while(i<=m){
   n=fibo(i);
   write(n);
   i=i+1;
   continue;
   break;
}
return 1;

}
```

5.1.2、实验三四测试用例

```
int fun(int j){
```

```
    return j-2;
}

int main(){
    int m,k=0;
    k=k+2;
    k=k*2;
    k=fun(k);
    if(k==0)
        m=2;
    return 1;
}
```

5.2 正确性测试

5.2.1、实验一词法分析结果：

```
columm=1    标示符    int
columm=1    变量命名错误    1a
columm=1    分号;
columm=2    标示符    float
columm=2    变量或函数名称    m
columm=2    逗号,
columm=2    变量或函数名称    n
columm=2    分号;
columm=3    标示符    char
columm=3    变量或函数名称    s
columm=3    分号;
columm=4    标示符    int
columm=4    变量或函数名称    fibo
columm=4    LP (
columm=4    标示符    int
columm=4    变量或函数名称    a
columm=4    RP )
columm=4    LC {
columm=5    关键字    if
columm=5    LP (
columm=5    变量或函数名称    a
columm=5    逻辑运算符==
columm=5    int 常量  1
columm=5    运算符    ||
columm=5    变量或函数名称    a
columm=5    逻辑运算符==
columm=5    int 常量  2
columm=5    RP )
columm=5    关键字    return
```

```
columm=5    int 常量  1
columm=5    分号;
columm=6    关键字    return
columm=6    变量或函数名称    fibo
columm=6    LP (
columm=6    变量或函数名称    a
columm=6    运算符    -
columm=6    int 常量  1
columm=6    RP )
columm=6    运算符    +
columm=6    变量或函数名称    fibo
columm=6    LP (
columm=6    变量或函数名称    a
columm=6    运算符    -
columm=6    int 常量  2
columm=6    RP )
columm=6    分号;
columm=7    RC }
columm=9    标示符    int
columm=9    变量或函数名称    main
columm=9    LP (
columm=9    RP )
columm=9    LC {
columm=10     标示符    int
columm=10     变量或函数名称    m
columm=10     逗号,
columm=10     变量或函数名称    n
columm=10     逗号,
columm=10     变量或函数名称    i
columm=10     分号;
columm=11     标示符    int
columm=11     变量命名错误    2a
columm=11     分号;
columm=12     标示符    string
columm=12     变量或函数名称    ss
columm=12     分号;
columm=13     变量或函数名称    ss
columm=13     运算符    =
columm=13     string 常量  "asdqwe"
columm=13     分号;
columm=14     变量或函数名称    m
columm=14     运算符    =
columm=14     变量或函数名称    read
columm=14     LP (
```

```
columm=14    RP )
columm=14    分号;
columm=15    变量或函数名称   i
columm=15    运算符   =
columm=15    int 常量  1
columm=15    分号;
columm=16    变量或函数名称   m
columm=16    运算符   =
columm=16    变量或函数名称   i
columm=16    运算符   +
columm=16    int 常量  2
columm=16    运算符   -
columm=16    变量或函数名称   n
columm=16    分号;
columm=17    变量或函数名称   s
columm=17    运算符   =
columm=17    char 常量  'a'
columm=17    分号;
this is comment.
//it's ok

columm=18    运算符   --
columm=18    变量或函数名称   i
columm=18    分号;
columm=19    关键字   for
columm=19    LP (
columm=19    变量或函数名称   i
columm=19    运算符   =
columm=19    int 常量  0
columm=19    分号;
columm=19    变量或函数名称   i
columm=19    逻辑运算符<=
columm=19    int 常量  2
columm=19    分号;
columm=19    变量或函数名称   i
columm=19    运算符   ++
columm=19    RP )
columm=20    LC {
columm=21    关键字   if
columm=21    LP (
columm=21    变量或函数名称   i
columm=21    逻辑运算符==
columm=21    int 常量  1
columm=21    RP )
```

```
columm=22    变量或函数名称    m
columm=22    运算符    =
columm=22    int 常量  2
columm=22    分号;
columm=23    关键字    else
columm=24    关键字    break
columm=24    分号;
columm=24    RC }
columm=25    关键字    while
columm=25    LP (
columm=25    变量或函数名称    i
columm=25    逻辑运算符<=
columm=25    变量或函数名称    m
columm=25    RP )
columm=25    LC {
columm=26    变量或函数名称    n
columm=26    运算符    =
columm=26    变量或函数名称    fibo
columm=26    LP (
columm=26    变量或函数名称    i
columm=26    RP )
columm=26    分号;
columm=27    变量或函数名称    write
columm=27    LP (
columm=27    变量或函数名称    n
columm=27    RP )
columm=27    分号;
columm=28    变量或函数名称    i
columm=28    运算符    =
columm=28    变量或函数名称    i
columm=28    运算符    +
columm=28    int 常量  1
columm=28    分号;
columm=29    关键字    continue
columm=29    分号;
columm=30    关键字    break
columm=30    分号;
columm=31    RC }
columm=32    关键字    return
columm=32    int 常量  1
columm=32    分号;
columm=34    RC }
```

5.2.2、实验一语法分析结果：

```
程序开始：
 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： s
 函数定义：(7)
   类型： int
   函数名：fibo
   函数形参：
      类型：int, 参数名：a
   复合语句：(7)
      复合语句的变量定义部分：
      复合语句的语句部分：
         条件语句(IF_THEN)：(6)
            条件：
               OR
                  ==
                     ID： a
                     INT：1
                  ==
                     ID： a
                     INT：2
            IF 子句：(6)
               返回语句：(5)
                  INT：1
         返回语句：(6)
            PLUS
               函数调用：(6)
                  函数名：fibo
                  第 1 个实际参数表达式：
                     MINUS
                        ID： a
                        INT：1

               函数调用：(6)
                  函数名：fibo
                  第 1 个实际参数表达式：
                     MINUS
                        ID： a
```

INT：2

函数定义：(34)
类型： int
函数名：main
无参函数
复合语句：(34)
复合语句的变量定义部分：
复合语句的语句部分：
表达式语句：(13)
ASSIGNOP
ID： ss
STRING： "asdqwe"
表达式语句：(14)
ASSIGNOP
ID： m
函数调用：(14)
函数名：read
表达式语句：(15)
ASSIGNOP
ID： i
INT：1
表达式语句：(16)
ASSIGNOP
ID： m
MINUS
PLUS
ID： i
INT：2
ID： n
表达式语句：(17)
ASSIGNOP
ID： s
CHAR: a
表达式语句：(18)
先自减 ID： i
循环语句 FOR：(24)
定义部分：
ASSIGNOP
ID： i
INT：0
循环条件：(24)
<=
ID： i

```
        INT：2
    运算部分：
        后自增 ID： i
    循环体：(24)
        复合语句：(24)
            复合语句的变量定义部分：
            复合语句的语句部分：
                条件语句(IF_THEN_ELSE)：(24)
                    条件：
                        ==
                            ID： i
                            INT：1
                    IF 子句：(24)
                        表达式语句：(22)
                            ASSIGNOP
                                ID： m
                                INT：2
                    ELSE 子句：(24)
                        表达式语句：(24)
                            BREAK
循环语句 WHILE：(31)
    循环条件：
        <=
            ID： i
            ID： m
    循环体：(31)
        复合语句：(31)
            复合语句的变量定义部分：
            复合语句的语句部分：
                表达式语句：(26)
                    ASSIGNOP
                        ID： n
                        函数调用：(26)
                            函数名：fibo
                            第 1 个实际参数表达式：
                                ID： i

                表达式语句：(27)
                    函数调用：(27)
                        函数名：write
                        第 1 个实际参数表达式：
                            ID： n

                表达式语句：(28)
```

```
                ASSIGNOP
                    ID： i
                    PLUS
                        ID： i
                        INT：1
            表达式语句：(29)
                CONTINUE
            表达式语句：(30)
                BREAK
    返回语句：(32)
        INT：1
```

5.2.3、实验二语义分析生成的符号表：

```
                ***符号表***
                ***Symbol Table***
-------------------------------------------------------------------
Index   Name    Level   Type    Flag    Param_num
-------------------------------------------------------------------
0       m       0       float   V
1       n       0       float   V
2       s       0       char    V
4       a       1       int     P
                ***fibo***
-------------------------------------------------------------------
Index   Name    Level   Type    Flag    Param_num
-------------------------------------------------------------------
ERROR! 第5行:变量a没有定义
ERROR! 第5行:变量a没有定义

0       m       0       float   V
1       n       0       float   V
2       s       0       char    V
3       fibo    0       int     F       1
4       a       1       int     P
-------------------------------------------------------------------
                ***main***
-------------------------------------------------------------------
Index   Name    Level   Type    Flag    Param_num
-------------------------------------------------------------------
6       m       1       int     T
7       i       1       int     T
ERROR! 第11行:相同局部变量名称m
ERROR! 第12行:变量k没有定义
ERROR! 第14 行：函数read未定义
```

```
                    ***¡***
-------------------------------------------------------------------
Index     Name      Level     Type      Flag      Param_num
-------------------------------------------------------------------
ERROR!第25行：函数调用的第2 个参数类型不匹配
ERROR!第25行：函数调用fibo参数个数不匹配
|
0         m         0         float     V
1         n         0         float     V
2         s         0         char      V
3         fibo      0         int       F         1
4         a         1         int       P
5         main      0         int       F         0
-------------------------------------------------------------------
```

5.2.4、实验三生成的中间代码：

```
FUNCTION fun :
PARAM var2
temp1 := #2
temp2 := var2 - temp1
RETURN temp2
LABEL label1 :
FUNCTION main :
temp3 := #0
var5 := temp3
temp4 := #2
temp5 := var5 + temp4
var5 := temp5
temp6 := #2
temp7 := var5 * temp6
var5 := temp7
ARG var5
temp8 := CALL fun
var5 := temp8
temp9 := #0
IF var5 == temp9 GOTO label7
GOTO label6
LABEL label7 :
temp10 := #2
var4 := temp10
LABEL label6 :
temp11 := #1
RETURN temp11
LABEL label2:
```

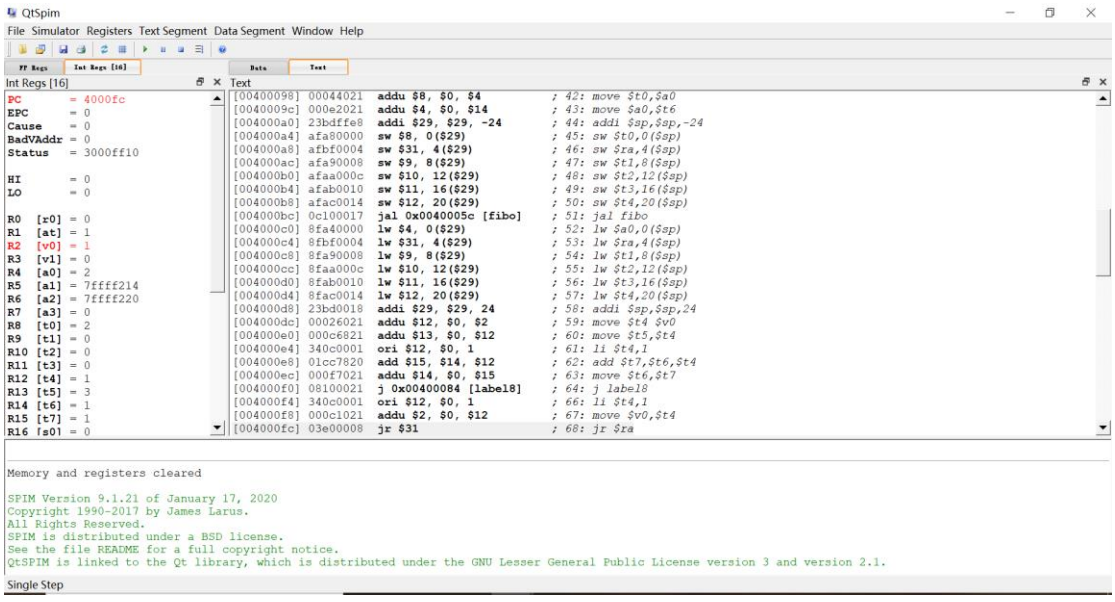
5.2.5、实验四生成的 MIPS 指令：

```
.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

print:
    li $v0,1
    syscall
    li $v0,4
    la $a0,_ret
    syscall
    move $v0,$0
    jr $ra
fibo:
    li $t1,0
    move $t2,$t1
    move $t1,$t3
    li $t4,1
    move $v0,$t4
    jr $ra
label1:
main:
    li $t4,2
    add $t5,$t6,$t4
    sub $t4,$t5,$t5
    move $t3,$t4
label5:
label8:
    ble $t6,$t3,label7
    j label6
label7:
    move $t0,$a0
    move $a0,$t5
    move $t0,$a0
    move $a0,$t6
```

```
        addi $sp,$sp,-24
        sw $t0,0($sp)
        sw $ra,4($sp)
        sw $t1,8($sp)
        sw $t2,12($sp)
        sw $t3,16($sp)
        sw $t4,20($sp)
        jal fibo
        lw $a0,0($sp)
        lw $ra,4($sp)
        lw $t1,8($sp)
        lw $t2,12($sp)
        lw $t3,16($sp)
        lw $t4,20($sp)
        addi $sp,$sp,24
        move $t4 $v0
        move $t5,$t4
        li $t4,1
        add $t7,$t6,$t4
        move $t6,$t7
        j label8
label6:
        li $t4,1
        move $v0,$t4
        jr $ra
label3:
```

5.2.6、目标代码执行过程：



5.3 报错功能测试

在实验二中，每一种类型的错误在测试用例中都体现了一次，输出结果如下：

columm=1　　变量命名错误　1a

ERROR! 第 5 行:变量 a 没有定义

ERROR! 第 11 行:相同局部变量名称 m

ERROR! 第 12 行:变量 k 没有定义

ERROR! 第 14 行：函数 read 未定义

ERROR!第 25 行：函数调用的第 2 个参数类型不匹配

ERROR!第 25 行：函数调用 fibo 参数个数不匹配

5.4　　系统的优点

系统采用朴素的寄存器分配法则，能够对短小的程序代码进行快速的寄存器分配，节省了中间代码向目标代码转换的时间，一定程度上提高了效率。

目标代码 MIPS 指令集指令精简，指令数目少，使用固定的指令格式，面向寄存器结构，采用硬布线控制逻辑，注重编译的优化，有利于减少指令执行周期，提高处理速度。

5.5　　系统的缺点

系统采用朴素的寄存器分配法则，没有使用堆栈，在当变量数量过多时，系统寄存器分配不够用，会出现程序容纳不下的状况。

没有使用代码优化，不利于代码效率的提升。

还有一些代码中的复杂功能，比如 switch 功能，没有进行中间代码生成，没有生成目标代码。还有待改进。

5 实验小结或体会

经历着四次编译原理实验，自己终于完成了一个小型的编译器，虽然其中还有很多的功能没有实现，但已经有了大体的框架，而且通过这次实验的联系，我对编译过程中的词法分析、语法分析、语义分析、中间代码生成和目标代码生成有了更深入的了解。对该实验的总结如下：

- 首先使用 lex 工具编写词法分析器，使用正则表达式识别单词，然后将识别出来的信息传递到语法分析工具中。
- 使用 bison 编写语法分析器，借助词法分析传递进来的单词，并根据语法分析的文法，实现整个程序的语法分析。
- 使用 mknode()函数实现语法树节点的创建，通过在语法分析过程中调用 mknode()函数来构建语法分析树，然后使用 display()函数来展示语法树。
- 在语法分析树建立的基础上，使用 Semantic_Analysis()函数创建符号表，并进行符号表管理，然后在符号表和语法树的基础上实现语义分析，识别语法错误并输出，最终输出符号表。
- 在语义分析过程中进行中间代码生成，提高编译效率。
- 利用生成的中间代码，生成最终的 MIPS32 指令，生成目标代码。

心得体会：

编译原理实验是一个比较难的实验，尤其是在刚开始着手做的时候，不知道从何下手，不知道从哪里开始着手整个编译器，不知道如何处理各个步骤之间的信息交流，但经过几节课的摸索之后，对整个编译器的原理有了更深的认识之后，就会觉得好做了许多，知道了如何实现各个部分的识别、如何实现不同文件之间的交流、知道了每一部分应该做怎样的处理，最终生成了自己的编译器。

在整个编译实验过程中，我对编译的词法分析、语法分析、语义分析、中间代码生成和目标代码生成有了更深的了解。纸上得来终觉浅，觉知此事要躬行。在学习课本知识时，觉得这部分很简单，但当自己真正着手做的时候，发现有好多问题要自己去处理，也正是在对这些问题的摸索中，才让我对知识有了更好的把握。

该实验还锻炼了我对 Linus 系统下 shell 终端的使用，使用命令行进行目录跳转、联合编译、运行可执行程序、输出重定向等等。

此次编译原理实验，不仅提高了我对编译过程的理解，巩固实践了知识，更提高了我的编程素养，让我获益匪浅。

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

注：源代码已在源代码部分提交