

# 华中科技大学

## 编译原理实验课程报告

**题目：MiniC 语言编译器设计与实现**

班 级\_\_\_\_\_信安 1805\_\_\_\_\_

姓 名\_\_\_\_\_李泽伟\_\_\_\_\_

学 号\_\_\_\_\_U201810470\_\_\_\_\_

指导教师\_\_\_\_\_刘铭老师\_\_\_\_\_

报告日期\_\_\_\_\_2020.11.20\_\_\_\_\_

网络空间安全学院

## 要 求

- 1、报告本人独立完成，内容真实，如发现抄袭，成绩无效；如果引用查阅的资料，需将资料出处列入参考文献，其格式请按华中科技大学本科毕业论文规范，并在正文中标注参考文献序号；
- 2、按编译原理实验内容，应包含：语言的文法、语言的词法及语法分析、语义分析、中间代码生成、目标代码生成；
- 3、应说明编译器编写调试时的系统环境、各编译器部分主要采用的设计思路、关键过程、设计的结果（主要源码文件功能、数据结构、函数说明）；同时，对设计实现中遇到的问题、解决过程进行记录和小节；并对完成的编译器过程进行总结，明确指出其优点、不足；
- 4、评分标准：4 个主要实验环节按任务书要求完成；采用的方法合适、设计合理；能体出分析问题、灵活运用知识解决实际问题的能力；报告条理清晰、语句通顺、格式规范；

格式 规范	词法 语法	语义 分析	中间 代码	目标 代码	总分
20	30	30	10	10	100

# 目 录

1 选题背景 .....	1
1.1 总体任务 .....	1
1.2 目标 .....	1
1.3 主要技术 .....	1
2 实验一 词法分析和语法分析 .....	2
2.1 语法的文法描述 .....	2
2.2 词法的文法描述 .....	3
2.3 语法分析器设计 .....	3
2.4 词法分析器设计 .....	7
2.5 词法及语法分析器运行截图 .....	8
2.6 小结 .....	12
3 实验二 语义分析 .....	13
3.1 语义子程序描述 .....	13
3.2 符号表的设计 .....	13
3.3 语义错误类型定义 .....	14
3.4 语义分析实现技术 .....	15
3.5 语义分析结果 .....	17
3.6 小结 .....	1
4 实验三 中间代码生成 .....	3
4.1 中间代码格式定义 .....	3
4.2 中间代码生成规则定义 .....	3
4.3 中间代码生成过程 .....	5
4.4 代码优化（可选） .....	7
4.5 中间代码生成结果 .....	7
4.6 小结 .....	11
5 实验四 目标代码生成 .....	12
5.1 指令集选择 .....	12
5.2 寄存器分配算法 .....	12
5.3 目标代码生成算法 .....	13
5.4 目标代码生成结果 .....	16
5.5 目标代码运行结果 .....	21
5.6 小结 .....	21
6 总结 .....	23
参考文献 .....	25

# 1 选题背景

## 1.1 总体任务

根据密码学编程时的某个函数（例如计算最大公约数函数）需要完成的程序功能，运用编译原理课程中的知识，设计一个类 C 语言程序设计语言相应的词法及语法，并实现该语言的编译器，将源码翻译为能在模拟器上运行的目标代码。

## 1.2 目标

通过资料查阅、分析设计、编程实现等步骤，逐步培养学生解决工程问题的能力，提高独立思考、灵活运用理论知识以解决实际问题的能力；

通过构造简化编译器的过程，了解编译器各部分的理论知识，找出理论与实践中的差异；灵活运用第三方工具协助解决工程问题；综合运用数据结构、算法、汇编语言、C 语言编程等前序课程的知识 and 技能。

## 1.3 主要技术

采用的具体开发环境、运行环境、完成编译器设计及实现中的关键技术；

## 2 实验一 词法分析和语法分析

实验一任务:

- 1、完成语言的语法描述和单词描述;完成语法规则、词法规则定义;
- 2、构造语法分析程序、词法分析程序;完成抽象语法树 AST 的设计;
- 3、构造合适的源程序样例,参考文献[1]1.1.6 必做内容;
- 4、将样例作为编译程序输入,输出其对应的抽象语法树;
- 5、根据参考文献[1],完成错误提示功能。

### 2.1 语法的文法描述

语法分析分为三个部分:声明部分,辅助定义部分,规则部分。

#### 声明部分

其中:%{到%}间的声明部分内容包含语法分析中需要的头文件包含,宏定义和全局变量的定义等,这部分会直接被复制到语法分析的 C 语言源程序中。

#### 辅助定义部分

在实验中要用到的几个主要内容有:

(1) 语义值的类型定义,mini-c 的文法中,有终结符,同时也有非终结符,每个符号(终结符和非终结符)都会有一个属性值,这个值的类型默认为整型。实际运用中,值得类型会有些差异,如 ID 的属性值类型是一个字符串,INT 的属性值类型是整型。在语法分析时,需要建立抽象语法树,这时 ExtDefList 的属性值类型会是树结点(结构类型)的指针。这样各种符号就会对应不同类型,这时可以用联合将这多种类型统一起来:%union

(2) 终结符定义,在 Flex 和 Bison 联合使用时,需要在 parser.y 中的%token 后面罗列出所有终结符(单词)的种类码标识符。

(3) 非终结符的属性值类型说明,对于非终结符,如果需要完成语义计算时,会涉及到非终结符的属性值类型,这个类型来源于(1)中联合的某个成员,可使用格式:%type <union 的成员名>非终结符。

(4) 优先级与结合性定义。对 Bison 文件进行翻译,得到语法分析程序的源程序时,通常会出现报错,大部分是移进和归约(shift/reduce),归约和归约(reduce/reduce)的冲突类的错误。为了改正这些错误,需要了解到底什么地方发生错误,这是,需要在翻译命令中,加上一个参数-v,即命令为:: bison -d -v parser.y 这时,会生成一个文件 parser.output。打开该文件,开始几行说明(LALR(1)分析法)哪几个状态有多少个冲突项,再根据这个说明中的状态序号去查看对应的状态进行分析、解决错误。

#### 规则部分

使用 Bison 采用的是 LR 分析法,需要在每条规则后给出相应的语义动作。例如对规则:Exp  $\rightarrow$  Exp =Exp, 在 parser.y 中为:

```
Exp: Exp ASSIGNOP Exp { $S=mknnode(ASSIGNOP,$1,$3,NULL,yylineno); }
```

同时在使用 Bison 的过程中，要完成报错和容错，使用 Bison 得到的语法分析程序，对 mini-c 程序进行编译时，一旦有语法错误，需要准确、及时的报错，这个由 yyerror 函数负责完成，需要补充的就是错误定位，在源程序的哪一行、哪一列有错。

## 2.2 词法的文法描述

词法分析分为三个部分：定义部分，规则部分，用户子程序定义部分。

第一个部分为定义部分，其中可以有一个%{到%}的区间部分，主要包含 c 语言的一些宏定义，如文件包含、宏名定义，以及一些变量和类型的定义和声明。会直接被复制到词法分析器源程序 lex.yy.c 中。%{到%}之外的部分是一些正规式宏名的定义，这些宏名在后面的规则部分会用到。

第二个部分为规则部分，一条规则的组成为：

正规表达式 动作

表示词法分析器一旦识别出正规表达式所对应的单词，就执行动作所对应的操作，返回单词的种类码。在这里可写代码显示（种类编码，单词的自身值），观察词法分析每次识别出来的单词，作为实验检查的依据。

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

第三个部分为用户子程序部分，这部分代码会原封不动的被复制到词法分析器源程序 lex.yy.c 中。

高级语言的词法分析器，需要识别的单词有五类：关键字（保留字）、运算符、界符、常量和标识符。依据 mini-c 语言的定义，在此给出各单词的种类码和相应符号说明：

INT→整型常量	RETURN →return
FLOAT→浮点型常量	IF→if
ID→标识符	ELSE →else
ASSIGNOP→=	WHILE → while
RELOP→> >= < <= == !=	SEMI → ;
PLUS→+	COMMA → ,
MINUS→ -	SEMI→ ;
STAR →*	LP→(
DIV→/	RP→)
AND →&&	LC→{
OR→   NOT →!	RC→}
TYPE →int   float	

## 2.3 语法分析器设计

语法的文法描述对应着的是文件 parser.y 的内容。

### 1).声明部分

声明部分，首先 **union** 部分是将各种符号对应的不同类型联合存放的地方，可以用联合将这多种类型统一起来：其中除了老师提供的 **int** 和 **float** 类型的变量，还加入了自己命名的变量 **type\_gowhere[100]**，其为自己定义的标识符，具体功能下文会提到。

```
%union {
    int type_int;
    float type_float;
    char type_gowhere[100];
    char type_id[32];
    struct node *ptr;
};
```

图 1-1 union 定义

接着用 **%type** 来指定非终结符的语义值类型，用 **<>** 选择 **union** 中某个类型，后面列出同类型的非终结符。

以 **%type<ptr> Program ExtDefList** 为例子，此语句意思为：非终结符 **ExtDefList** 属性值的类型对应联合中成员 **ptr** 的类型，在本实验中对应该一个树结点的指针。类似的，还有自己添加的功能 **gowhere**，对应着 **union** 的 **type\_gowhere**。

```
%token <type_int> INT //指定是type_int类型，用于AST树建立
%token <type_id> ID RELOP TYPE //指定是type_id 类型
%token <type_float> FLOAT //指定是type_float类型
%token <type_gowhere> GOWHERE //指定是type_gowhere类型
```

图 1-2 token 定义

接着定义运算符符号的优先级与结合性：其中 **YH** 为自己添加的功能，表示异或。

```
%token LP RP LC RC SEMI COMMA
%token PLUS MINUS STAR DIV ASSIGNOP AND OR NOT IF ELSE WHILE RETURN
%left ASSIGNOP
%left OR YH
%left AND
%left RELOP
%left PLUS MINUS
%left STAR DIV
%right UMINUS NOT
```

图 1-3 运算符符号定义

## 2).语法规则部分

接下来就是语法树的部分，大部分和指导书的一致，下面具体说明自己添加的功能部分：

(1) 在文法生成到 **TYPE** 的时候，原来的程序是仅对 **int** 和 **float** 类型进行判断，若不是 **int** 则是 **float** 类型，由于只添加一个自定义变量 **gowhere**，故类似的添加语句：

```
!strcmp($1,"gowhere"?GOWHERE:FLOAT
Spectfier: TYPE { $$=mknnode(TYPE,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);$$->type=!strcmp($1,"int"?INT:(!strcmp($1,"gowhere"?GOWHERE:FLOAT));}
```

图 1-4 自定义变量 **gowhere** 定义

(2)在文法推导到 EXP 的时候添加自己定义的异或符号 YH,并建立四元组节点,将“YH”放入新建立节点的 type\_id 中。

```
| Exp OR Exp    {$$=mknode(OR,$1,$3,NULL,yylineno);strcpy($$->type_id,"OR");}
| Exp YH Exp    {$$=mknode(YH,$1,$3,NULL,yylineno);strcpy($$->type_id,"YH");}
| Exp BELONG Exp {$$=mknode(BELONG,$1,$3,NULL,yylineno);strcpy($$->type_id,"BELONG");} //词法分析器
```

图 1-5 自定义运算符 YH 创建节点

同时在推导到类型符的时候,同样添加自定义的 gowhere 标识符。

```
| FLOAT    {$$=mknode(FLOAT,NULL,NULL,NULL,yylineno);$->type_float=$1;$->type=FLOAT;}
| GOWHERE  {$$=mknode(GOWHERE,NULL,NULL,NULL,yylineno);strcpy($$->type_gowhere,$1);$->type=GOWHERE;}
|
```

图 1-6 自定义变量 gowhere 创建节点

```
Exp:Exp YH Exp {$$=mknode(YH,$1,$3,NULL,yylineno);
strcpy($$->type_id,"YH");}
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。
```

### 3) 用户函数部分

此处与实验模板一致。

```
%%
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 Found at Line %d Column %d: ", yyloc.first_line,yyloc.first_column);
    vfprintf(stderr, fmt, ap);
    fprintf(stderr, ".\n");
}
|
```

图 1-7 用户函数部分

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,大部分与模板一致,以下只列出自己定义的部分:



```

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
        float type_float;    //由浮点常数生成的叶结点
        char type_gowhere[100];    //由ziji生成的叶结点
    };
};

```

图 1-8 node 部分

将 gowhere 的生成的叶结点加入 node 结构，以便后期调用的时候可以存储 gowhere 标识符的内容，要注意的是由于只分配了 100 的大小，故此表示符的长度最大只有 100。

其他的部分和模板一致，故不过多赘述。

## 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。

首先看 mknode 函数：

```

struct node * mknode(int kind,struct node *first,struct node *second, struct node *third,int position )
{
    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->position=position;
    return T;
}

```

图 1-9 抽象语法树部分

我们知道，本次实验的语法树建立的四元组，ptr 数组的三个值分别指代着 kind 确定的子树，Position 代表着语法单位所在位置行号。Kind 表示存放的结点类型。

我们在下面的 case 处添加我们自己定义的 GOWHERE 标识符：

```

-----
case GOWHERE:    printf("%cGOWHERE : %s\n",indent, ' ',T->type_gowhere);
    break;
-----

```

图 1-10 case 添加 gowhere 部分

同时由于 YH（异或）与 or 等类型一样，是左连接的运算符，故在其中添加 case YH，如下图所示：

```

case ASSIGNOP:
case AND:
case OR:
case YH:
case RELOP:
case PLUS:
case MINUS:
case STAR:
case DIV:
    printf("%c%s\n",indent, ' ',T->type_id);
    displayAST(T->ptr[0],indent+3);
    displayAST(T->ptr[1],indent+3);
    break;

```

图 1-11 case 添加 YH 部分

## 2.4 词法分析器设计

使用 flex 工具编写 lex.l 文件，对指定的高级语言程序进行词法分析。

### 1) 定义部分

根据实验参考书籍《编译原理实践与指导教程》、《Flex&Bison》，我们将语法单元定位的宏 YY\_USER\_ACTION 补全，如下图所示：

```
/*完成语法单元定位的宏YY_USER_ACTION，在每个动作之前被执行的代码，参考文献[1]31页*/  
/*扩充返回类型YYLVAL*/  
#define YY_USER_ACTION \  
yylloc.first_line=yylloc.last_line=yylineno; \  
yylloc.first_column=yycolumn; \  
yylloc.last_column=yycolumn+yyldeng-1; \  
yycolumn+=yyldeng ;
```

图 2-1 YY\_USER\_ACTION 补全

定义表示，在每一次用户动作后的执行如下语句：将 yylineno 赋予 yylloc.first\_line，yylloc.last\_line，表示现在在的行数，将行数的具体位置赋予 yylloc.first\_column，将长度加一。

接着添加自己定义的功能到 YYLVAL，此处要与头文件中的一致：

```
typedef union {  
    int type_int ;  
    float type_float ;  
    char type_gowhere[100];  
    char type_id[32];  
    struct node *ptr;|  
}YYLVAL;  
#define YYSTYPE YYLVAL  
%}
```

图 2-1 node 补全

### 2) 规则部分

添加自己定义功能 gowhere：以~~开头的字符串：

```
{gowhere}      {strcpy(yylval.type_gowhere, yytext); return GOWHERE;}
```

匹配到后的功能：

```
"gowhere"      {strcpy(yylval.type_id, yytext);return TYPE;}
```

异或符号（YH）^：

```
"^"  
                {return YH;}
```

```

%%
{int}      {yyval.type_int=atoi(yytext); return INT;}/*INT FLOAT TYPE...is defined by parser.y*/
{float}    {yyval.type_float=atof(yytext); return FLOAT;}
{gowhere}  {strcpy(yyval.type_gowhere, yytext); return GOWHERE;}

"int"      {strcpy(yyval.type_id, yytext);return TYPE;}
"float"    {strcpy(yyval.type_id, yytext);return TYPE;}/*copy zhizheng yyval.type_id zhong de shuju into
"gowhere"  {strcpy(yyval.type_id, yytext);return TYPE;}

"return"   {return RETURN;}
"if"       {return IF;}
"else"     {return ELSE;}
"while"    {return WHILE;}
{id}       {strcpy(yyval.type_id, yytext); return ID; }
";"        {return SEMI;}
","        {return COMMA;}
">"|"<"|>="|<="|"=="|"!=" {strcpy(yyval.type_id, yytext);return RELOP;}
"="        {return ASSIGNOP;}
"^"        {return YH;}
"+"        {return PLUS;}
"_"        {return MINUS;}

```

图 2-3 规则自定义补全

补全注释以及错误提示:

```

...
[ \r\t]    {}
.          {fprintf(stderr,"%4d bad include line %d \nbad string is %s\n",yycolumn,yylineno,yytext);}

\\\[^\n]*   {}
\\\[^\n]*   {}

```

图 2-4 注释补全

### 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。

由于共同中,main 冲突不能用了，故删除。

```

%%
int yywrap()
{
return 1;
}

```

图 2-5 函数补全

## 2.5 词法及语法分析器运行截图

(1)测试样例一代码

```

int a, b, c;
float m, n;
gowhere beijin;
int fibo (int a){
    if(a==1||a==2) return 1;
    return fibo (a-1)+fibo ( a-2);
}
int main()
{int m, n,i,h;
gowhere beijin;
m=read () ;
i=1;
h=i^i;
beijin=~a;|
while (i<=m)/*ihlll*/
{
n=fibo (i) ;write (n) ;i=i+l;}
return l;}

```

图 3-1 测试代码

可以看到自己添加的 gowhere 标识符号成功运行，YH 异或运算成功识别。

```

lzw@ubuntu: ~/Desktop/YH and gowhere
File Edit View Search Terminal Help
lzw@ubuntu:~/Desktop/YH and gowhere$ '/home/lzw/Desktop/YH and gowhere/parser' '
/home/lzw/Desktop/YH and gowhere/text.c'
外部变量定义：
  类型： int
  变量名：
    ID： a
    ID： b
    ID： c
外部变量定义：
  类型： float
  变量名：
    ID： m
    ID： n
外部变量定义：
  类型： gowhere
  变量名：
    ID： beijin
函数定义：
  类型： int
  函数名： fibo
  函数形参：
    类型： int, 参数名： a
复合语句：
  复合语句的变量定义：

```

图 3-2 运行截图

```
lzw@ubuntu: ~/Desktop/YH and gowhere
File Edit View Search Terminal Help
函数定义：
  类型：int
  函数名：main
  无参函数
复合语句：
  复合语句的变量定义：
    LOCAL VAR_NAME：
      类型：int
      VAR_NAME：
        m
        n
        i
        h
    LOCAL VAR_NAME：
      类型：gowhere
      VAR_NAME：
        beijin
  复合语句的语句部分：
    表达式语句：
      ASSIGNOP
      ID：m
      函数调用：
        函数名：read
    表达式语句：
```

图 3-2 运行截图

```
lzw@ubuntu: ~/Desktop/YH and gowhere
File Edit View Search Terminal Help
ASSIGNOP
  ID：i
  INT：1
表达式语句：
  ASSIGNOP
    ID：h
    YH
      ID：i
      ID：i
表达式语句：
  ASSIGNOP
    ID：beijin
    GOWHERE：~~a
循环语句：
  循环条件：
    <=
      ID：i
      ID：m
  循环体：
    复合语句：
      复合语句的变量定义：
      复合语句的语句部分：
      表达式语句：
        ASSIGNOP
```

图 3-3 运行截图

注释代码成功去除：

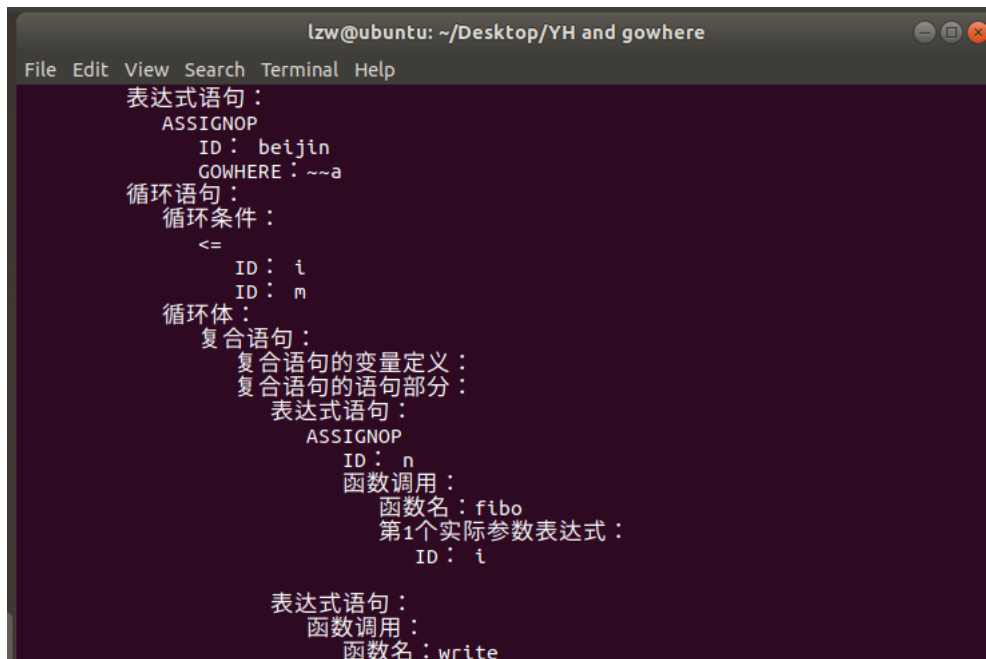


图 3-4 运行截图

(2) 测试样例二代码:

```
int a, b, c;
float m, n;
char i;
gowhere beijin;
int fibo (int a){
    if(a==1||a==2) return 1;
    return fibo (a-1)+fibo ( a-2);
}
int main()
{int m, n,i,h
gowhere beijin;//adawdawdaw
m=read () ;
i=1;
h=i^i;
beijin=~a;
while (i<=m)/*ihlll*/
{
    n=fibo (i) ;write (n) ;i=i+1;}
return 1;}
```

图 3-5 错误代码截图

代码中有两处错误：1.main 函数中确实分号结束。3.beijin 的赋值出现问题。

```
lzw@ubuntu: ~/Desktop/YH and gowhere
File Edit View Search Terminal Help
表达式语句：
函数调用：
    函数名：write
    第1个实际参数表达式：
        ID： n

表达式语句：
    ASSIGNOP
        ID： i
        PLUS
            ID： i
            ID： l

返回语句：
    ID： l

lzw@ubuntu:~/Desktop/YH and gowhere$ touch text2.c
lzw@ubuntu:~/Desktop/YH and gowhere$ '/home/lzw/Desktop/YH and gowhere/parser' '
/home/lzw/Desktop/YH and gowhere/text2.c'
Grammar Error Found at Line 3 Column 1: syntax error, unexpected ID, expecting $
end or TYPE.
Grammar Error Found at Line 11 Column 1: syntax error, unexpected TYPE, expectin
g SEMI.
    9 bad include line 15
bad string is ~
lzw@ubuntu:~/Desktop/YH and gowhere$
```

图 3-6 错误代码运行截图

## 2.6 小结

第一次做编译实验，其实完全是茫然的。完全不知道要做些什么，也不清楚如何去查找自己需要学习的东西。并且做的东西跟理论课关系好像不是很大，最主要的语法建立过程好像已经被省略掉了。剩下得东西也很迷惑。

后来通过对书本的学习，以及在网上查询相关的原理实现，结合老师给的代码，逐渐懂得了如何去实现老师给的代码，但一开始学到的东西很少，毕竟大多数都是老师给好得，于是我决定要自己添加一些功能，通过自己得添加功能，才能对整个系统体系有更清楚的了解与学习。明白了如何通过 **bison** 和 **flex** 进行编译与自动实现链接。

我在做实验的过程中，渐渐明白了词法分析与语法分析的实际步骤，的确，要想理解词法分析与语法分析，做实验是一个非常好的方法。但是个人认为老师在第一次实验时应该多有一些引导，不论是资料还是提示，我们查找资料和思考的时间真的占用很多时间。

通过自己添加关键字 **gowhere** 和运算符 **YH**，我对于 **flex** 和 **bison** 的运行、编写方式都有了更多的理解。语法规则先是遵循原有的，后来在原有的基础上能够识别后加上了关键字 **gowhere**、运算符 **YH** 等其它运算。

在写程序的过程中，遇到了很多错误，也吸取了很多教训，最后还是把自己想做的功能实现出来，还是学到了很多。

## 3 实验二 语义分析

实验二任务：

- 1、设计符号表存储结构；
- 2、修改语法分析程序中的语义子程序，增加合适的语义信息，填写符号表信息；
- 3、根据符号表信息，能完成静态语义检查，并对样例程序进行分析，识别语义错误；参考文献[1]2.1.6 必做样例；
- 4、可以根据需要输出符号表信息，其中包含符号的类型、作用域层次等信息。

### 3.1 语义子程序描述

对源程序样例进行词法和语法分析，对正确的源程序，建立了抽象语法树 AST，在此基础上完成对源程序的语义分析。要完成的任务如下：

- 1) 用不同的样例覆盖语法规则，进行语义分析，遍历对应的 AST 树，发现语义的错误并输出到屏幕上。
- 2) 通过 AST 算法，对实现递归下降法的算法  
递归下降分析法是一种自顶向下的分析方法，文法的每个非终结符对应一个递归过程(函数)。分析过程就是从文法开始符出发执行一组递归过程(函数)，这样向下推导直到推出句子；或者说从根节点出发，自顶向下为输入串寻找一个最左匹配序列，建立一棵语法树。
- 3) 建立符号表，并在合适的分析位置上进行输出符号表。
- 4) 对预估问题的结论，对错误代码进行纠正，位于每一个对错误的代码进行纠正，在每一个子程序的节点进行代码检测，如果满足输出的条件，就输出输错符号，并同时退出程序

### 3.2 符号表的设计

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。

源程序中的每个标识符在符号表中有 1 个条目，一般由两部分组成:名字栏和信息栏。

如果一个语言对标识符的最大长度有限制，可设计名字栏的域大小为最大长度来容纳整个标识符;若该语言对标识符最大长度无限制或最大长度较大(如: 32，为节省存储空间，可另用一个字符数组存储标识符，在名字栏域中存储其起始地址和长度(字符个数)。

源程序中的标识符种类繁多，不同种类的标识符所需要存储的信息不同。如：变量需存储其类型、存储地址等，数组应存储其数组维数 nm 数组元素类型 T 各维元素个数 d、起始地址 base 等，指针应存储其指向对象类型的位置，函数应存储其参数及类型、返回值类型等……



源程序中的说明将标识符与具有某种类型属性的数据对象相关联。同一个标识符在不同程序位置被说明时代表不同的数据对象。当出现对一个标识符的引用时，需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符号，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。

符号表定义如下：

```
};
//符号表
typedef struct symboltable{
    struct symbol symbols[MAXLENGTH];
    int index;
} symbolTable;

typedef struct symbol_scope_begin {
    //当前作用域的符号在符号表的起始位置序号,这是一个栈结构,当使用顺序表作为符号表时,进入、退出一个作用域时需要对其操作,以完成符号表的管理。对其它形式/
    构
    int TX[30];
    int top;
} symbol_scope_TX;
```

图 3-1 函数补全

其中 symbol 的结构为：

```
struct symbol {           //这里只列出了一个符号表项的部分属性，没考虑属性间的互斥
    char name[33];        //变量或函数名
    int level;            //层号
    int type;             //变量类型或函数返回值类型
    int paramnum;         //对函数适用，记录形式参数个数
    char alias[10];       //别名，为解决嵌套层次使用
    char flag;            //符号标记，函数：'F' 变量：'V' 参数：'P' 临时变量：'T'
    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，
    char struct_name[33];
    // struct Array *arrayPtr;
    struct Struct *structPtr;
    int array_size;
    //或记录函数活动记录大小，目标代码生成时使用
    //函数入口等实验可能会用到的属性...
};
//符号表
```

图 3-2 函数补全

### 3.3 语义错误类型定义

编程是很困难的，而且有很多办法犯错误。误差一般可分为两类：语法错误和语义错误（逻辑错误），此处讨论程序结构不符合语法（包括词法）规则的错误。例如：

A[x, y := 3,1416 + T (T + H)有三处错误：

- A[x, y: 中括号未闭合，缺少右中括号
- 3,1416: 小数点写成了逗号
- T (T + H): 缺少一个算术运算符

由于错误的种类太多，无法一一实现，故本次实验仅完成部分典型错误的鉴别。如下所示：

- (1) 使用未定义的变量；
- (2) 调用未定义或未声明的函数；
- (3) 在同一作用域，名称的重复定义（如变量名、函数名、结构类型名以及结构体成员名等）。为更清楚说明语义错误，这里也可以拆分成几种类型的错误，如变量重复定义、函数重复定义、结构体成员名重复等；

- (4) 对非函数名采用函数调用形式;
- (5) 对函数名采用非函数调用形式访问;
- (6) 函数调用时参数个数不匹配, 如实参表达式个数太多、或实参表达式个数太少;
- (7) 函数调用时实参和形参类型不匹配;
- (8) 对非数组变量采用下标变量的形式访问;
- (9) 数组变量的下标不是整型表达式;
- (10) 对非结构变量采用成员选择运算符“.”;
- (11) 结构成员不存在;
- (12) 赋值号左边不是左值表达式;
- (13) 对非左值表达式进行自增、自减运算;
- (14) 对结构体变量进行自增、自减运算;
- (15) 类型不匹配。如数组名与结构变量名间的运算, 需要指出类型不匹配错误; 有些需要根据定义的语言的语义自行进行界定, 比如:  $32+'A'$ ,  $10*12.3$ , 如果使用强类型规则, 则需要报错, 如果按 C 语言的弱类型规则, 则是允许这类运算的, 但需要在后续阶段需要进行类型转换, 类型统一后再进行对应运算;
- (16) 函数返回值类型与函数定义的返回值类型不匹配;
- (17) 函数没有返回语句 (当函数返回值类型不是 `void` 时);

### 3.4 语义分析实现技术

在语义分析中, 构造了函数 `semantic_Analysis` 来进行语义分析, 采用堆的结构, 进行深度优先遍历, 完成对每一个模块的分析, 同时进行出错检测, 若出现错误就跳过, 同时输出错误代码地址以及错误原因。

其代码描述如下: 其中对数组的定义:

```
break;
case ARRAY_DEC:
    rtn = searchSymbolTable(T->type_id);
    if(rtn != -1){
        if(myTable.symbols[rtn].level == level){
            semantic_error(T->pos, "", "数组名重复定义");
        }
    }
    else{
        strcpy(myTable.symbols[myTable.index].name, T->type_id);
        myTable.symbols[myTable.index].level = level;
        myTable.symbols[myTable.index].flag = 'A';
        myTable.symbols[myTable.index].type = type;
        myTable.symbols[myTable.index].array_size = 0;
        myTable.index++;
        semantic_Analysis(T->ptr[0], type, level, 'A', 0);
    }
break;
```

```

case ARRAY_DEC:
    rtn = searchSymbolTable(T->type_id);
    if(rtn != -1){
        if(myTable.symbols[rtn].level == level){
            semantic_error(T->pos, "", "数组名重复定义");
        }
    }
    else{
        strcpy(myTable.symbols[myTable.index].name, T->type_id);
        myTable.symbols[myTable.index].level = level;
        myTable.symbols[myTable.index].flag = 'A';
        myTable.symbols[myTable.index].type = type;
        myTable.symbols[myTable.index].array_size = 0;
        myTable.index++;
        semantic_Analysis(T->ptr[0], type, level, 'A', 0);
    }
    break;|
case ARRAY_LIST:
    type1 = semantic_Analysis(T->ptr[0], type, level, flag, command);
    if(type1 != INT){
        semantic_error(T->pos, "", "数组下标不是整型表达式");
    }
    else{
        if(command == 0){
            myTable.symbols[myTable.index-1].array_size++;
        }
        else{
            array_size++;
        }
        semantic_Analysis(T->ptr[1], type, level, flag, command);
    }
    break;

```

图 3-3 符号表的判断以及出错检测

对数组的出错判断:

```

case STRUCT_DEC:
    rtn = searchSymbolTable(T->ptr[0]->type_id);
    if(rtn == -1){
        semantic_error(T->pos, T->type_id, "结构体未定义");
    }
    else{
        stru_dec = 1;
        strcpy(struct_name, T->ptr[0]->type_id);
    }
    break;
case EXP_ELE:
    rtn = searchSymbolTable(T->ptr[0]->type_id);
    flag1 = 0;

    if(rtn == -1){
        semantic_error(T->pos, T->ptr[0]->type_id, "结构体变量未定义");
    }
    else{
        if(myTable.symbols[rtn].type != STRUCT){
            semantic_error(T->pos, T->ptr[0]->type_id, "不是结构体");
        }
        else{
            rtn = searchSymbolTable(myTable.symbols[rtn].struct_name);
            if(rtn == -1){
                semantic_error(T->pos, "", "结构体未定义");
                return 0;
            }
            num = rtn;
            exp_ele = 1;
            do{
                num++;
                if(!strcmp(myTable.symbols[num].name, T->type_id)){
                    flag1 = 1;
                    break;
                }
            } while(num < myTable.index && myTable.symbols[num].flag == 'M');
            if(!flag1){
                semantic_error(T->pos, "结构体不含成员变量", T->type_id);
            }
            exp_ele = 0;
            flag1 = 0;
        }
    }

```

以及自己添加的一个自定义的类型 gowhere

```
case FLOAT:
    return FLOAT;
case GOWHERE: //zijiade
    return GOWHERE;
case STRING:
    return STRING;
```

### 3.5 语义分析结果

语义分析的源文件为 test2.c，其内容如下：

```
// 函数参数与局部变量重复 V
// 赋值表达式不是左值 V
// 返回值类型错误 V
int f2(int a,int b)
{
    gowhere h1;
    int a,b;
    a+b=10;
    (a+b)++;
    ++a++;
    return 12.3;
}

// break 不在循环语句中
// continue 不再循环语句中
int f3()
{
    int a,b;
    if (a-12.3)
        continue;
    else
        break;
    while ( a || f3())
    {
        while (1)
            break;
        continue;
    }
    for(a=1;a>0 && f3()>0;a++)
        if (a+b==0.0)
            break;
    return 1;
}

struct A{
    int a;
    float b;
    char c;
};

// 结构体不是左值 2
// x 不是结构体
// m.a 不是结构体
// 缺少 return
float f4(float x)
{
    struct A m;
    int a;
    m=12;
    m++;
    m.b=17;
    x.a=m.b;
    m.d=m.a.a;
}

// 返回值类型不匹配
// a 不是数组
// f5 不是数组
// 数组下标不是整型表达式
int f5(int a)
{
    float x[10][20],y;
    x=a[1]+f5[1];
    x[1]=x[1][y]+x[1+1][1];
    return y;
}

int f1(int a, int y){
    return 0;
}
```

运行结果如下图所示：  
可以看到自己添加的 gowhere 变量类型成功识别出来。

```
lzw@ubuntu:~/Desktop/lab2$
lzw@ubuntu:~/Desktop/lab2$ '/home/lzw/Desktop/lab2/parser' '/home/lzw/Desktop/lab2/test.c'
ERROR! 第8行, 赋值表达式需要左值
ERROR! 第9行, 自增自减表达式需要左值
ERROR! 第11行, 返回值类型错误

***Symbol Table***
-----
Index  Name    Level  Type    Flag    Param_num  Array_size
-----
0      read    0      int     F       0
1      x        1      int     P
2      write   0      int     F       1
3      f2       0      int     F       2
4      a        1      int     P
5      b        1      int     P
6      h1       1      gowhere T
7      a        1      int     T
8      b        1      int     T
-----

ERROR! 第20行, continue语句要在循环语句中
ERROR! 第22行, break语句要在循环语句或switch语句中

***Symbol Table***
-----
Index  Name    Level  Type    Flag    Param_num  Array_size
-----
0      read    0      int     F       0
1      x        1      int     P
2      write   0      int     F       1
3      f2       0      int     F       2
4      f3       0      int     F       0
5      a        1      int     T
6      b        1      int     T
-----

ERROR! 第48行, 赋值表达式需要左值
ERROR! 第49行, m 不是左值
ERROR! 第51行, x 不是结构体
ERROR! 第52行, 结构体不含成员变量 d
ERROR! 第52行, a 不是结构体

***Symbol Table***
-----
Index  Name    Level  Type    Flag    Param_num  Array_size
-----
0      read    0      int     F       0
1      x        1      int     P
2      write   0      int     F       1
3      f2       0      int     F       2
4      f3       0      int     F       0
5      A        0      struct  S
6      a        0      int     M
7      b        0      float   M
8      c        0      char    M
9      f4       0      float   F       1
10     x        1      float   P
11     m        1      struct  T
12     a        1      int     T
-----

ERROR! 第62行, a 不是数组
ERROR! 第62行, f5 不是数组
ERROR! 第63行, x 数组维数不一致
ERROR! 第63行, 数组下标不是整型表达式
ERROR! 第63行, x 数组维数不一致
ERROR! 第64行, 返回值类型错误
```

图 3-5 实验结果

### 3.6 小结

本次实验实际上是继承了上一次遍历语法树的思想，绝大部分代码还是参考了附录和网上的样例，进行对代码的理解与记忆之后，在框架结构完整的前提下，进行对框架的修改与更新，加入了部分自定义的功能。

其次，本次实验使用了构造最简单的符号表—顺序表，通过本次实验，查阅了《编译原理实践与指导教程》一书中的思想以及 `def.h` 文件中原本的结构，使我对语义分析有了进一步的认识。

通过对书本的学习，以及在网上查询相关的原理实现，结合老师给的代码，逐渐懂得了如何去实现老师给的代码，但一开始学到的东西很少，毕竟大多数都是老师给好的，于是我决定要自己添加一些功能，比如添加自定义的标识符 `gowhere` 的变量和错误判断，通过自己得添加功能，才能对整个系统体系有更清楚的了解与学习。我在做实验的过程中，渐渐明白了词法分析与语法分析的实际步骤，的确，要想理解词法分析与语法分析，做实验是一个非常好的方法。但是个人认为老师在第一次实验时应该多有一些引导，不论是资料还是提示，我们查找资料和思考的时间真的占用很多时间。

通过自己添加关键字 `gowhere` 和运算符 `YH`，我对于 `flex` 和 `bison` 的运行、编写方式都有了更多的理解。语法规则先是遵循原有的，后来在原有的基础上能够识别后加上了关键字 `gowhere`、运算符 `YH` 等其它运算。在写程序的过程中，遇到了很多错误，也吸取了很多教训，最后还是把自己想做的功能实现出来，最后学到了很多。

## 4 实验三 中间代码生成

实验三任务：

- 1、设计中间代码生成需要的数据结构；
- 2、完成对 AST 遍历并生成 TAC 序列，并输出；参考文献[1]3.1.6；

### 4.1 中间代码格式定义

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 4-1 所示。

表 4-1 中间代码定义

语法	描述	Op	Opn1	Opn2	Result
LABEL x	定义标号 x	LABEL			X
FUNCTION f:	定义函数 f	FUNCTION			F
x := y	赋值操作	ASSIGN	X		X
x := y + z	加法操作	PLUS	Y	Z	X
x := y - z	减法操作	MINUS	Y	Z	X
x := y * z	乘法操作	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

### 4.2 中间代码生成规则定义

通过前面对 AST 遍历，完成了语义分析后，如果没有语法语义错误，就可以再次对 AST 进行遍历，计算相关的属性值，建立符号表，并生成以三地址代码 TAC 作为中间语言的中间语言代码序列。本文中，在这里对本实验的实现做了一些限制，假设数据类型只包含整数类型，不包含如浮点数、数组、结构和指针等其它数据类型。其它数据类型的实现，自行选择。

#### 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加法、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。

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 <sub>1</sub>	label1 = new_label() label2 = new_label() code1 = translate_Cond(Exp, label1, label2, sym_table) code2 = translate_Stmt(Stmt <sub>1</sub> , sym_table) return code1 + [LABEL label1] + code2 + [LABEL label2]
IF LP Exp RP Stmt <sub>1</sub> ELSE Stmt <sub>2</sub>	label1 = new_label() label2 = new_label() label3 = new_label() code1 = translate_Cond(Exp, label1, label2, sym_table) code2 = translate_Stmt(Stmt <sub>1</sub> , sym_table) code3 = translate_Stmt(Stmt <sub>2</sub> , sym_table) return code1 + [LABEL label1] + code2 + [GOTO label3] + [LABEL label2] + code3 + [LABEL label3]
WHILE LP Exp RP Stmt <sub>1</sub>	label1 = new_label() label2 = new_label() label3 = new_label() code1 = translate_Cond(Exp, label2, label3, sym_table) code2 = translate_Stmt(Stmt <sub>1</sub> , sym_table) return [LABEL label1] + code1 + [LABEL label2] + code2 + [GOTO label1] + [LABEL label3]

图 4.2 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 <sub>1</sub> RELOP Exp <sub>2</sub>	<pre> t1 = new_temp() t2 = new_temp() code1 = translate_Exp(Exp<sub>1</sub>, sym_table, t1) code2 = translate_Exp(Exp<sub>2</sub>, sym_table, t2) op = get_relop(RELOP); code3 = [IF t1 op t2 GOTO label_true] return code1 + code2 + code3 + [GOTO label_false] </pre>
NOT Exp <sub>1</sub>	<pre> return translate_Cond(Exp<sub>1</sub>, label_false, label_true, sym_table) </pre>
Exp <sub>1</sub> AND Exp <sub>2</sub>	<pre> label1 = new_label() code1 = translate_Cond(Exp<sub>1</sub>, label1, label_false, sym_table) code2 = translate_Cond(Exp<sub>2</sub>, label_true, label_false, sym_table) return code1 + [LABEL label1] + code2 </pre>
Exp <sub>1</sub> OR Exp <sub>2</sub>	<pre> label1 = new_label() code1 = translate_Cond(Exp<sub>1</sub>, label_true, label1, sym_table) code2 = translate_Cond(Exp<sub>2</sub>, label_true, label_false, sym_table) return code1 + [LABEL label1] + code2 </pre>
(other cases)	<pre> t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) code2 = [IF t1 != #0 GOTO label_true] return code1 + code2 + [GOTO label_false] </pre>

图 4.3 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	<pre> function = lookup(sym_table, ID) if (function.name == "read") return [READ place] return [place := CALL function.name] </pre>
ID LP Args RP	<pre> 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] </pre>

图 4.4 函数调用翻译模式

translate_Args(Args, sym_table, arg_list) = case Args of	
Exp	<pre> t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) arg_list = t1 + arg_list return code1 </pre>
Exp COMMA Args <sub>1</sub>	<pre> t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) arg_list = t1 + arg_list code2 = translate_Args(Args<sub>1</sub>, sym_table, arg_list) return code1 + code2 </pre>

图 4.5 函数参数的翻译模式

### 4.3 中间代码生成过程

中间代码生成过程整体部分参考的是实验附录中的代码，就不做多余的赘述了，以下主要介绍由自己实现的额外的功能。

#### 1. 自定义的标识符 gowhere

其一是添加了一个自定义的标识符，具体定义在前两个实验已经介绍过。

```

        break;
    case GOWHERE: // TODO
        gowhere_exp(T);
        break;

```

当 EXP 函数的 switch 函数运行到了 GOWHERE 时调用 gowhere\_exp 函数进行处理，增加对应的中间代码。

```

void gowhere_exp(struct ASTNode *T)
{
    int rtn, num, width;
    struct opn opn1, opn2, result;
    T->place = fill_Temp(newTemp(), LEV, T->type, 'T', T->offset); //为整常量生成一个临时变量
    T->type = GOWHERE;
    opn1.kind = GOWHERE;
    opn1.const_gowhere = T->type_gowhere;
    result.kind = ID;
    strcpy(result.id, symbolTable.symbols[T->place].alias);
    result.offset = symbolTable.symbols[T->place].offset;
    T->code = genIR(ASSIGNOP, opn1, opn2, result);
    // T->width = 1;
}

```

T->place = fill\_Temp(newTemp(), LEV, T->type, 'T', T->offset);这一语句的作用是给生成一个新的四元式，并将 T->type 的内容，偏移量等保存到四元式中，标识四元式的元素属性。

GENIR 函数生成一条 TAC 代码的结点组成的双向循环链表，返回头指针，以便后面遍历生成中间代码。

```

//输出中间代码
void prnIR(struct codenode *head)
{
    char opnstr1[32], opnstr2[32], resultstr[32];
    struct codenode *h = head;
    if(h) {
        do
        {
            if (h->opn1.kind == INT)
                sprintf(opnstr1, "%d", h->opn1.const_int);
            if (h->opn1.kind == CHAR)
                sprintf(opnstr1, "%c", h->opn1.const_char);
            if (h->opn2.kind == GOWHERE)
                sprintf(opnstr2, "%s", h->opn2.const_gowhere);
            ...

```

同时要在 code.c 文件中输出代码的函数中加入对应的判断条件以便成功输出对应新增加的标识符。

## 2.array 数组

第二个自主添加的功能就是添加了对数组的识别。

```

void exp_array(struct ASTNode *T){
    int rtn;
    struct ASTNode *T0;
    rtn=searchSymbolTable(T->type_id);
    if (rtn==-1)
        semantic_error(T->pos,T->type_id, "变量未定义");
    else if(symbolTable.symbols[rtn].flag != 'A')
        semantic_error(T->pos,T->type_id, "变量不是数组");
    else {
        ...

```

首先需要对数组进行最开始判断，在 searchSymbolTable 函数中寻找对应的数组 id，若没有直接推出，若不是数组，也直接退出。

接下来对数组的各个属性的定义的处理。

```

    tv = t->ptr[0],
    T->place = rtn; //结点保存变量在符号表中的位置

```

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，中间结果需要用临时变量保存，临时变量也需要登记到符号表中。

```

T->code = NULL; //标识符不需要生成TAC

```

.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。由于标识符不需要生成 TAC，故此相为空。

```

// 计算宽度
T->offset = (T->type == INT ? 4 : (T->type == FLOAT ? 8 : 1)) * compute_width(T->ptr[0], symbolTable.symbols[rtn].array, 0); // 内存中偏移值
// printf("offset: %d\n", T->offset);

```

.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。要根据数组的类型进行定义，若是 int 则为 4，若是 float 则为 8，否则就是 char 为 1。

```

T->width = 0; //不再使用新单元
while(T0->kind==ARRAY_LIST){
    Exp(T0->ptr[0]);
    if(T0->ptr[0]->type != INT){
        semantic_error(T->pos, "", "数组维数需要整型");
        break;
    }
    if(index == 8){
        semantic_error(T->pos, "", "数组维度超过最大值");
        break;
    }
    else if(symbolTable.symbols[rtn].array[index] <= T0->type_int){
        semantic_error(T->pos, "", "数组维度超过定义值");
        break;
    }
    index++;
    T0=T0->ptr[1];
}
}

```

接下来就是循环对数组中的每个元素进行遍历，同时判断相对应的约束条件，为了方便起见，对所要求的 index 值都以 8 作为最大上限值。

## 4.4 代码优化（可选）

## 4.5 中间代码生成结果

测试文件 test.c 内容如下所示：

```

int a,b,c;
char arr[10];
float m,n;
struct node{
    int a;
    int b;
};
int main()
{
    int m, n, i;
    float f;
    char ch;

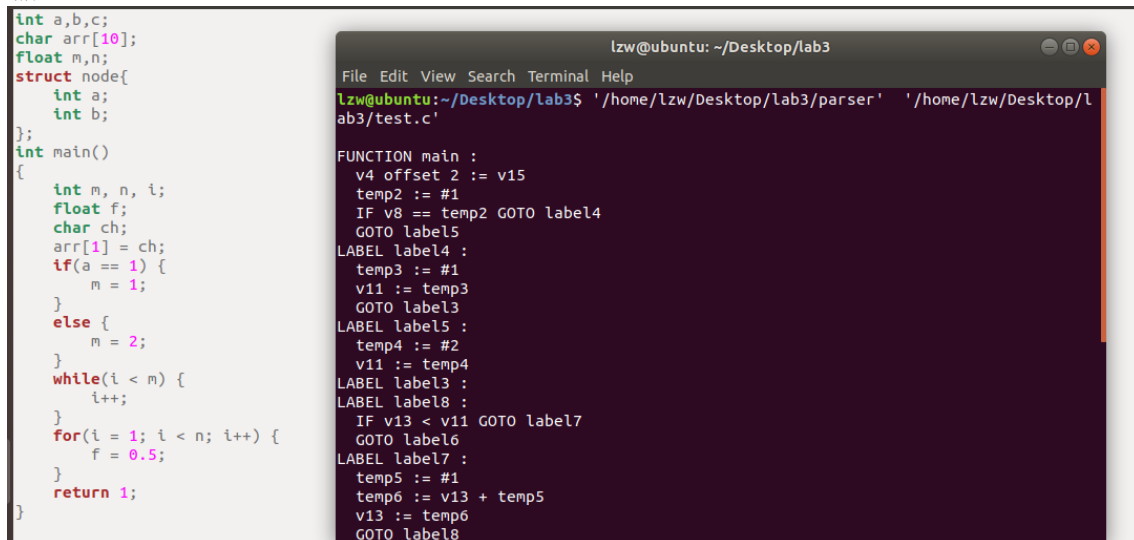
```

```

arr[1] = ch;
if(a == 1) {
    m = 1;
}
else {
    m = 2;
}
while(i < m) {
    i++;
}
for(i = 1; i < n; i++) {
    f = 0.5;
}
return 1;
}

```

输出结果如下所示:



```

int a,b,c;
char arr[10];
float m,n;
struct node{
    int a;
    int b;
};
int main()
{
    int m, n, i;
    float f;
    char ch;
    arr[1] = ch;
    if(a == 1) {
        m = 1;
    }
    else {
        m = 2;
    }
    while(i < m) {
        i++;
    }
    for(i = 1; i < n; i++) {
        f = 0.5;
    }
    return 1;
}

```

```

lzw@ubuntu: ~/Desktop/lab3
File Edit View Search Terminal Help
lzw@ubuntu:~/Desktop/lab3$ '/home/lzw/Desktop/lab3/parser' '/home/lzw/Desktop/lab3/test.c'

FUNCTION main :
    v4 offset 2 := v15
    temp2 := #1
    IF v8 == temp2 GOTO label4
    GOTO label5
LABEL label4 :
    temp3 := #1
    v11 := temp3
    GOTO label3
LABEL label5 :
    temp4 := #2
    v11 := temp4
LABEL label3 :
LABEL label8 :
    IF v13 < v11 GOTO label7
    GOTO label6
LABEL label7 :
    temp5 := #1
    temp6 := v13 + temp5
    v13 := temp6
    GOTO label8

```

测试代码 2 如下:

```

int a, b, c;
float m, n;
int b[10][10]; // test3 conflict declare
int d[10][10];

// int test(int a); // 未实现函数声明

// typedef struct
// {
//     int a;
//     int b;
// } _TestStruct; // 未实现 typedef

```

```

struct node{
    int a;
    int b;
};

//double *f, g;
int fibo(int a)
{
    // _TestStruct tt; // 未实现 typedef
    struct node tt;

    if ((--a) == 1 || a == 2)
        return 1;
    return fibo(a - 1) + fibo(a - 2);
    //just Test
    if (a > 10)
        return fibo(a - 1, a - 2); //test6 too many arguments
    //
    if (b > 100)
        return fibo(a - 1) + fibo(a - 2);
    else
    {
        b--;
    }

    fibo(tt); //test 7

    a = 1;
    b = 2;
    tt.a = d[a][b]; // test 9
    return 0;
}
char testf()
{
    return 'a';
}
int main(int argc)
{

    int m=10, n, i;

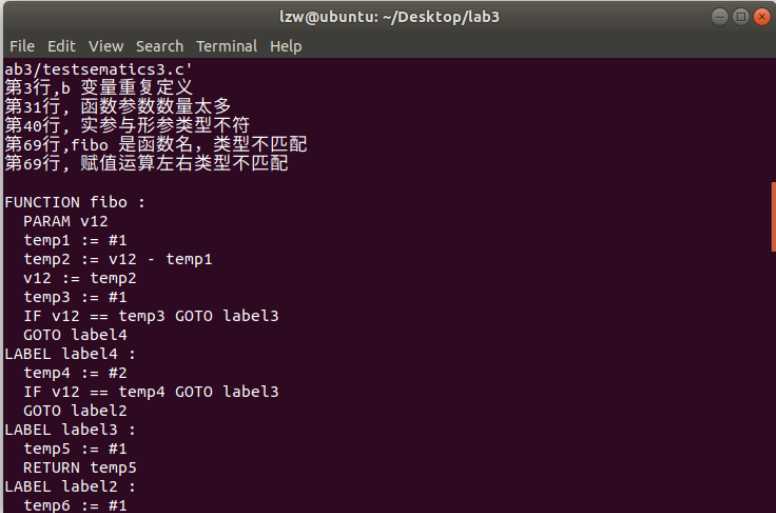
```

```

i = 1;
while ((i+1) <= m)
{
    n = fibo(i);
    i = i + 1;
    if(i>10)
    {
        break; //test18
    }
    else
        continue; // test19
}
n = fibo; // test 5
return 1;
}

```

结果如下所示:

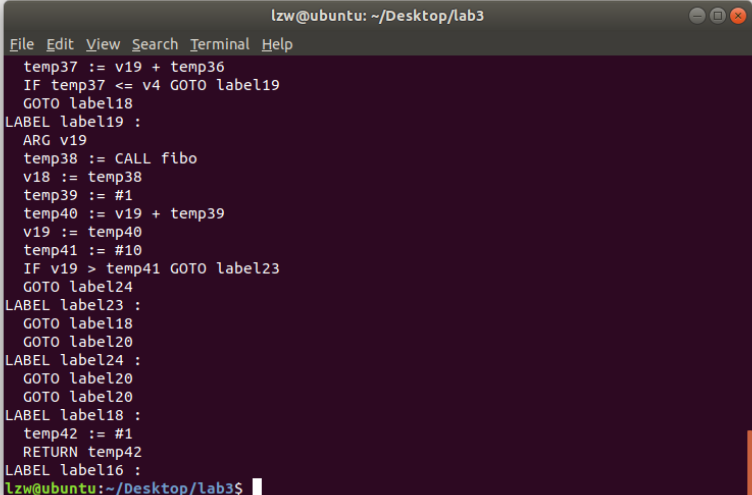


```

lzw@ubuntu: ~/Desktop/lab3
File Edit View Search Terminal Help
ab3/testsemantics3.c'
第3行, b 变量重复定义
第31行, 函数参数数量太多
第40行, 实参与形参类型不符
第69行, fibo 是函数名, 类型不匹配
第69行, 赋值运算左右类型不匹配

FUNCTION fibo :
  PARAM v12
  temp1 := #1
  temp2 := v12 - temp1
  v12 := temp2
  temp3 := #1
  IF v12 == temp3 GOTO label3
  GOTO label4
LABEL label4 :
  temp4 := #2
  IF v12 == temp4 GOTO label3
  GOTO label2
LABEL label3 :
  temp5 := #1
  RETURN temp5
LABEL label2 :
  temp6 := #1

```



```

lzw@ubuntu: ~/Desktop/lab3
File Edit View Search Terminal Help
temp37 := v19 + temp36
IF temp37 <= v4 GOTO label19
GOTO label18
LABEL label19 :
  ARG v19
  temp38 := CALL fibo
  v18 := temp38
  temp39 := #1
  temp40 := v19 + temp39
  v19 := temp40
  temp41 := #10
  IF v19 > temp41 GOTO label23
  GOTO label24
LABEL label23 :
  GOTO label18
  GOTO label20
LABEL label24 :
  GOTO label20
  GOTO label20
LABEL label18 :
  temp42 := #1
  RETURN temp42
LABEL label16 :
lzw@ubuntu:~/Desktop/lab3$

```

## 4.6 小结

通过对书本的学习，以及在网上查询相关的原理实现，结合老师给的代码，逐渐懂得了如何去实现老师给的代码。

编译原理的实验中最核心的数据结构之一就是中间代码生成。我所采用得中间代码表达形式是中层次得中间代码，因为之前没有考虑到第四次得目标代码生成，所以没有考虑到更底层得中间代码表示。

第三次实验应该是最难的一次，我改动的地方不算多，但其中数组函数 `array-exp` 的定义的原理实现还是花费了大量的时间去寻找对应的教程，虽然其实大部分逻辑都是沿用一二次的代码。此次工程实在是有点多，阅读了大量资料和别人的代码。最终，经过不断尝试最终成功调试出。通过本次实验，查阅了《编译原理实践与指导教程》一书中的思想以及 `def.h` 文件中原本的结构，使我对中间代码生成的原理和逻辑有了进一步的认识。

## 5 实验四 目标代码生成

实验四任务：

- 1、将 TAC 指令序列转换成目标代码；
- 2、完成样例的分析转换；参考文献[1]4.1.6；
- 3、能在模拟器上运行并得到结果；例如 **SPIM Simulator**；

### 5.1 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 5-1 所示。其中  $\text{reg}(x)$  表示变量  $x$  所分配的寄存器。

表 5-1 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 $\text{reg}(x)$ , k
$x := y$	move $\text{reg}(x)$ , $\text{reg}(y)$
$x := y + z$	add $\text{reg}(x)$ , $\text{reg}(y)$ , $\text{reg}(z)$
$x := y - z$	sub $\text{reg}(x)$ , $\text{reg}(y)$ , $\text{reg}(z)$
$x := y * z$	mul $\text{reg}(x)$ , $\text{reg}(y)$ , $\text{reg}(z)$
$x := y / z$	div $\text{reg}(y)$ , $\text{reg}(z)$ mflo $\text{reg}(x)$
GOTO x	j x
RETURN x	move \$v0, $\text{reg}(x)$ jr \$ra
IF $x == y$ GOTO z	beq $\text{reg}(x)$ , $\text{reg}(y)$ , z
IF $x != y$ GOTO z	bne $\text{reg}(x)$ , $\text{reg}(y)$ , z
IF $x > y$ GOTO z	bgt $\text{reg}(x)$ , $\text{reg}(y)$ , z
IF $x \geq y$ GOTO z	bge $\text{reg}(x)$ , $\text{reg}(y)$ , z
IF $x < y$ GOTO z	ble $\text{reg}(x)$ , $\text{reg}(y)$ , z
IF $x \leq y$ GOTO z	blt $\text{reg}(x)$ , $\text{reg}(y)$ , z
X:=CALL f	jal f move $\text{reg}(x)$ , \$v0

### 5.2 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，那些未被闲置的继勋奇也



没有对减少目标代码的方寸次数做出任何贡献。

### 5.3 目标代码生成算法

目标代码生成算法如表 5-2 所示。

表 5-2 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 $x \geq y$ 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对中间代码进行翻译。核心思想是将实验三生成的中间代码读入, 分别进行翻译处理。

### 1.对定义的 int, float, char, 数组类型的翻译

```
int cnt = 0;
do{
    cnt++;
    switch (h->op)
    {
        case ASSIGNOP:
            if(h->opn1.kind == INT) {
                fprintf(fp, " li $t3, %d\n", h->opn1.const_int);
            }
            else if(h->opn1.kind == FLOAT) {
                fprintf(fp, " li $t3, %f\n", h->opn1.const_float);
            }
            else if(h->opn1.kind == CHAR) {
                fprintf(fp, " li $t3, %c\n", h->opn1.const_char);
            }
            else{
                int rtn;
                char sub[20];
                if(strstr(h->opn1.id, " offset ") != NULL) {
                    substr(sub, h->opn1.id, 0, 2);
                    rtn = searchSymbolTableByAlias(sub);
                }
                else
                    rtn = searchSymbolTableByAlias(h->opn1.id);
                // printf("%d %s %d %d\n", rtn, , symbolTable.symbols[rtn].level, LEV);
                if(rtn != -1 && symbolTable.symbols[rtn].level == 0){
                    fprintf(fp, " lw $t1, %d($gp)\n", h->opn1.offset);
                    fprintf(fp, " move $t3, $t1\n");
                }
                else{
                    fprintf(fp, " lw $t1, %d($sp)\n", h->opn1.offset);
                    fprintf(fp, " move $t3, $t1\n");
                }
            }
    }
}
```

当输入变量为以上标识符的时候, 将其分别翻译为对应的 mips 语句, 写入文件 object.c 中, 否则若不是 int, float, char 就对输入的四元式进行判断, 用 strstr 函数寻找第一个对应的 offset 值, 如果没有匹配到, 那么就是 string 字符串, 用 substr 函数进行复制, 否则为数组, 直接等于赋值。最后再进行对应的输出。

```
int rtn;
char sub[20] = {'\0'};
if(strstr(h->result.id, " offset ") != NULL) {
    for(int i = 0; i < strlen(h->result.id); i++){
        if(h->result.id[i] != ' '){
            sub[i] = h->result.id[i];
        }
    }
    rtn = searchSymbolTableByAlias(sub);
}
else
    rtn = searchSymbolTableByAlias(h->result.id);
if(rtn != -1 && symbolTable.symbols[rtn].level == 0){
    fprintf(fp, " sw $t3, %d($gp)\n", h->result.offset);
}
else{
    fprintf(fp, " sw $t3, %d($sp)\n", h->result.offset);
}
break;
```

对输出的结果的属性进行处理。

## 2.对运算符的处理

```
case PLUS:
case MINUS:
case STAR:
case DIV:
case MOD:
    fprintf(fp, "    lw $t1, %d($sp)\n", h->opn1.offset);
    fprintf(fp, "    lw $t2, %d($sp)\n", h->opn2.offset);
    if(h->op == PLUS)
        fprintf(fp, "    add $t3, $t1, $t2\n");
    else if(h->op == MINUS)
        fprintf(fp, "    sub $t3, $t1, $t2\n");
    else if(h->op == STAR)
        fprintf(fp, "    mul $t3, $t1, $t2\n");
    else if(h->op == DIV){
        fprintf(fp, "    div $t1, $t2\n");|
        fprintf(fp, "    mflo $t3\n");
    }
    else{
        fprintf(fp, "    div $t1, $t2\n");
        fprintf(fp, "    mfhi $t3\n");
    }
    fprintf(fp, "    sw $t3, %d($sp)\n", h->result.offset);
    break;

case JLE:
case JLT:
case JGE:
case JGT:
case EQ:
case NEQ:
    fprintf(fp, "    lw $t1, %d($sp)\n", h->opn1.offset);
    fprintf(fp, "    lw $t2, %d($sp)\n", h->opn2.offset);
    if (h->op==JLE)
        fprintf(fp, "    ble $t1,$t2,%s\n", h->result.id);
    else if (h->op==JLT)
        fprintf(fp, "    blt $t1,$t2,%s\n", h->result.id);
    else if (h->op==JGE)
        fprintf(fp, "    bge $t1,$t2,%s\n", h->result.id);
    else if (h->op==JGT)
        fprintf(fp, "    bgt $t1,$t2,%s\n", h->result.id);
    else if (h->op==EQ)
        fprintf(fp, "    beq $t1,$t2,%s\n", h->result.id);
    else
        fprintf(fp, "    bne $t1,$t2,%s\n", h->result.id);
    break;
case ARG:
    break;
```

用前面的对应的 mips 的翻译方法进行一一对应翻译，难度不大。

## 3.对函数的处理

```
case FUNCTION:
    if(!strcmp(h->result.id, "main")){
        main_flag = 1;
        LEV = 0;
        main_call = 0;
    }
    fprintf(fp, "\n%s:\n", h->result.id);
    if(!strcmp(h->result.id, "main"))
        fprintf(fp, "    addi $sp, $sp, -%d\n", symbolTable.symbols[h->result.offset].offset);
    break;
```

分为两种情况，若是 main 函数，则要将对应的 flag 标志进行置 1 处理，并同时两个参数置零。若为普通函数，只需要输出函数名即可。

## 4.call 函数的处理

```

case CALL:
    call_flag = 1;
    LEV++;
    if (!strcmp(h->opn1.id,"read")){
        fprintf(fp, "    addi $sp, $sp, -4\n");
        fprintf(fp, "    sw $ra,0($sp)\n");
        fprintf(fp, "    jal read\n");
        fprintf(fp, "    lw $ra,0($sp)\n");
        fprintf(fp, "    addi $sp, $sp, 4\n");
        fprintf(fp, "    sw $v0, %d($sp)\n",h->result.offset);
        break;
    }
    if (!strcmp(h->opn1.id,"write")){
        fprintf(fp, "    lw $a0, %d($sp)\n",h->prior->result.offset);
        fprintf(fp, "    addi $sp, $sp, -4\n");
        fprintf(fp, "    sw $ra,0($sp)\n");
        fprintf(fp, "    jal write\n");
        fprintf(fp, "    lw $ra,0($sp)\n");
        fprintf(fp, "    addi $sp, $sp, 4\n");
        break;
    }
    for(p = h,i = 0;i < symbolTable.symbols[h->opn1.offset].paramnum; i++)
        p=p->prior;
    fprintf(fp, "    move $t0,$sp\n"); |
    fprintf(fp, "    addi $sp, $sp, -%d\n", symbolTable.symbols[h->opn1.offset].offset);
    fprintf(fp, "    sw $ra,0($sp)\n");
    i=h->opn1.offset+1;
    while (symbolTable.symbols[i].flag == 'P'){
        fprintf(fp, "    lw $t1, %d($t0)\n", p->result.offset);
        fprintf(fp, "    move $t3,$t1\n");
        fprintf(fp, "    sw $t3,%d($sp)\n", symbolTable.symbols[i].offset);
        p=p->next;
        i++;
    }
    fprintf(fp, "    jal %s\n",h->opn1.id);
    fprintf(fp, "    lw $ra,0($sp)\n");
    fprintf(fp, "    addi $sp,$sp,%d\n",symbolTable.symbols[h->opn1.offset].offset);
    fprintf(fp, "    sw $v0,%d($sp)\n", h->result.offset);
    break;

```

遇到 call 函数就判断被调用的函数时 read 函数还是 write 函数，对不同的方法调用先对应的 MIPS 代码。

## 5.4 目标代码生成结果

输入的测试代码为：

```

int a, b, c;
float m, n;
int d[10][10];
struct node{
    int a;
    int b;
};
int fibo(int a)
{
    struct node tt;
    if (a == 1 || a == 2){
        return 1;
    }
    if (a>=3)
    {
        tt.a = d[1][2];
        c = d[1][2];
        return fibo(a-1) + fibo(a-2);
    }
    return 0;
}

int main(int argc)
{
    int i, temp;
    for(i = 1;i<10;i++)
    {
        if(i>9)
        {
            temp = fibo(i);
            write(temp);
            break;
        }
        else
        {
            temp = fibo(i);
            write(temp);
            continue;
        }
        i++;
    }
    return 1;
}

```

结果为：

.data

```
_Prompt: .ascii "Enter an integer:  "
```

```
_ret: .ascii "\n"
```

```
.globl main
```

```
.text
```

```
j main
```

```
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
```

```
fibonacci:
```

```
    li $t3, 1
```

```
    sw $t3, 24($sp)
```

```
    lw $t1, 12($sp)
```

```
    lw $t2, 24($sp)
```

```
    beq $t1,$t2,label3
```

```
    j label4
```

```
label4:
```

```
    li $t3, 2
```

```
    sw $t3, 28($sp)
```

```
    lw $t1, 12($sp)
```

```
    lw $t2, 28($sp)
```

```
    beq $t1,$t2,label3
```

```
    j label2
```

```
label3:
```

```
    li $t3, 1
```

```
    sw $t3, 32($sp)
```

```
    lw $v0,32($sp)
```

```
    jr $ra
```

```
label2:
```

```
    li $t3, 3
```

```
    sw $t3, 36($sp)
```

```
    lw $t1, 12($sp)
```

```

lw $t2, 36($sp)
bge $t1,$t2,label6
j label5
label6:
lw $t1, 76($gp)
move $t3, $t1
sw $t3, 16($sp)
lw $t1, 76($gp)
move $t3, $t1
sw $t3, 8($gp)
li $t3, 1
sw $t3, 56($sp)
lw $t1, 12($sp)
lw $t2, 56($sp)
sub $t3, $t1, $t2
sw $t3, 60($sp)
move $t0,$sp
addi $sp, $sp, -88
sw $ra,0($sp)
lw $t1, 60($t0)
move $t3,$t1
sw $t3,12($sp)
jal fibo
lw $ra,0($sp)
addi $sp,$sp,88
sw $v0,64($sp)
li $t3, 2
sw $t3, 68($sp)
lw $t1, 12($sp)
lw $t2, 68($sp)
sub $t3, $t1, $t2
sw $t3, 72($sp)
move $t0,$sp
addi $sp, $sp, -88
sw $ra,0($sp)
lw $t1, 72($t0)
move $t3,$t1
sw $t3,12($sp)
jal fibo
lw $ra,0($sp)
addi $sp,$sp,88
sw $v0,76($sp)
lw $t1, 64($sp)
lw $t2, 76($sp)

```

```

    add $t3, $t1, $t2
    sw $t3, 80($sp)
    lw $v0, 80($sp)
    jr $ra
label5:
    li $t3, 0
    sw $t3, 84($sp)
    lw $v0, 84($sp)
    jr $ra
label1:

main:
    addi $sp, $sp, -56
    li $t3, 1
    sw $t3, 24($sp)
    lw $t1, 24($sp)
    move $t3, $t1
    sw $t3, 16($sp)
label20:
    li $t3, 10
    sw $t3, 28($sp)
    lw $t1, 16($sp)
    lw $t2, 28($sp)
    blt $t1, $t2, label11
    j label10
label11:
    li $t3, 9
    sw $t3, 0($sp)
    lw $t1, 16($sp)
    lw $t2, 0($sp)
    bgt $t1, $t2, label14
    j label15
label14:
    move $t0, $sp
    addi $sp, $sp, -88
    sw $ra, 0($sp)
    lw $t1, 16($t0)
    move $t3, $t1
    sw $t3, 12($sp)
    jal fibo
    lw $ra, 0($sp)
    addi $sp, $sp, 88
    sw $v0, 4($sp)
    lw $t1, 4($sp)

```

```

move $t3, $t1
sw $t3, 20($sp)
lw $a0, 20($sp)
addi $sp, $sp, -4
sw $ra, 0($sp)
jal write
lw $ra, 0($sp)
addi $sp, $sp, 4
j label10
j label13
label15:
move $t0, $sp
addi $sp, $sp, -88
sw $ra, 0($sp)
lw $t1, 16($t0)
move $t3, $t1
sw $t3, 12($sp)
jal fibo
lw $ra, 0($sp)
addi $sp, $sp, 88
sw $v0, 12($sp)
lw $t1, 12($sp)
move $t3, $t1
sw $t3, 20($sp)
lw $a0, 20($sp)
addi $sp, $sp, -4
sw $ra, 0($sp)
jal write
lw $ra, 0($sp)
addi $sp, $sp, 4
j label12
label13:
li $t3, 1
sw $t3, 20($sp)
lw $t1, 16($sp)
lw $t2, 20($sp)
add $t3, $t1, $t2
sw $t3, 24($sp)
lw $t1, 24($sp)
move $t3, $t1
sw $t3, 16($sp)
label12:
li $t3, 1
sw $t3, 32($sp)

```

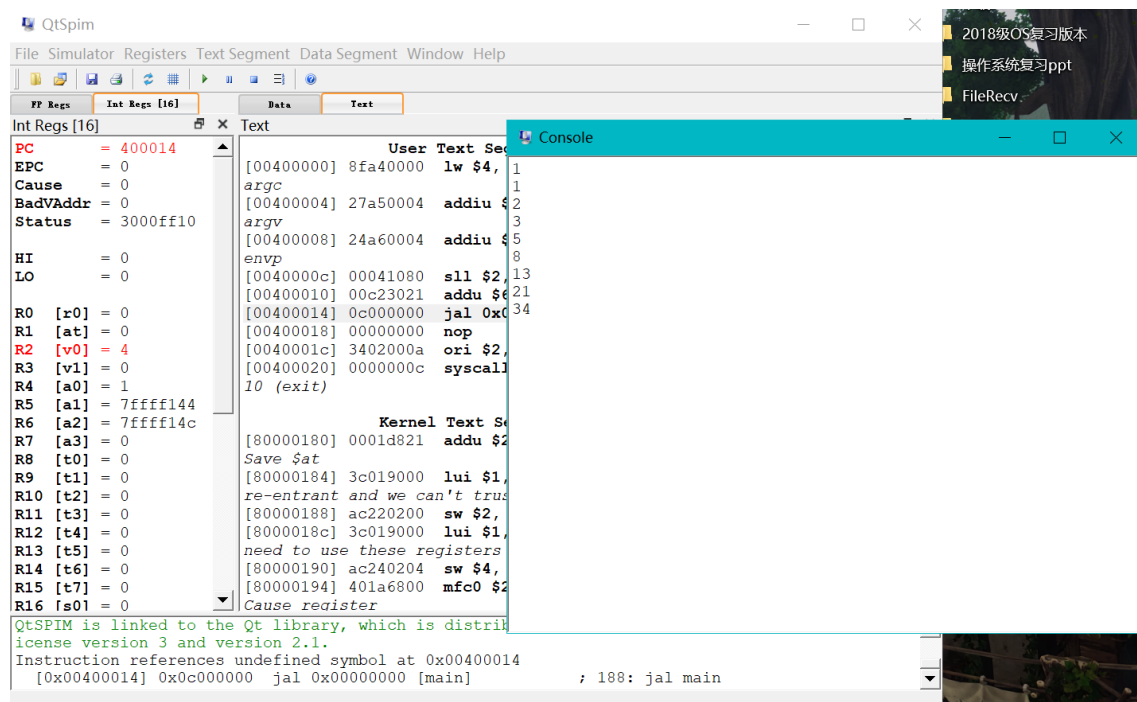


```

lw $t1, 16($sp)
lw $t2, 32($sp)
add $t3, $t1, $t2
sw $t3, 36($sp)
lw $t1, 36($sp)
move $t3, $t1
sw $t3, 16($sp)
j label20
label10:
li $t3, 1
sw $t3, 52($sp)
li $v0, 10
syscall
label9:

```

## 5.5 目标代码运行结果



## 5.6 小结

实验四需要建立在实验三的基础上完成，在动手写代码前，应该先熟悉 SPIM simulator 的使用方法，确定自己是否对 MIPS32 代码的编写熟悉以及完善一些细节部分的内容。

完成实验四的第一步是确定指令选择机制以及寄存器分配算法。部分的代码是参考 GitHub 上的代码，再次基础之上进行对自己的代码的添加以及修改，指令选择算法比较简单，其功能甚至可以由中间代码的某个函数稍加修改而得到。寄存

器分配算法则需要定义一些数据结构。当然本次实验采用的分配算法比较简单，所以 并没有费太大的力气。

确定之后，就可以开始动手，开始时可以先专心处理其它类型的中间代码。你可以先假设 寄存器有无限多个，试着完成指令选择，再打印看指令选择后的代码是否正确。如果以上测试都没有问题，那就要继续完成 ARG、PARAM、RETURN、

CALL 语句的翻译。最后在 SPIM simulator 上进行对输出文件 object.s 的翻译即运行。

## 6 总结

第一次做编译实验，其实完全是茫然的。完全不知道要做些什么，也不清楚如何去查找自己需要学习的东西。并且做的东西跟理论课关系好像不是很大，最主要的语法建立过程好像已经被省略掉了。剩下得东西也很迷惑。

后来通过对书本的学习，以及在网上查询相关的原理实现，结合老师给的代码，逐渐懂得了如何去实现老师给的代码，但一开始学到的东西很少，毕竟大多数都是老师给好的，于是我决定要自己添加一些功能，通过自己得添加功能，才能对整个系统体系有更清楚的了解与学习。明白了如何通过 **bison** 和 **flex** 进行编译与自动实现链接。

我在做实验的过程中，渐渐明白了词法分析与语法分析的实际步骤，的确，要想理解词法分析与语法分析，做实验是一个非常好的方法。但是个人认为老师在第一次实验时应该多有一些引导，不论是资料还是提示，我们查找资料和思考的时间真的占用很多时间。

第二次实验实际上是继承了上一次遍历语法树的思想，绝大部分代码还是参考了附录和网上的样例，进行对代码的理解与记忆之后，在框架结构完整的前提下，进行对框架的修改与更新，加入了部分自定义的功能。

其次，本次实验使用了构造最简单的符号表—顺序表，通过本次实验，查阅了《编译原理实践与指导教程》一书中的思想以及 **def.h** 文件中原本的结构，使我对语义分析有了进一步的认识。

通过对书本的学习，以及在网上查询相关的原理实现，结合老师给的代码，逐渐懂得了如何去实现老师给的代码，但一开始学到的东西很少，毕竟大多数都是老师给好的，于是我决定要自己添加一些功能，比如添加自定义的标识符 **gowhere** 的变量和错误判断，通过自己得添加功能，才能对整个系统体系有更清楚的了解与学习。我在做实验的过程中，渐渐明白了词法分析与语法分析的实际步骤，的确，要想理解词法分析与语法分析，做实验是一个非常好的方法。但是个人认为老师在第一次实验时应该多有一些引导，不论是资料还是提示，我们查找资料和思考的时间真的占用很多时间。

通过自己添加关键字 **gowhere** 和运算符 **YH**，我对于 **flex** 和 **bison** 的运行、编写方式都有了更多的理解。语法规则先是遵循原有的，后来在原有的基础上能够识别后加上了关键字 **gowhere**、运算符 **YH** 等其它运算。在写程序的过程中，遇到了很多错误，也吸取了很多教训，最后还是把自己想做的功能实现出来，最后学到了很多。通过对书本的学习，以及在网上查询相关的原理实现，结合老师给的代码，逐渐懂得了如何去实现老师给的代码。

第三次实验中，编译原理的实验中最核心的数据结构之一就是中间代码生成。我所采用得中间代码表达形式是中层次得中间代码，因为之前没有考虑到第四次得目标代码生成，所以没有考虑到更底层得中间代码表示。

第三次实验应该是最难的一次，我改动的地方不算多，但其中数组函数 **array-exp** 的定义的原理实现还是花费了大量的时间去寻找对应的教程，虽然其实大部分逻辑都是沿用一二次的代码。此次工程实在是有点多，阅读了大量资料和别人的代码。最终，经过不断尝试最终成功调试出。通过本次实验，查阅了《编译原理实践与指导教程》一书中的思想以及 **def.h** 文件中原本的结构，使我对中间代码生成的原理和逻辑有了进一步的认识。

实验四需要建立在实验三的基础上完成，在动手写代码前，应该先熟悉

SPIM simulator 的使用方法，确定自己是否对 MIPS32 代码的编写熟悉以及完善一些细节部分的内容。

完成实验四的第一步是确定指令选择机制以及寄存器分配算法。部分的代码是参考 GitHub 上的代码，再次基础之上进行对自己的代码的添加以及修改，指令选择算法比较简单，其功能甚至可以由中间代码的某个函数稍加修改而得到。寄存器分配算法则需要定义一些数据结构。当然本次实验采用的分配算法比较简单，所以并没有费太大的力气。

确定之后，就可以开始动手，开始时可以先专心处理其它类型的中间代码。你可以先假设 寄存器有无限多个，试着完成指令选择，再打印看指令选择后的代码是否正确。如果以上测试都没有问题，那就要继续完成 ARG、PARAM、RETURN、

以上是对本次实验的总结。

# 参考文献

(格式同华中科技大学毕业论文)

- [1] 许畅 等编著. 《编译原理实践与指导教程》.机械工业出版社
- [2] John Levine著, 陆军 译. 《Flex与Bison》.东南大学出版社
- [3] 王生原等编著. 《编译原理 (第3版)》.清华大学出版社