華中科技大學

编译原理实验
课程报告

Mini-C 语言编译器

班　　级 1701

姓　　名 罗南清

学　　号 U201714868

指导教师 刘　铭

报告日期 2020-01-08

网络空间安全学院

要　求

1、报告需本人独立完成，内容真实。引用资料时，需进行标注说明，并列入参考文献中；如发现抄袭，成绩无效；

2、应说明实验的操作系统环境、采用的主要方法、设计的过程、设计的结果（主要源码文件功能、数据结构、函数说明）、遇到的问题、测试运行的情况；

3、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；

4、评分标准：格式规范美观，符合华中科技大学论文格式要求；采用的方法合适、设计合理；4 个主要实验环节按任务书要求完成。

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 任务

通过简单自定义语言编译器的完整实现，掌握编译原理理论知识，提高灵活运用理论知识以解决实际问题的能力；提高系统软件编写能力。

1.2 目标

课程目标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。

1.3 语言定义

采用简化的 C 语言的文法：mini-c 作为本次实验的语言。

其文法如下所示：

G[program]:

program → ExtDefList

ExtDefList→ExtDef ExtDefList | ε

ExtDef→Specifier ExtDecList ; |Specifier FunDec CompSt

Specifier→int | float | char

ExtDecList→VarDec | VarDec , ExtDecList

VarDec→ID

FucDec→ID (VarList) | ID ( )

VarList→ParamDec , VarList | ParamDec

ParamDec→Specifier VarDec

CompSt→{ DefList StmList }

StmList→Stmt StmList | ε

Stmt→Exp ; | CompSt | return Exp ;

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

DefList→Def DefList | ε

Def→Specifier DecList ;

DecList→Dec | Dec , DecList

Dec→VarDec | VarDec = Exp

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

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

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

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

Args→Exp , Args | Exp

说明：program 为文件开始符号；ExtDefList 指程序语句列表；ExtDef 指某一行的语句；ExtDefList 指后面多行语句；标识符 Specifier；变量声明列表 ExtDecList；函数 FunDec；函数体 CompSt；函数声明 FunDec；变量列表 VarList；参数声明 ParamDec；变量声明 VarDec；Dec 代表声明语句；Stmt 代表

函数内部声明语句；StmtList 代表声明语句列表；Exp 代表表达式；ID 为对应的变量或函数名字。

1.4 主要技术

开发环境：Ubuntu18.04 系统，gnome terminal，Visual Studio Code 编辑器

关键技术：Flex、Bison、C 语言基本知识、抽象语法树 AST、mips32 汇编指令、终端相关操作。

总体说明：本实验由四个部分组成：词法分析、语法分析、语义分析、中间代码生成、目标代码生成。首先将设定的识别词法规则编写进 lex 文件中，将对应的语法规则编写进 bison 文件中，利用此文件对相应的 C 代码进行词法分析与语法分析；对抽象语法树进行遍历输出相应的节点信息。在此过程中，构建符号表，结合抽象语法树的节点信息对程序进行语义分析，生成中间代码。最后，将相应的汇编代码翻译成目标代码 MIPS32 指令，生成的指令最终能在 QtSpim 机器上成功运行。

2 实验一 词法分析和语法分析

2.1 单词文法描述

按照语法定义列出所有的终结符以及非终结符，如表 2.1 所示：

表 2.1: 语法符号

单词符号类型	单词种类码	正则表达式
{id}	ID	[A-Za-z][A-Za-z0-9]*
{int}	INT	[0-9]+
{float}	FLOAT	([0-9]*\.[0-9]+)|([0-9]+\.)
"int"	TYPE	
"float"	TYPE	
"return"	RETURN	
"if"	IF	
"else"	ELSE	
"while"	WHILE	
"for"	FOR	
";"	SEMI	
","	COMMA	
">"|"<"|">="|"<="|"=="|"!="	RELOP	
"="	ASSIGNOP	
"+"	PLUS	
“+=”	COMADD	
“-=”	COMSUB	
“++”	AUTOADD	
“—"	AUTOSUB	
"-"	MINUS	
"*"	STAR	
"/"	DIV	
"&&"	AND	
"||"	OR	
"!"	NOT	
"("	LP	
")"	RP	
"{"	LC	
"}"	RC	
"["	LB	
"]"	RB	
"//"[^\n]*	代表单行注释	
“/”*(\s|.)*?*”/”	代表多行注释	

2.2 语法文法描述

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

定义的非终结符如下：

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

举一个例子进行说明，其中%type <ptr> program ExtDefList，这表示非终结符 ExtDefList 属性值的类型对应联合中成员 ptr 的类型，在本实验中对应一个树结点的指针。

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

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

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

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

%token <type_char> CHAR

%token LP RP LC RC SEMI COMMA LB RB

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

然后，定义运算符的优先级与结合性，如表 2.2 所示

表 2.2 算符优先级与结合性

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

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

```
program: ExtDefList { display($1,0);} /*归约到 program，开始显示语法树,语义分析*/
        ;
/*ExtDefList：外部定义列表，即是整个语法树*/
ExtDefList: {$$=NULL;}/*整个语法树为空*/
          | ExtDef ExtDefList {$$=mknode(EXT_DEF_LIST,$1,$2,NULL,yylineno);}//每一个 EXTDEFLIST 的结点,其第 1 棵子树对应一个外部变量声明或函数
          ;
```

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

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

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

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

举其中一个例子进行说明，对如下语句

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

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

其中，错误处理部分，Stmt(statement)→error SEMI 表示对语句分析时，一旦有错，跳过分号（SEMI），继续进行语法分析。

2.3 词法分析器设计

使用 flex 工具编写 lex 文件，对指定的高级语言程序进行词法分析。

1. 定义部分

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

Name definition

其中 name 是名字，definition 是任意的正则表达式。

在 lex 文件中定义部分如下：

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

2. 规则部分

规则部分是由正则表达式和相应的响应函数组成，其格式为

Pattern {action}

其中 pattern 为正则表达式，其书写规则与前面部分的正则表达式定义相同。而 action 则为将要进行的具体操作，这些操作可以用一段 C 代码表示。Flex 将按照这部分给出饿内容依次尝试每一个规则，尽可能匹配最长的输入串。如果有些内容不匹配任何规则，Flex 默认之将其拷贝到标准输出，想要修改这个默认行为

只需要在所有郭泽的最后加上一条“.” (即匹配任何输入)规则，然后在其对应的action 部分书写想定义的行为即可。

规则部分的代码如下：

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

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

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

3. 用户自定义代码部分

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

以下是用户子程序部分：

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

2.4 语法分析器设计

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

1. 定义部分

所有的词法单元的定义都可以放到这部分。这段 bison 代码以“%{“和”%}“开头，被“%{“和”%}“包含的内容主要是对 stdio.h 的引用。接下来是一些以%token 开头的此法单元（终结符）定义，如果需要采用 Flex 生成的 yylex（），那么在这里定义的词法单元都可以作为 Flex 源代码里的返回值。与终结符相对的，所有未被定义为%token 的符号都会被看作非终结符，这些非终结符要求必须在任意产生式的左边至少出现一次。

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

此外，bison 文件中可能会有辅助定义部分，辅助定义部分如下：

```
%union {
  int type_int;
```

```
  float type_float;
  char type_char;
  char type_id[32];
  struct node *ptr;
};
```

辅助声明部分利用%union 将各种类型统一起来。Bison 中默认将所有的语义值都定义为 int 类型，可以通过定义宏 YYSTYPE 来改变值的类型。如果有多个值类型，则需要通过在 Bison 声明中使用%union 列举出所有的类型。

2. 规则部分

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

3. 用户函数部分

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

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

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

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

4. 抽象语法树节点的建立

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

首先是 AST 树结点的定义：

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

其次是实例化一个语法树节点：

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

5. 显示抽象语法树

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

```
void display(struct node* T,int indent){
```

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

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

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

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

2.5 词法及语法分析器实现结果

联合使用 FLEX 和 Bison 构造词法、语法分析器，通过在终端输入如下命令来构造可执行文件：

```
flex lex.l
bison –d test.y
gcc -o T1 lex.yy.c test.tab.c display.c
```

使用分析的源文件为 test.c，其内容如下：

```
int a,b,c;//可改错误 1：缺少分号
float m,n;
char c1,c2;//增加 char 类型
char a[10];//增加数组的定义
int fibo(int a)
```

```
{/*注释部分自动去掉*/
    int i;
    if(a == 1 || a == 2){
     return 1;
     }
    for(i<15){//增加了 for 语句循环
       i++;
    }
    return fibo(a-1)+fibo(a-2);
}
int main()//注释部分自动去掉
{
    int m,n,i;
    char c;
    float ar[20];
    m=read();
    i=1;
    i++;
    --i;//加了自增和自减
    m+=i+15;//加了复合赋值运算
    while(i <= m){
        n=fibo(i);
        write(n);
        i=i+1;
    }
    return 1;
}
```

执行之后，输出结果如下：

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
 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：
    复合语句的语句部分：
      条件语句(IF_THEN)：
        条件：
          比较运算
            或运算
              比较运算
                ID： a
                INT：1
              ID： a
            INT：2
        IF 子句：
          返回语句：
            INT：1
      返回语句：
        加法
          函数调用：
            函数名：fibo
            第 1 个实际参数表达式：
              减法
                ID： a
                INT：1

          函数调用：
            函数名：fibo
            第 1 个实际参数表达式：
              减法
                ID： a
                INT：2

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：
      内部变量定义：
        类型： int
        变量名：
          m
          n
          i
    复合语句的语句部分：
```

```
表达式语句：
    赋值
        ID： m
        函数调用：
            函数名：read
表达式语句：
    赋值
        ID： i
        INT：1
条件语句(WHILE)：
    条件：
        比较运算
            ID： i
            ID： m
    WHILE 子句：
        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：
                    赋值
                        ID： n
                        函数调用：
                            函数名：fibo
                            第 1 个实际参数表达式：
                                INT：1

                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：
                    赋值
                        ID： i
                        加法
                            ID： i
                            INT：1
返回语句：
    INT：1
```

2.6 小结

虽然是第一次实验，但是第一次接触到这个项目的时候，其实是茫然的。完

全不知道自己的方向在哪里，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，也不知道学习的时候 LR（1）文法在实验中是如何体现出来的。但是我认为在做实验的过程中，渐渐明白了词法分析与语法分析的实际步骤，而我们学习的理论是建立文法的一个发展过程。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步骤。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。

期间我基本学习完了那本 flex 和 bison 的书籍，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自增、自减等其它运算。

在写程序的过程中，不要想着一次性将所有的规则都写完了再去测试，先为自己准备一些小的简单的程序，试着先将这些检测出来，再去构思完整的情况。

3 实验二 语义分析

3.1 语义表示方法描述

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

2）建立符号表，并在合适的分析位置上进行输出符号表。

3.2 符号表结构

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

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

符号表定义如下：

```
//符号表,是一个顺序栈,index 初值为 0
typedef struct symboltable{
  struct symbol symbols[MAXLENGTH];
  int index;
}SYMBOLTABLE;
```

其中 symbol 的结构为：

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

3.3 错误类型定义

1）错误类型 1：变量在使用时未定义

2）错误类型 2：函数在调用时为经定义

3）错误类型 3：变量出现重复定义或变量域前面定义过的其它语法结构重名

4）错误类型 4：函数出现重复定义

5）错误类型 5：赋值号两边的表达式类型不匹配

6）错误类型 6：赋值号左边只有一个右值的表达式

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

8）错误类型 8：return 语句的返回类型域函数定义的返回类型不匹配

9）错误类型 9：函数调用时实参或形参的数目或类型不匹配。

3.4 语义分析实现技术

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

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

int Semantic_Analysis(struct node* T,int type,int level,char flag,int command)
{
   int type1,type2;
   if(T){
      switch(T->kind){
         case EXT_DEF_LIST:
            Semantic_Analysis(T->ptr[0],type,level,flag,command);
            Semantic_Analysis(T->ptr[1],type,level,flag,command);
            break;
         case EXT_VAR_DEF://外部变量声明
            type=Semantic_Analysis(T->ptr[0],type,level,flag,command);
            Semantic_Analysis(T->ptr[1],type,level,flag,command);
            break;
         case ARRAY_DEF:
            type = Semantic_Analysis(T->ptr[0],type,level,flag,command);
            Semantic_Analysis(T->ptr[1],type,level,flag,command);
            break;
         case ARRAY_DEC:
            flag = 'A';//Array
            strcpy(new_table.symbols[new_table.index].name,T->type_id);
            new_table.symbols[new_table.index].level=level;
            new_table.symbols[new_table.index].type=type;
            new_table.symbols[new_table.index].flag=flag;
            new_table.index++;
            break;
         case TYPE:
            return T->type;
         case EXT_DEC_LIST:
```

```
                flag='V';
                Semantic_Analysis(T->ptr[0],type,level,flag,command);
                Semantic_Analysis(T->ptr[1],type,level,flag,command);
                break;
            case ID://检测新的变量名是否唯一
                i=0;
                while(new_table.symbols[i].level!=level&&i<new_table.index)//转到相
同作用域
                    i++;
                if(command==0){//定义变量
                    while(i<new_table.index){
                        if(strcmp(new_table.symbols[i].name,T->type_id)==0 && new_tabl
e.symbols[i].flag==flag){
                            if(flag=='V')
                                printf("ERROR！第%d 行：全局变量中出现相同变量
名%s\n",T->pos,T->type_id);
                            else if(flag=='F')
                                printf("ERROR！第%d 行：函数定义中出现了相同的函数
名%s\n",T->pos,T->type_id);
                            else if(flag=='T')
                                printf("ERROR！第%d 行：局部变量中出现了相同的变量
名%s\n",T->pos,T->type_id);
                            else
                                printf("ERROR！第%d 行：函数参数中中出现了相同的变
量名%s\n",T->pos,T->type_id);
                            return 0;
                        }
                        i++;
                    }
                    strcpy(new_table.symbols[new_table.index].name,T->type_id);
                    new_table.symbols[new_table.index].level=level;
                    new_table.symbols[new_table.index].type=type;
                    new_table.symbols[new_table.index].flag=flag;
                    new_table.index++;
                }
                else{//使用变量
                    i=new_table.index-1;
                    while(i>=0){
                        if(strcmp(new_table.symbols[i].name,T->type_id)==0&&(new_table
.symbols[i].flag=='V'||new_table.symbols[i].flag=='T')){
                            return new_table.symbols[i].type;
                        }
                        i--;
                    }
```

```
                    if(i<0){
                      printf("ERROR！第%d 行：变量名%s 未定义
\n",T->pos,T->type_id);
                    }
                  }
                  break;
              case FUNC_DEF://函数声明
                  type=Semantic_Analysis(T->ptr[0],type,level+1,flag,command);
                  Semantic_Analysis(T->ptr[1],type,1,flag,command);
                  Semantic_Analysis(T->ptr[2],type,1,flag,command);
                  break;
              case FUNC_DEC:
                  strcpy(new_table.symbols[new_table.index].name,T->type_id);
                  new_table.symbols[new_table.index].level=0;
                  new_table.symbols[new_table.index].type=type;
                  new_table.symbols[new_table.index].flag='F';
                  new_table.index++;
                  counter=0;
                  Semantic_Analysis(T->ptr[0],type,level,flag,command);//函数形参
                  new_table.symbols[new_table.index - counter - 1].paramnum=counter;
                  break;
              case PARAM_LIST:
                  counter++;
                  Semantic_Analysis(T->ptr[0],type,level,flag,command);
                  Semantic_Analysis(T->ptr[1],type,level,flag,command);
                  break;
              case PARAM_DEC:
                  flag='P';
                  type=Semantic_Analysis(T->ptr[0],type,level+1,flag,command);
                  Semantic_Analysis(T->ptr[1],type,level,flag,command);
                  break;
              case COMP_STM:
                  flag='T';
                  command=0;
                  new_scope.TX[new_scope.top]=new_table.index;
                  new_scope.top++;
                  Semantic_Analysis(T->ptr[0],type,level,flag,command);//分析定义列表
                  command=1;
                  Semantic_Analysis(T->ptr[1],type,level+1,flag,command);//分析语句列
表
                  new_table.index=new_scope.TX[new_scope.top-1];
                  new_scope.top--;
                  if (new_scope.top == 0)
                    DisplaySymbolTable();
```

```
    break;
case DEF_LIST:
    Semantic_Analysis(T->ptr[0],type,level,flag,command);
    Semantic_Analysis(T->ptr[1],type,level,flag,command);
    break;
case VAR_DEF:
    type=Semantic_Analysis(T->ptr[0],type,level+1,flag,command);
    Semantic_Analysis(T->ptr[1],type,level,flag,command);
    break;
case DEC_LIST:
    Semantic_Analysis(T->ptr[0],type,level,flag,command);
    Semantic_Analysis(T->ptr[1],type,level,flag,command);
    break;
case STM_LIST:
    Semantic_Analysis(T->ptr[0],type,level,flag,command);//第一个语句
    Semantic_Analysis(T->ptr[1],type,level,flag,command);//其他语句
    break;
case EXP_STMT:
    Semantic_Analysis(T->ptr[0],type,level,flag,command);
    break;
case RETURN:
    Semantic_Analysis(T->ptr[0],type,level,flag,command);
    break;
case IF_THEN:
case WHILE:
case FOR:
    Semantic_Analysis(T->ptr[0],type,level,flag,command);
    Semantic_Analysis(T->ptr[1],type,level,flag,command);
    break;
case IF_THEN_ELSE:
    Semantic_Analysis(T->ptr[0],type,level,flag,command);
    Semantic_Analysis(T->ptr[1],type,level,flag,command);
    Semantic_Analysis(T->ptr[2],type,level,flag,command);
    break;
case ASSIGNOP:
case OR:
case AND:
case RELOP:
case PLUS:
case MINUS:
case STAR:
case DIV:
case COMADD:
case COMSUB:
```

```
            type1=Semantic_Analysis(T->ptr[0],type,level,flag,command);
            type2=Semantic_Analysis(T->ptr[1],type,level,flag,command);
            if(type1==type2)
              return type1;
            break;
          case AUTOADD_L:
          case AUTOSUB_L:
          case AUTOADD_R:
          case AUTOSUB_R:
            Semantic_Analysis(T->ptr[0],type,level,flag,command);
            break;
          case INT:
            return INT;
          case FLOAT:
            return FLOAT;
          case CHAR:
            return CHAR;
          case FUNC_CALL:
            j=0;
            while(new_table.symbols[j].level==0&&j<new_table.index){
              if(strcmp(new_table.symbols[j].name,T->type_id)==0){
                if(new_table.symbols[j].flag!='F')
                  printf("ERROR！第%d 行：函数名%s 在符号表中定义为变量
\n",T->pos,T->type_id);
                break;
              }
              j++;
            }
            if(new_table.symbols[j].level==1||j==new_table.index){
              printf("ERROR！第%d 行：函数%s 未定义\n",T->pos,T->type_id);
              break;
            }
            type=new_table.symbols[j+1].type;
            counter=0;
            Semantic_Analysis(T->ptr[0],type,level,flag,command);//分析参数
            if(new_table.symbols[j].paramnum!=counter)
              printf("ERROR！第%d 行：函数调用%s 参数个数不匹配
\n",T->pos,T->type_id);
            break;
          case ARGS:
            counter++;
            t=Semantic_Analysis(T->ptr[0],type,level,flag,command);
            if(type!=t)
              printf("ERROR！第%d 行：函数调用的第%d 个参数类型不匹配
```

```
\n",T->pos,counter);
                type=new_table.symbols[j+counter+1].type;
                Semantic_Analysis(T->ptr[1],type,level,flag,command);
                break;
            }
        }
        return 0;
    }
```

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

```
    void DisplaySymbolTable()
    {
        int i;
        printf("\t\t***Symbol Table***\n");
        printf("-------------------------------------------------\n");
        printf("%s\t%s\t%s\t%s\t%s\t%s\n","Index","Name","Level","Type","Flag","Par
am_num");
        printf("-------------------------------------------------\n");
        for(i=0;i<new_table.index;i++){
            printf("%d\t",i);
            printf("%s\t",new_table.symbols[i].name);
            printf("%d\t",new_table.symbols[i].level);
            if(new_table.symbols[i].type==INT)
                printf("%s\t","int");
            else if(new_table.symbols[i].type==FLOAT)
                printf("%s\t","float");
            else
                printf("%s\t","char");
            printf("%c\t",new_table.symbols[i].flag);
            if(new_table.symbols[i].flag=='F')
                printf("%d\n",new_table.symbols[i].paramnum);
            else
                printf("\n");
        }
        printf("-------------------------------------------------\n");
        printf("\n");
    }
```

3.5 语义分析结果

语义分析的源文件为 test2.c，其内容如下：

```
    int a,b,c;
    float m,n;
    char c1,c2;
    char h[10];
```

```
float a,b;//全局变量中出现相同变量名

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

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

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

int h1(int a, int a){}//出现了相同函数参数

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

float h1(){}//重复的函数名
```

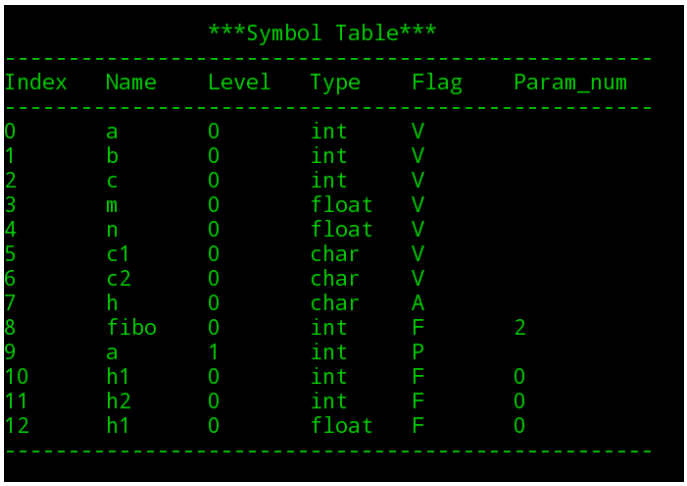
运行结果如下图所示：

```
ERROR! 第21行： 变量名j未定义
ERROR! 第22行： 函数haha未定义
ERROR! 第23行： 函数调用fibo参数个数不匹配
ERROR! 第24行： 函数调用的第1个参数类型不匹配
                ***Symbol Table***
-------------------------------------------------
Index   Name    Level   Type    Flag    Param_num
-------------------------------------------------
0       a       0       int     V
1       b       0       int     V
2       c       0       int     V
3       m       0       float   V
4       n       0       float   V
5       c1      0       char    V
6       c2      0       char    V
7       h       0       char    A
8       fibo    0       int     F       1
9       a       1       int     P
-------------------------------------------------

ERROR! 第29行： 函数参数中中出现了相同的变量名a
ERROR! 第29行： 函数参数中中出现了相同的变量名a
```

图 3.1 错误类型检测

符号表内容如下图所示：



Symbol Table

Index	Name	Level	Type	Flag	Param_num
0	a	0	int	V	
1	b	0	int	V	
2	c	0	int	V	
3	m	0	float	V	
4	n	0	float	V	
5	c1	0	char	V	
6	c2	0	char	V	
7	h	0	char	A	
8	fibo	0	int	F	2
9	a	1	int	P	
10	h1	0	int	F	0
11	h2	0	int	F	0
12	h1	0	float	F	0

图 3.2 符号表输出

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，所以实现起来相对比较简单，一开始没有清晰理解 SymbolTable 和 Symbol_Scope_TX 的作用与区别，到后来明确了 SymbolTable 是装入文法符号的，Symbol_Scope_TX 是用来标识作用域的。在退出对应的饿作用域后，Symbol_Scope_TX 相应的也进行退栈操作。

其次，本次实验使用了构造最简单的符号表—顺序表，而我通过查阅资料发现编译器实现主要是靠多表结构来组织，并且是以链表来管理元组。我认为他们这样做是对识别的程序容量大小有着充分的考量，并且多表的实现更有利于并行识别多文件工程，会在速度上有着更大的提升。

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。

4 实验三 中间代码生成

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

1.基本表达式翻译模式

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

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

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

4）如果 Exp 产生了算数运算表达式 Exp PLUS Exp，则先对 Exp 进行翻译（运算结果储存在临时变量 t1 中），再对 Exp 进行翻译（运算结果储存在临时变量 t2 中），最后生成一句中间代码 place：=t1+t2，并将刚翻译好的这三段代码合并后返回即可。使用类似的翻译模式也可以对剑法、乘法和除法进行翻译。

5）如果 Exp 产生了屈服表达式 MINUS Exp，则先对 Exp 进行翻译（运算结

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

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

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

图 4.1 基本表达式翻译模式

2.语句翻译模式

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

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

图 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_1 RELOP Exp_2	t1 = new_temp() t2 = new_temp() code1 = translate_Exp(Exp_1, sym_table, t1) code2 = translate_Exp(Exp_2, sym_table, t2) op = get_relop(RELOP); code3 = [IF t1 op t2 GOTO label_true] return code1 + code2 + code3 + [GOTO label_false]
NOT Exp_1	return translate_Cond(Exp_1, label_false, label_true, sym_table)
Exp_1 AND Exp_2	label1 = new_label() code1 = translate_Cond(Exp_1, label1, label_false, sym_table) code2 = translate_Cond(Exp_2, label_true, label_false, sym_table) return code1 + [LABEL label1] + code2
Exp_1 OR Exp_2	label1 = new_label() code1 = translate_Cond(Exp_1, label_true, label1, sym_table) code2 = translate_Cond(Exp_2, label_true, label_false, sym_table) return code1 + [LABEL label1] + code2
(other cases)	t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) code2 = [IF t1 != #0 GOTO label_true] return code1 + code2 + [GOTO label_false]

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

图 4.4 函数调用翻译模式

translate_Args(Args, sym_table, arg_list) = case Args of	
Exp	t1 = new_temp() code1 = translate_Exp(Exp, sym_table, t1) arg_list = t1 + arg_list return code1
Exp COMMA $Args_1$	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

图 4.5 函数参数的翻译模式

4.3 中间代码生成过程

1．定义 opn 结构体：

struct opn{

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

其包含类型、种类、层号、偏移量等信息。

2．定义 codenode 结构体：

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

其采用双向循环链表的方式存储中间代码。

3．定义 node 结构体：

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

其包含诸多含义：

place 记录该结点操作数在符号表中的位置序号；

type 记录该数据的类型，用于表达式计算；

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

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

code 记录中间代码序列的起始位置；

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

Snext 记录该结点的语句序列执行完后，要转移到的程序位置。

4．newAlias 函数

代码描述：

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

函数作用：生成一个新的别名。

5．newLabel 函数

代码描述：

```
char *newLabel()
{ //生成新标号
    static int k = 1;
    static char result[10];
    char s[10];
    snprintf(s, 10, "%d", k++);
    strcpy(result, "label");
    strcat(result, s);
    return result;
}
```

函数作用：生成一个新的标号。

6．newTemp 函数

代码描述：

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

函数作用：生成一个新的临时变量。

7．makeopn 函数

代码描述：

```
struct opn makeopn(struct node *T,int kind)
{
    struct opn op;
    int judge=check_var(T);
    op.kind=kind;
    op.type=symbolTable.symbols[judge].type;
    if(kind==FUNCTION)
        strcpy(op.id,symbolTable.symbols[judge].name);
    else
        strcpy(op.id,symbolTable.symbols[judge].alias);
    return op;
}
```

函数作用：建立 opn 结构体用来存储各类信息。

8．genIR 函数

代码描述：

```
struct codenode *genIR(int op, struct opn opn1, struct opn opn2, struct opn result)
{
    struct codenode *h = (struct codenode *)malloc(sizeof(struct codenode));
    h->op = op;
    h->opn1 = opn1;
    h->opn2 = opn2;
    h->result = result;
    h->next = h->prior = h;
    return h;
}
```

函数作用：生成一条 TAC 代码的结点组成的双向循环链表，返回头指针

9．genLable 函数：

代码描述：

```
struct codenode *genLabel(char *label)
{
    struct codenode *h = (struct codenode *)malloc(sizeof(struct codenode));
    h->op = LABEL;
    strcpy(h->result.id, label);
    h->next = h->prior = h;
    return h;
}
```

函数作用：生成一条标号语句，返回头指针

10．genGoto 函数：

代码描述：

```
struct codenode *genGoto(char *label)
{
    struct codenode *h = (struct codenode *)malloc(sizeof(struct codenode));
```

```
    h->op = GOTO;
    strcpy(h->result.id, label);
    h->next = h->prior = h;
    return h;
}
```

函数作用：生成 GOTO 语句，返回头指针

11．merge 函数

代码描述：

```
struct codenode *merge(int num, ...)
{
    struct codenode *h1, *h2, *p, *t1, *t2;
    va_list ap;
    va_start(ap, num);
    h1 = va_arg(ap, struct codenode *);
    while (--num > 0)
    {
        h2 = va_arg(ap, struct codenode *);
        if (h1 == NULL)
            h1 = h2;
        else if (h2)
        {
            t1 = h1->prior;
            t2 = h2->prior;
            t1->next = h2;
            t2->next = h1;
            h1->prior = t2;
            h2->prior = t1;
        }
    }
    va_end(ap);
    return h1;
}
```

函数作用：合并双向循环链表

12．prnIR 函数

代码描述：

```
void prnIR(struct codenode *head)
{
    char opnstr1[32], opnstr2[32], resultstr[32];
    struct codenode *h = head;
    do
    {
        if (h->opn1.kind == INT)
            sprintf(opnstr1, "#%d", h->opn1.const_int);
```

```
                if (h->opn1.kind == FLOAT)
                    sprintf(opnstr1, "#%f", h->opn1.const_float);
                if (h->opn1.kind == ID)
                    sprintf(opnstr1, "%s", h->opn1.id);
                if (h->opn2.kind == INT)
                    sprintf(opnstr2, "#%d", h->opn2.const_int);
                if (h->opn2.kind == FLOAT)
                    sprintf(opnstr2, "#%f", h->opn2.const_float);
                if (h->opn2.kind == ID)
                    sprintf(opnstr2, "%s", h->opn2.id);
                sprintf(resultstr, "%s", h->result.id);
                switch (h->op)
                {
                case ASSIGNOP:
                    printf("    %s := %s\n", resultstr, opnstr1);
                    break;
                case PLUS:
                case MINUS:
                case STAR:
                case DIV:
                    printf("    %s := %s %c %s\n", resultstr, opnstr1,
                            h->op == PLUS ? '+' : h->op == MINUS ? '-' : h->op ==
STAR ? '*' : '\\', opnstr2);
                    break;
                case FUNCTION:
                    printf("\nFUNCTION %s :\n", h->result.id);
                    break;
                case PARAM:
                    printf("    PARAM %s\n", h->result.id);
                    break;
                case LABEL:
                    printf("LABEL %s :\n", h->result.id);
                    break;
                case GOTO:
                    printf("    GOTO %s\n", h->result.id);
                    break;
                case JLE:
                    printf("    IF %s <= %s GOTO %s\n", opnstr1, opnstr2, resultstr);
                    break;
                case JLT:
                    printf("    IF %s < %s GOTO %s\n", opnstr1, opnstr2, resultstr);
                    break;
                case JGE:
                    printf("    IF %s >= %s GOTO %s\n", opnstr1, opnstr2, resultstr);
```

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

函数作用：输出中间代码

13. 处理函数

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

```
void id_exp(struct node *T);
void int_exp(struct node *T);
void assignop_exp(struct node *T);
void relop_exp(struct node *T);
void args_exp(struct node *T);
void op_exp(struct node *T);
void func_call_exp(struct node *T);
void not_exp(struct node *T);
void ext_var_list(struct node *T);
void ext_def_list(struct node *T);
void ext_var_def(struct node *T);
void func_def(struct node *T);
void func_dec(struct node *T);
```

```
void param_list(struct node *T);
void param_dec(struct node *T);
void comp_stm(struct node *T);
void def_list(struct node *T);
void var_def(struct node *T);
void stmt_list(struct node *T);
void if_then(struct node *T);
void if_then_else(struct node *T);
void while_dec(struct node *T);
void exp_stmt(struct node *T);
void return_dec(struct node *T);
```

4. 4 中间代码生成结果

测试的源文件为 test3.c，内容如下：

```
int fact(int n){
   int temp;
   if(n==1)
      return n;
   else{
      temp=(n*fact(n-1));
      return temp;
   }
}

int main()
{
   int result,times;
   times=read();
   for(int i=0;i<times;i++){
      int m = read();
      if( m > 1) {
         result=fact(m);
      }
      else
         result = 1;
      print(result);
   }
   return 0;
}
```

生成的中间代码如下：

```
FUNCTION fact :
PARAM var0
temp0 := #1
IF var0 == temp0 GOTO label0
```

```
GOTO label1
LABEL label0 :
RETURN var0
GOTO label2
LABEL label1 :
temp2 := #1
temp3 := var0 - temp2
ARG temp3
temp4 := CALL fact
temp5 := var0 * temp4
var1 := temp5
RETURN var1
LABEL label2 :
FUNCTION main :
temp6 := CALL read
var3 := temp6
temp7 := #0
var4 := temp7
LABEL label3 :
IF var4 < var3 GOTO label4
GOTO label5
LABEL label4 :
temp9 := CALL read
var5 := temp9
temp10 := #1
IF var5 > temp10 GOTO label6
GOTO label7
LABEL label6 :
ARG var5
temp12 := CALL fact
var2 := temp12
GOTO label8
LABEL label7 :
temp13 := #1
var2 := temp13
LABEL label8 :
ARG var2
CALL print
temp15 := #1
var4 := var4 + temp15
GOTO label3
LABEL label5 :
temp16 := #0
RETURN temp16
```

4.5 小结

编译器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。实际上，中层次的中间代码是集中 IR 中最难设计的一种。在这个层次上，变量和联合变量可能已经有了区分，控制流也可能已经被简化为无条件跳转、有条件跳转、函数调用和函数返回四种操作。低层次的代码中会加入寄存器的相关信息，大部分代码和目标代码中的指令中红往往存在着一一对应的关系，即使没有对应，而这之间也属于一次指令就能完成任务，所以相对中间层次的目标代码会更加简单。这是本次实验未能清楚考虑清楚的地方。

第三次实验应该是最难的一次，我改动了非常多的地方，但其实大部分逻辑都是沿用一二次的代码，此次工程实在是有带你多，阅读了大量资料和别人的代码。但 github 上编译器的代码往往很复杂，有时候根本读不完。最终，经过不断尝试最终成功调试出。

5 实验四 目标代码生成

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

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。

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","s

```
5","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{//没找到
        vector<string> keys;
        for (auto & it : table)//已经分配寄存器的变量 key
          keys.emplace_back(it.first);

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

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

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>=y GOTO z	lw $t1, x 的偏移量($sp) lw $t2, y 的偏移量($sp) bge $t1,$t2,z
IF x<y GOTO z	lw $t1, x 的偏移量($sp) lw $t2, y 的偏移量($sp) ble $t1,$t2,z
IF x<=y GOTO z	lw $t1, x 的偏移量($sp) lw $t2, y 的偏移量($sp) blt $t1,$t2,z
X:=CALL f	

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

```
string translate(string temp_str){
  //将每行 string 按空格存成数组
  vector<string> line;
  string temp_res;
```

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

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

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

5.4 目标代码生成结果

为了测试翻译程序的正确性，我使用了两个递归程序的中间代码进行测试，最后都成功在 Spim 机器上运行。此处为节省空间，只输出目标代码 1 的生成结果。目标代码 1 生成结果如下：

```
.data
_prompt: .asciiz "Enter an integer:"
_ret: .asciiz "\n"
.globl main
.text
read:
    li $v0,4
    la $a0,_prompt
    syscall
    li $v0,5
    syscall
    jr $ra

print:
```

```
        li $v0,1
        syscall
        li $v0,4
        la $a0,_ret
        syscall
        move $v0,$0
        jr $ra
fact:
li $t1,1
beq $a0,$t1,label0
j label1
label0:
move $v0,$a0
jr $ra
j label2
label1:
li $t1,1
sub $t2,$a0,$t1
move $t0,$a0
move $a0,$t2
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 fact
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 $t1 $v0
mul $t2,$a0,$t1
move $t1,$t2
move $v0,$t1
jr $ra
label2:
main:
addi $sp,$sp,-4
sw $ra,0($sp)
```

```
jal read
lw $ra,0($sp)
move $t2,$v0
addi $sp,$sp,4
move $t3,$t2
li $t2,0
move $t4,$t2
label3:
blt $t4,$t3,label4
j label5
label4:
addi $sp,$sp,-4
sw $ra,0($sp)
jal read
lw $ra,0($sp)
move $t2,$v0
addi $sp,$sp,4
move $t5,$t2
li $t2,1
bgt $t5,$t2,label6
j label7
label6:
move $t0,$a0
move $a0,$t5
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 fact
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 $t2 $v0
move $t6,$t2
j label8
label7:
li $t2,1
```

```
move $t6,$t2
label8:
move $t0,$a0
move $a0,$t6
addi $sp,$sp,-4
sw $ra,0($sp)
jal print
lw $ra,0($sp)
addi $sp,$sp,4
li $t2,1
add $t4,$t4,$t2
j label3
label5:
li $t2,0
move $v0,$t2
jr $ra
```

5.5 目标代码运行结果

目标代码运行结果如图 5-1 所示。

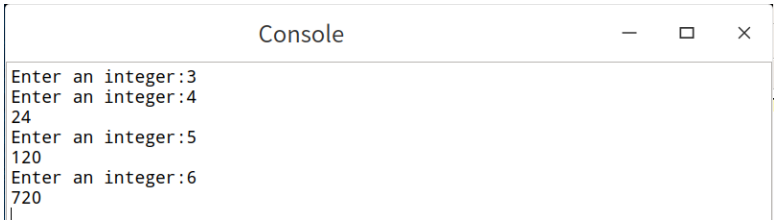


图 5.1 SPIM Simulator

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，然后自己写几个建安的 MIPS32 汇编送到 SPIM simulator 运行，以确定自己是否已经清楚 MIPS32 代码如何书写。

完成实验四的第一步是确定指令选择机智以及寄存器分配算法。指令选择算法比较简单，其功能甚至可以由中间代码的大一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，开始时可以无视与函数有关的调用的 ARG、PARAM、RETURN、CALL 语句，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。处理 ARG 和 PARAM 要注意实参和形参的顺序不要搞错，另外计算实参时如果没有使用$fp，也要注意使用各临时变量相对于$sp 偏移量的修改。如果调用序列出现问题，还需要使用 SPIM 机器进行调试。

6 总结

四次编译原理实验任务最终完成了一个小型的编译器，虽然还有很多功能不支持，但是在这过程中对于词法分析、语法分析、语义分析、中间代码生成和目标代码生成的原理有了更加深入的理解。我完成的任务梳理如下：

- 运用 lex 工具编写词法分析运用的正则表达式规则，实现程序的词法分析。
- 运用 bison 编写语法分析部分，并将在其中加入了新的语法规则以及错误处理规则，实现了源代码的语法分析。
- 设计 display 函数、mknode 函数在分析中建立抽象语法树
- 在语法分析和词法分析的基础上，设计 semantic_Analysis 函数，生成符号表，利用符号表和语法树成功进行了语义分析，并能够辨别出不同类型的错误，最后输出符号表。
- 在语义分析的时候就进行中间代码的生成，提高编译程序的效率
- 利用生成的中间代码生成最终的 MIPS32 目标代码

在实验中，我有如下的心得体会：

编译原理实验是一个具有挑战性的实验，整个过程不断在巩固课上没有重点讲的内容，让我在实践当中学习到了更多知识。同时，很锻炼我的编码能力，此次实验我学会了如何将复杂问题拆分为简单问题，从简单问题入手逐步解决困难问题。以前一直不是很清楚正则式的用法，但是这次实验明白了正则式的功能强大与方便，还学会了很多正则式的语法。

实验的核心部分是中间代码生成，但是实验指导书介绍的很有限，希望老师以后多给些资料阅读，其实看完指导书仅仅是算入门，对于真正上手写代码其实还有很长距离。第三次实验花费了很多精力，从从头开始翻阅理论书了解原理，到发现原理和实现的距离，再读各类编译器的代码，再不断思考之后才写出来。的确，前期的积累是很痛苦的，但是一旦理解开始，就非常快，几百行代码很快就出来了，因为逻辑是清晰的。

编译原理的知识是整个计算机学科中的核心之一，我们学习的仅仅是入门级别的知识，我阅读的大量代码，其实现难度和规模都远在我之上，可见我们的水平还未达到最好的状态。

最后，感谢网络空间安全学院，感谢《编译原理》课程组，感谢指导教师刘铭老师，在本次实验中都给予了我十分大的帮助。今后我也会带着收获的知识与教诲，更加刻苦努力地学习，实现人生价值。

参考文献

[1] 王生原,董渊,张素琴,吕映之,蒋维杜.编译原理,清华大学出版社,2019.3

[2] 刘铭,骆婷.编译原理实验指导教程,华中科技大学网络空间安全学院,2019.10