

**课程实验报告**

**题目： miniC语言编译器**

**课程名称： 编译原理**

**专业班级： CSEE1701**

**学 号： U201714487**

**姓 名： 蔡靖涛**

**指导教师： 徐丽萍,祝建华**

**报告日期： 2020/06/25**

**计算机科学与技术学院**

**目录**

[1概述 1](#_Toc43913010)

[2系统描述 2](#_Toc43913011)

[2.1自定义语言概述及单词文法描述 2](#_Toc43913012)

[2.2语句文法描述 4](#_Toc43913013)

[2.3 符号表结构定义 7](#_Toc43913014)

[2.4 错误类型码定义 8](#_Toc43913015)

[2.5中间代码结构定义 9](#_Toc43913016)

[2.6 目标代码指令集选择 9](#_Toc43913017)

[3系统设计与实现 12](#_Toc43913018)

[3.1词法语法分析器（实验一） 12](#_Toc43913019)

[3.1.1词法分析器 12](#_Toc43913020)

[3.3.2语法分析器 14](#_Toc43913021)

[3.2编译程序符号表结构 18](#_Toc43913022)

[3.3语义分析（实验二） 18](#_Toc43913023)

[3.4编译程序报错功能 19](#_Toc43913024)

[3.5中间代码生成功能（实验三） 24](#_Toc43913025)

[3.6汇编代码生成功能（实验四） 29](#_Toc43913026)

[4系统测试与评价 30](#_Toc43913027)

[4.1 测试用例及测试结果 30](#_Toc43913028)

[4.2 系统的优点 41](#_Toc43913029)

[4.3 系统的缺点 41](#_Toc43913030)

[5实验小结或体会 42](#_Toc43913031)

# 1概述

本次实验需要完成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。具体任务是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。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。高级语言建议大家选用 C语言的简单集合SC语言。

开发环境：Windows 10专业版系统，Visual Studio Code 编辑器

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.l 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 MIPS汇编指令，生成的指令最终能在 QtSpim 机器上成功运行并得到正确结果。

# 2系统描述

## 2.1自定义语言概述及单词文法描述

这次我自定义的mini C语言大致上类似C语言，具有基本类型int、float、char、bool等， 具有关键字if，else，while，for等，可以完成基本的判断逻辑和循环逻辑，可以完成基本的定义、计算、赋值、引用等操作。

我的mini C中的单词根据类型可以分为6类：标识符、关键字、运算符、界符、常量以及注释，具体每种单词的文法在lex.l文件中定义，单词定义部分如下表所示：

|  |  |  |
| --- | --- | --- |
| 单词符号说明 | 单词种类码 | 正则表达式 |
| {char} | CHAR | \'[0-9a-zA-Z]\' |
| {int} | INT | [0-9]+ |
| {float} | FLOAT | ([0-9]\*\.[0-9]+)|([0-9]+\.) |
| “char” | TYPE |  |
| “int” | TYPE |  |
| “float” | TYPE |  |
| break | BREAK |  |
| continue | CONTINUE |  |
| return | RETURN |  |
| if | IF |  |
| else | ELSE |  |
| while | WHILE |  |
| for | FOR |  |
| id | ID | [A-Za-z][A-Za-z0-9]\* |
| “;” | SEMI |  |
| “,” | COMMA |  |
| ">"|"<"|">="|"<="|"=="|"!=" | RELOP |  |
| “=” | ASSIGNOP |  |
| “+” | PLUS |  |
| “-“ | MINUS |  |
| “\*” | STAR |  |
| “/” | DIV |  |
| “++” | SELFADD |  |
| “--” | SELFDEC |  |
| “+=” | ADD\_ASSIGNOP |  |
| “-=” | MINUS\_ASSIGNOP |  |
| “\*=” | STAR\_ASSIGNOP |  |
| “/=” | DIV\_ASSIGNOP |  |
| “&&” | AND |  |
| “||” | OR |  |
| "." | DOT |  |
| “!” | NOT |  |
| “(” | LP |  |
| “)” | RP |  |
| "[" | LB |  |
| "]" | RB |  |
| “{” | LC |  |
| “}” | RC |  |
| [\n] |  |  |
| [\r\t] |  |  |
| "//" | 单行注释，无种类码 |  |
| "/\*" | 多行注释，无种类码 |  |

表2.1：lex.l文件中单词的文法定义

注：多行注释将转义为一个自定义函数multipleLinesComment()，实现如下：

int multipleLinesComment()

{

char c,c1;

loop:

while((c=input())!='\*'&&c!=0){}

if((c1 = input())!='/'&&c!=0)

{

unput(c1);

goto loop;

}

return 0;

}

至此，单词定义已经说明完毕。

## 2.2语句文法描述

对语句的文法定义如下代码所示：

|  |
| --- |
| program: ExtDefList {/\*display($1, 0);\*/udisplay($1, 0);semantic\_Analysis0($1);}  ;  ExtDefList: {$$=NULL;}  | ExtDef ExtDefList {$$=mknode(EXT\_DEF\_LIST,$1,$2,NULL,yylineno);}  ;  ExtDef: Specifier ExtDecList SEMI {$$=mknode(EXT\_DEF\_VAR,$1,$2,NULL,yylineno);}  |Specifier FuncDec CompSt {$$=mknode(EXT\_DEF\_FUNC,$1,$2,$3,yylineno);}  | error SEMI {$$=NULL; }  // |Specifier SEMI {}  ;  ExtDecList: VarDec {$$=$1;}  |VarDec COMMA ExtDecList {$$=mknode(EXT\_VARDEC\_LIST,$1,$3,NULL,yylineno);}  ;  //Specifiers  Specifier: TYPE {$$=mknode(TYPE,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);$$->type=!strcmp($1,"int")?INT:(!strcmp($1,"float")?FLOAT:CHAR);}  ;  //Declarators  VarDec: ID LB INT RB {$$=mknode(ARRAY,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);$$->size=$3;}  |ID {$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}  ;  FuncDec: ID LP VarList RP {$$=mknode(FUNC\_DEC,$3,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}  |ID LP RP {$$=mknode(FUNC\_DEC,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}  ;  VarList: ParamDec {$$=mknode(FUNC\_PARAM\_LIST,$1,NULL,NULL,yylineno);}  | ParamDec COMMA VarList {$$=mknode(FUNC\_PARAM\_LIST,$1,$3,NULL,yylineno);}  ;  ParamDec: Specifier VarDec {$$=mknode(FUNC\_PARAM\_DEC,$1,$2,NULL,yylineno);}  ;  //Statements  CompSt: LC DefList StmtList RC {$$=mknode(COMP\_STM,$2,$3,NULL,yylineno);}  ;  StmtList: {$$=NULL; }  | Stmt StmtList {$$=mknode(COMPSTM\_LIST,$1,$2,NULL,yylineno);}  ;  Stmt: Exp SEMI {$$=mknode(COMPSTM\_EXP,$1,NULL,NULL,yylineno);}  | CompSt {$$=$1;}  | RETURN Exp SEMI {$$=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 {$$=mknode(WHILE,$3,$5,NULL,yylineno);}  | FOR LP ExpForCond RP Stmt {$$=mknode(FOR,$3,$5,NULL,yylineno);}  ;  ExpForCond: ExpFor1 SEMI ExpFor2 SEMI ExpFor3 {$$=mknode(FOR\_CONDITION,$1,$3,$5,yylineno);}  ;  //Local Definitions  DefList: {$$=NULL; }  | Def DefList {$$=mknode(DEF\_LIST,$1,$2,NULL,yylineno);}  ;  Def: Specifier DecList SEMI {$$=mknode(VAR\_DEF,$1,$2,NULL,yylineno);}  ;  DecList: Dec {$$=mknode(VARDEC\_LIST,$1,NULL,NULL,yylineno);}  | Dec COMMA DecList {$$=mknode(VARDEC\_LIST,$1,$3,NULL,yylineno);}  ;  Dec: VarDec {$$=$1;}  | VarDec ASSIGNOP Exp {$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,"ASSIGNOP");}  ;  //Expressions  Exp: Exp ASSIGNOP Exp {$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,"ASSIGNOP");}  | Exp AND Exp {$$=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($$->type\_id,$2);}  | Exp PLUS Exp {$$=mknode(PLUS,$1,$3,NULL,yylineno);strcpy($$->type\_id,"PLUS");}  | Exp MINUS Exp {$$=mknode(MINUS,$1,$3,NULL,yylineno);strcpy($$->type\_id,"MINUS");}  | Exp STAR Exp {$$=mknode(STAR,$1,$3,NULL,yylineno);strcpy($$->type\_id,"STAR");}  | Exp DIV Exp {$$=mknode(DIV,$1,$3,NULL,yylineno);strcpy($$->type\_id,"DIV");}  | LP Exp RP {$$=$2;}  | MINUS Exp %prec UMINUS {$$=mknode(UMINUS,$2,NULL,NULL,yylineno);strcpy($$->type\_id,"UMINUS");}  | NOT Exp {$$=mknode(NOT,$2,NULL,NULL,yylineno);strcpy($$->type\_id,"NOT");}  | VarDec SELFADD {$$=mknode(SELFADD,$1,NULL,NULL,yylineno);strcpy($$->type\_id,"SELFADD");}  | SELFADD VarDec {$$=mknode(SELFADD,NULL,$2,NULL,yylineno);strcpy($$->type\_id,"SELFADD");}  | VarDec SELFDEC {$$=mknode(SELFDEC,$1,NULL,NULL,yylineno);strcpy($$->type\_id,"SELFDEC");}  | SELFDEC VarDec {$$=mknode(SELFDEC,NULL,$2,NULL,yylineno);strcpy($$->type\_id,"SELFDEC");}  | VarDec ADD\_ASSIGNOP Exp {$$=mknode(ADD\_ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,"ADD\_ASSIGNOP");}  | VarDec MINUS\_ASSIGNOP Exp {$$=mknode(MINUS\_ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,"MINUS\_ASSIGNOP");}  | VarDec STAR\_ASSIGNOP Exp {$$=mknode(STAR\_ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,"STAR\_ASSIGNOP");}  | VarDec DIV\_ASSIGNOP Exp {$$=mknode(DIV\_ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,"DIV\_ASSIGNOP");}  | ID LP Args RP {$$=mknode(FUNC\_CALL,$3,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}  | ID LP RP {$$=mknode(FUNC\_CALL,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}  | ID LB Exp RB {$$=mknode(ARRAY\_CALL,$3,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}  | VarDec {$$=$1;}  | INT %prec LOWER\_THAN\_RB {$$=mknode(INT,NULL,NULL,NULL,yylineno);$$->type\_int=$1;$$->type=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;}  | BREAK {$$=mknode(\_BREAK,NULL,NULL,NULL,yylineno);}  | CONTINUE {$$=mknode(\_CONTINUE,NULL,NULL,NULL,yylineno);}  ;  Args: Exp COMMA Args {$$=mknode(ARGS,$1,$3,NULL,yylineno);}  | Exp {$$=mknode(ARGS,$1,NULL,NULL,yylineno);}  ;  ExpFor1:{$$=NULL; }  |ExpFor1List {$$=$1;}  ;  ExpFor1Atom:  Specifier Dec {$$=mknode(VAR\_DEF,$1,$2,NULL,yylineno);}  |VarDec ASSIGNOP Exp {$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,"ASSIGNOP");}  ;  ExpFor1List: ExpFor1Atom{$$=$1;}  |ExpFor1Atom COMMA ExpFor1List {$$=mknode(EXP\_FOR1\_LIST,$1,$3,NULL,yylineno);}  ;  ExpFor2: {$$=NULL; }  |VarDec RELOP Exp {$$=mknode(RELOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,$2);}  ;  ExpFor3: {$$=NULL;}  | ExpFor3List {$$=$1;}  ;  ExpFor3List: Exp {$$=$1;}  | Exp COMMA ExpFor3List {$$=mknode(EXP\_FOR3\_LIST,$1,$3,NULL,yylineno);}  ; |

## 2.3 符号表结构定义

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

符号表的属性列包括：变量名、别名、层号、类型、标记、形参个数以及作用域 / 偏移量。各属性具体作用如下：（别名在实验2后续的实验步骤中将会用到，后续还会加入偏移量等信息）

1. 变量名：记录变量的定义名称。
2. 别名：变量别名，解决不同作用域的重名问题。
3. 层号：用来记录符号在所在的作用域的层级的，每当程序进入一个新函数时层号置为0（外部定义也为0），每当再进入一个复合语句时，层号就加1，退出时层号就减1。
4. 类型：记录变量的数据类型或者是函数的返回值类型。
5. 标记：符号的标记，便于各种判断。具体包括如下几种：‘F’(函数)、’V’(变量)、’P’(参数)、’T’(临时变量)
6. 形参个数：记录函数的形参个数（非函数则不考虑）
7. 作用域：说明当前大作用域的名字（外部/函数名）（仅实验2）
8. 偏移量：说明当前变量在作用域内的偏移字节数（实验2后）

## 2.4 错误类型码定义

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

语义分析阶段负责检查各种语义错误，主要包括：

（1）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）赋值号左边不是左值表达式；

（11）对非左值表达式进行自增、自减运算；

（12） 类型不匹配。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；

（13）函数返回值类型与函数定义的返回值类型不匹配；

（14）函数没有返回语句（当函数返回值类型不是void时）；

（15）break语句不在循环语句或switch语句中；

（16）continue语句不在循环语句中；

## 2.5中间代码结构定义

选用四元式作为中间代码的形式，各种定义如表2.2所示：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **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 <= y GOTO z | 小于等于转移 | JLE | X | Y | Z |
| IF x < y GOTO z | 小于转移 | JLT | X | Y | Z |
| IF x >= y GOTO z | 大于等于转移 | JGE | X | Y | Z |
| IF x > y GOTO z | 大于转移 | JGT | X | Y | Z |
| IF x == y GOTO z | 等于转移 | EQ | X | Y | Z |
| IF x != y GOTO z | 不等转移 | NEQ | 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 |

表2.2：中间代码定义

## 2.6 目标代码指令集选择

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

|  |  |
| --- | --- |
| 中间代码形式 | 目标代码形式 |
| LABEL name | Label\_name: #直接打印标号名 |
| Return x | move $v0, x的偏移量($sp)  jr $ra |
| 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 |
| 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) |
| 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)  blt $t1,$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 | move $t0,$sp  addi $sp, $sp, -(f的偏移量)  sw $ra,0($sp)  lw $t1, 参数的偏移量($sp)  …（取出全部参数）  move $t3,$t1  sw $t3,f形参偏移量($sp)  …（对全部形参存入）  jal f  lw $ra,0($sp)  addi $sp,$sp,(f的偏移量)  sw $v0,返回值偏移量($sp) |

表2.3：中间代码与目标代码的对应关系

# 3系统设计与实现

## 3.1词法语法分析器（实验一）

### 3.1.1词法分析器

miniC中的单词分为6类：关键字、标识符、运算符、常量、界符、注释。利用flex设计此法分析器，只需要根据每一类单词的特征写出其对应的正规式，flex即可自动生成对应的词法分析程序。这里需要注意的是，关键字的规则要写在标识符的前面，因为flex是前面的规则先匹配，如果标识符的规则写在了前面，那么所有的关键字都会被识别为标识符。

（1）关键字

关键字是需要完全匹配，书写关键字的正则表达式只需要列举即可

|  |
| --- |
| "int" {strcpy(yylval.type\_id, yytext);return TYPE;}  "float" {strcpy(yylval.type\_id, yytext);return TYPE;}  "char" {strcpy(yylval.type\_id, yytext);return TYPE;}  "break" {return BREAK;}  "continue" {return CONTINUE;}  "return" {return RETURN;}  "if" {return IF;}  "else" {return ELSE;}  "while" {return WHILE;}  "for" {return FOR;} |

（2）标识符

标识符是以字母开头的，字母数字。

|  |
| --- |
| id [A-Za-z][A-Za-z0-9]\*  {id} {strcpy(yylval.type\_id, yytext); return ID;} |

（3）常量

常量包括整形常量、浮点常量、字符常量以及字符串常量。

|  |
| --- |
| int [0-9]+  float ([0-9]\*\.[0-9]+)|([0-9]+\.)  char \'[0-9a-zA-Z]\'  {int} {yylval.type\_int=atoi(yytext); return INT;}  {float} {yylval.type\_float=atof(yytext); return FLOAT;}  {char} {yylval.type\_char=yytext[1]; return CHAR;} |

（4）界符

|  |
| --- |
| ";" {return SEMI;}  "," {return COMMA;}  "(" {return LP;}  ")" {return RP;}  "[" {return LB;}  "]" {return RB;}  "{" {return LC;}  "}" {return RC;}  [\n] {yycolumn=1;}  [ \r\t] {} |

（5）运算符

运算符包括算数运算符以及逻辑运算符等。

|  |
| --- |
| ">"|"<"|">="|"<="|"=="|"!=" {strcpy(yylval.type\_id, yytext);return RELOP;}  "=" {return ASSIGNOP;}  "+" {return PLUS;}  "-" {return MINUS;}  "\*" {return STAR;}  "/" {return DIV;}  "++" {return SELFADD;}  "--" {return SELFDEC;}  "+=" {return ADD\_ASSIGNOP;}  "-=" {return MINUS\_ASSIGNOP;}  "\*=" {return STAR\_ASSIGNOP;}  "/=" {return DIV\_ASSIGNOP;}  "&&" {return AND;}  "||" {return OR;}  "." {return DOT;}  "!" {return NOT;} |

（6）注释

注释有两种形式，一种是"\\"只能单行注释，一种是"/\*\*/"可以多行注释。其中多行注释采用一个内置函数实现识别。

|  |
| --- |
| "//" {char c=input();while(c!='\n') c=input();}  "/\*" {moreLinesComment();}  …  int moreLinesComment()  {  char c,c1;  loop:  while((c=input())!='\*'&&c!=0){}  if((c1 = input())!='/'&&c!=0)  {  unput(c1);  goto loop;  }  return 0;  } |

### 3.3.2语法分析器

使用bison只需要写好文法规则并设计好规约动作就可以自动生成语法分析器。

|  |
| --- |
| program: ExtDefList {/\*display($1, 0);\*/udisplay($1, 0);semantic\_Analysis0($1);}  ;  ExtDefList: {$$=NULL;}  | ExtDef ExtDefList {$$=mknode(EXT\_DEF\_LIST,$1,$2,NULL,yylineno);}  ;  ExtDef: Specifier ExtDecList SEMI {$$=mknode(EXT\_DEF\_VAR,$1,$2,NULL,yylineno);}  |Specifier FuncDec CompSt {$$=mknode(EXT\_DEF\_FUNC,$1,$2,$3,yylineno);}  | error SEMI {$$=NULL; }  // |Specifier SEMI {}  ;  ExtDecList: VarDec {$$=$1;}  |VarDec COMMA ExtDecList {$$=mknode(EXT\_VARDEC\_LIST,$1,$3,NULL,yylineno);}  ;  //Specifiers  Specifier: TYPE {$$=mknode(TYPE,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);$$->type=!strcmp($1,"int")?INT:(!strcmp($1,"float")?FLOAT:CHAR);}  ;  //Declarators  VarDec: ID LB INT RB {$$=mknode(ARRAY,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);$$->size=$3;}  |ID {$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}  ;  FuncDec: ID LP VarList RP {$$=mknode(FUNC\_DEC,$3,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}  |ID LP RP {$$=mknode(FUNC\_DEC,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}  ;  VarList: ParamDec {$$=mknode(FUNC\_PARAM\_LIST,$1,NULL,NULL,yylineno);}  | ParamDec COMMA VarList {$$=mknode(FUNC\_PARAM\_LIST,$1,$3,NULL,yylineno);}  ;  ParamDec: Specifier VarDec {$$=mknode(FUNC\_PARAM\_DEC,$1,$2,NULL,yylineno);}  ;  //Statements  CompSt: LC DefList StmtList RC {$$=mknode(COMP\_STM,$2,$3,NULL,yylineno);}  ;  StmtList: {$$=NULL; }  | Stmt StmtList {$$=mknode(COMPSTM\_LIST,$1,$2,NULL,yylineno);}  ;  Stmt: Exp SEMI {$$=mknode(COMPSTM\_EXP,$1,NULL,NULL,yylineno);}  | CompSt {$$=$1;}  | RETURN Exp SEMI {$$=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 {$$=mknode(WHILE,$3,$5,NULL,yylineno);}  | FOR LP ExpForCond RP Stmt {$$=mknode(FOR,$3,$5,NULL,yylineno);}  ;  ExpForCond: ExpFor1 SEMI ExpFor2 SEMI ExpFor3 {$$=mknode(FOR\_CONDITION,$1,$3,$5,yylineno);}  ;  //Local Definitions  DefList: {$$=NULL; }  | Def DefList {$$=mknode(DEF\_LIST,$1,$2,NULL,yylineno);}  ;  Def: Specifier DecList SEMI {$$=mknode(VAR\_DEF,$1,$2,NULL,yylineno);}  ;  DecList: Dec {$$=mknode(VARDEC\_LIST,$1,NULL,NULL,yylineno);}  | Dec COMMA DecList {$$=mknode(VARDEC\_LIST,$1,$3,NULL,yylineno);}  ;  Dec: VarDec {$$=$1;}  | VarDec ASSIGNOP Exp {$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,"ASSIGNOP");}  ;  //Expressions  Exp: Exp ASSIGNOP Exp {$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,"ASSIGNOP");}  | Exp AND Exp {$$=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($$->type\_id,$2);}  | Exp PLUS Exp {$$=mknode(PLUS,$1,$3,NULL,yylineno);strcpy($$->type\_id,"PLUS");}  | Exp MINUS Exp {$$=mknode(MINUS,$1,$3,NULL,yylineno);strcpy($$->type\_id,"MINUS");}  | Exp STAR Exp {$$=mknode(STAR,$1,$3,NULL,yylineno);strcpy($$->type\_id,"STAR");}  | Exp DIV Exp {$$=mknode(DIV,$1,$3,NULL,yylineno);strcpy($$->type\_id,"DIV");}  | LP Exp RP {$$=$2;}  | MINUS Exp %prec UMINUS {$$=mknode(UMINUS,$2,NULL,NULL,yylineno);strcpy($$->type\_id,"UMINUS");}  | NOT Exp {$$=mknode(NOT,$2,NULL,NULL,yylineno);strcpy($$->type\_id,"NOT");}  | VarDec SELFADD {$$=mknode(SELFADD,$1,NULL,NULL,yylineno);strcpy($$->type\_id,"SELFADD");}  | SELFADD VarDec {$$=mknode(SELFADD,NULL,$2,NULL,yylineno);strcpy($$->type\_id,"SELFADD");}  | VarDec SELFDEC {$$=mknode(SELFDEC,$1,NULL,NULL,yylineno);strcpy($$->type\_id,"SELFDEC");}  | SELFDEC VarDec {$$=mknode(SELFDEC,NULL,$2,NULL,yylineno);strcpy($$->type\_id,"SELFDEC");}  | VarDec ADD\_ASSIGNOP Exp {$$=mknode(ADD\_ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,"ADD\_ASSIGNOP");}  | VarDec MINUS\_ASSIGNOP Exp {$$=mknode(MINUS\_ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,"MINUS\_ASSIGNOP");}  | VarDec STAR\_ASSIGNOP Exp {$$=mknode(STAR\_ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,"STAR\_ASSIGNOP");}  | VarDec DIV\_ASSIGNOP Exp {$$=mknode(DIV\_ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,"DIV\_ASSIGNOP");}  | ID LP Args RP {$$=mknode(FUNC\_CALL,$3,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}  | ID LP RP {$$=mknode(FUNC\_CALL,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}  | ID LB Exp RB {$$=mknode(ARRAY\_CALL,$3,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}  | VarDec {$$=$1;}  | INT %prec LOWER\_THAN\_RB {$$=mknode(INT,NULL,NULL,NULL,yylineno);$$->type\_int=$1;$$->type=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;}  | BREAK {$$=mknode(\_BREAK,NULL,NULL,NULL,yylineno);}  | CONTINUE {$$=mknode(\_CONTINUE,NULL,NULL,NULL,yylineno);}  ;  Args: Exp COMMA Args {$$=mknode(ARGS,$1,$3,NULL,yylineno);}  | Exp {$$=mknode(ARGS,$1,NULL,NULL,yylineno);}  ;  ExpFor1:{$$=NULL; }  |ExpFor1List {$$=$1;}  ;  ExpFor1Atom:  Specifier Dec {$$=mknode(VAR\_DEF,$1,$2,NULL,yylineno);}  |VarDec ASSIGNOP Exp {$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,"ASSIGNOP");}  ;  ExpFor1List: ExpFor1Atom{$$=$1;}  |ExpFor1Atom COMMA ExpFor1List {$$=mknode(EXP\_FOR1\_LIST,$1,$3,NULL,yylineno);}  ;  ExpFor2: {$$=NULL; }  |VarDec RELOP Exp {$$=mknode(RELOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,$2);}  ;  ExpFor3: {$$=NULL;}  | ExpFor3List {$$=$1;}  ;  ExpFor3List: Exp {$$=$1;}  | Exp COMMA ExpFor3List {$$=mknode(EXP\_FOR3\_LIST,$1,$3,NULL,yylineno);}  ; |

## 3.2编译程序符号表结构

定义一个符号表中一个符号的结构体如下：

符号表的一行是一个包括如下内容的结构体：

|  |
| --- |
| struct symbol{  char name[33]; //变量或函数名  int level; //外部或函数层号为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; //形参个数  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，数组：’A' 函数：'F' 变量：'V' 参数：'P' 临时变量：'T'  int offset; //外部变量和局部变量在其静态数据区或活动记录中的偏移量，  //或记录函数活动记录大小，目标代码生成时使用  }; |

其中各个字段的含义如注释所示。

整个符号表包含一个index值和若干行，如下所示：

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

这里的符号表采用数组的形式来实现。每当插入一个新的结点时，就将index加一，所以可以看作一个模拟栈，index指示栈顶。

## 3.3语义分析（实验二）

语义分析这部分的一个非常重要的工作就是符号表的管理，在编译过程中，编译器使用符号表来记录源程序中各种名字的特性信息。所谓“名字”包括：程序名、过程名、函数名、用户定义类型名、变量名、常量名、枚举值名、标号名等，所谓“特性信息”包括： 上述名字的种类、 具体类型、维数（如果语言支持数组）、函数参数个数、常量数值及目标地址（存储单元偏移地址）等。

语义分析这部分完成的是静态语义分析，主要包括：

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

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

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

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

变量的作用域：

在语义分析过程中，各个变量名有其对应的作用域，一个作用域内不允许名字重复，为此，通过一个全局变量LEV来管理，LEV的初始值为0。这样在处理外部变量名，以及函数名时，对应符号的层号值都是1；处理函数形式参数时，固定形参名在填写符号表时，层号为1。由于mini\_C中允许有复合语句，复合语句中可定义局部变量，函数体本身也是一个复合语句，这样在AST的遍历中，通过LEV的修改来管理不同的作用域。

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

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

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

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

## 3.4编译程序报错功能

编译程序的报错大体分为以下四个方面：

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

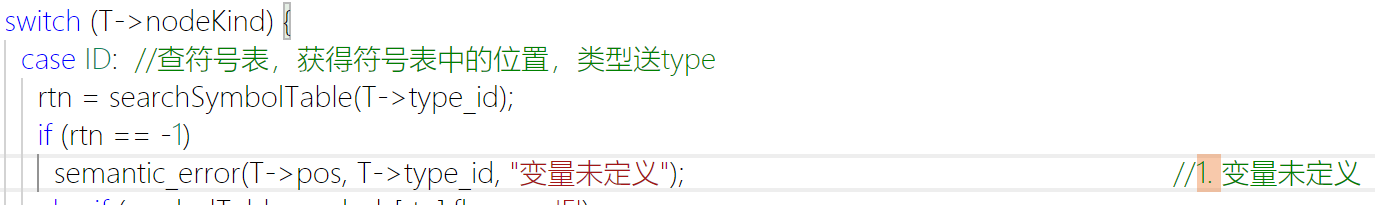
（2）唯一性检查。对于某些不能重复定义的对象或者元素，如同一作用域的标识符不能同名，需要在语义分析阶段检测出来。

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

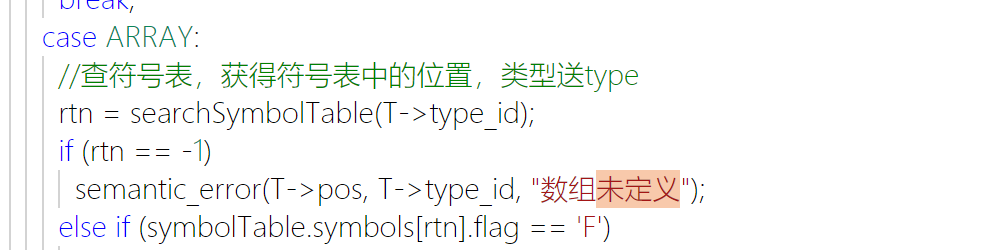
（4）类型检查包括检查函数参数传递过程中形参与实参类型是否匹配、是否进行自动类型转换等等。

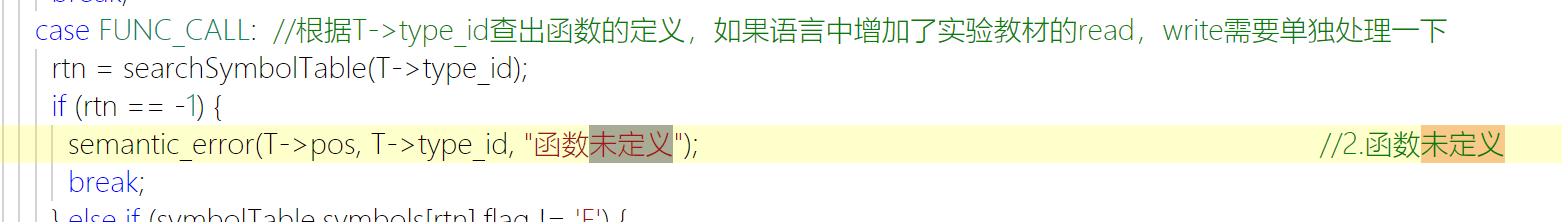
具体的检查方案在Analysis.c文件中均有注释标明，错误类型的具体标号也有写出。有的错误需要在多处判断，例如类型不匹配等。接下来列出一些错误的判断实现：

1.变量未定义

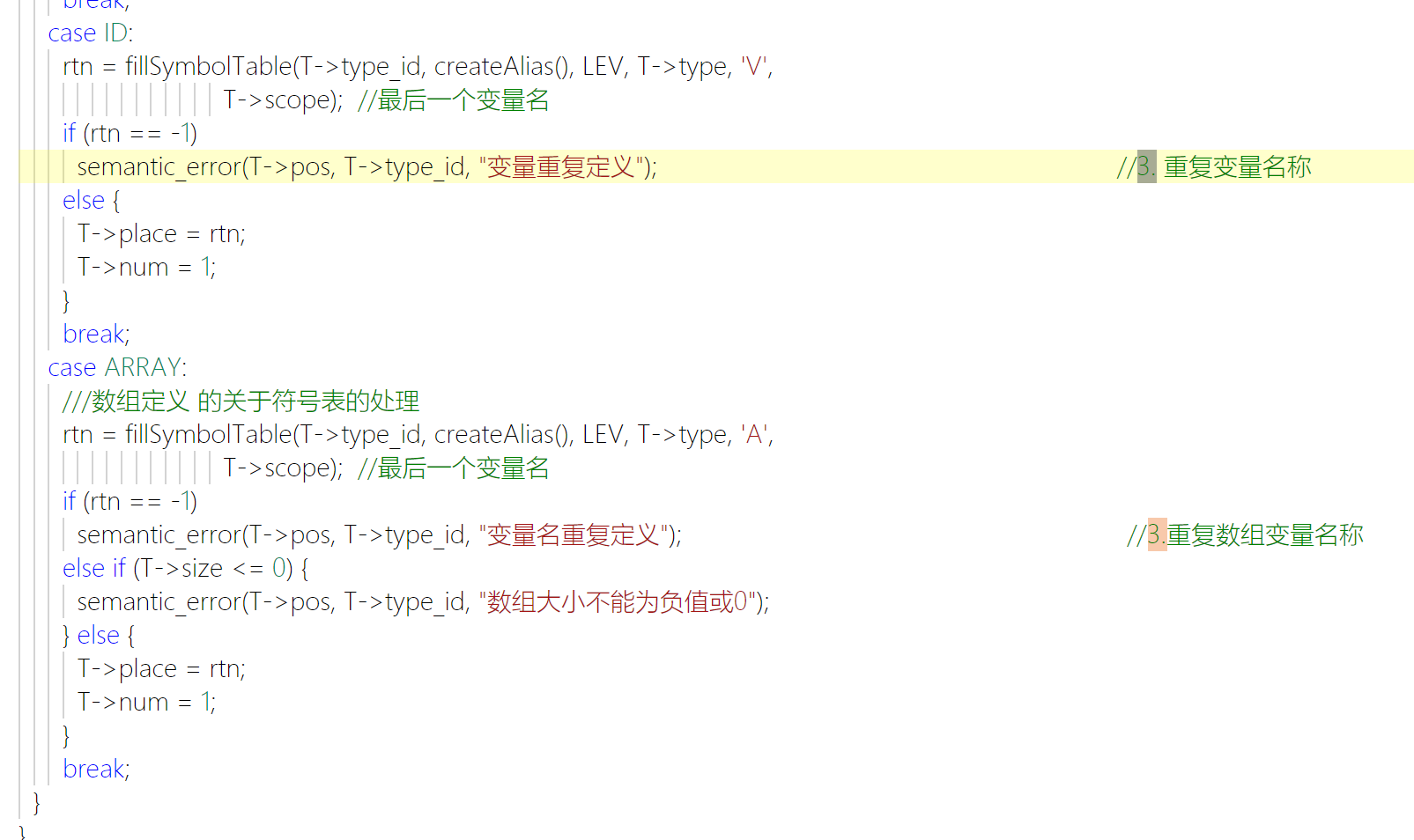


2.函数/数组未定义

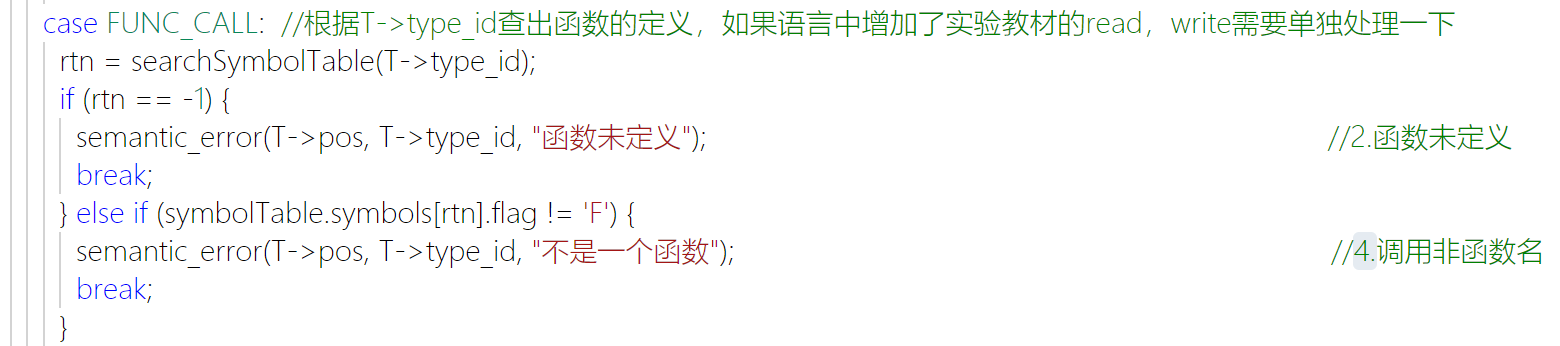




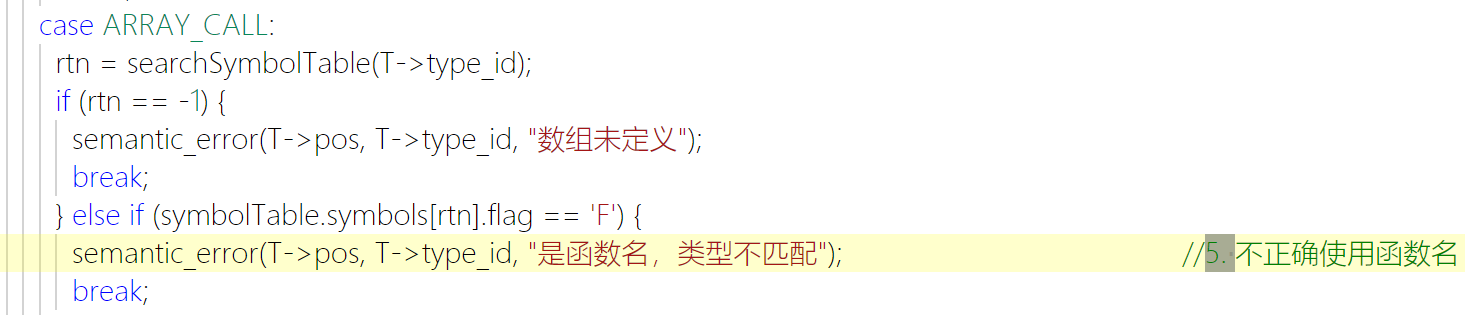
3.变量名重复

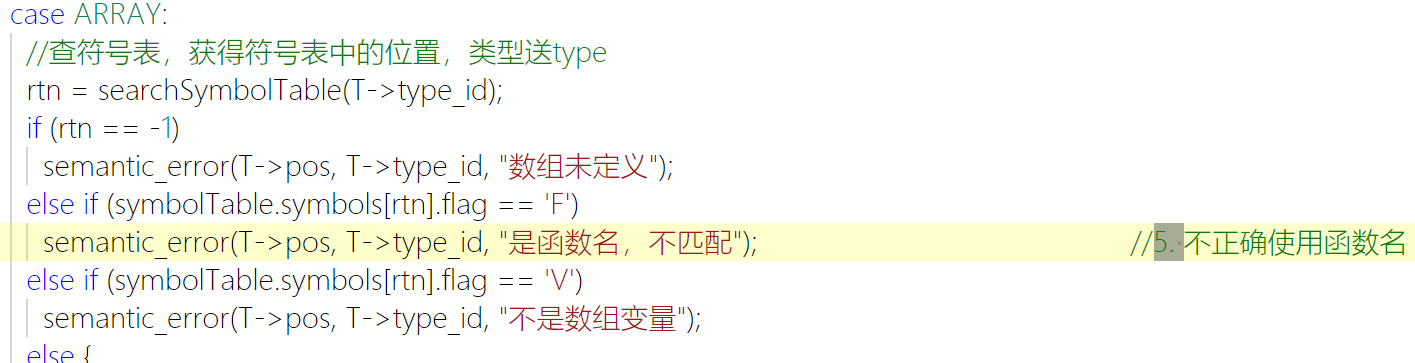


4. 调用非函数名

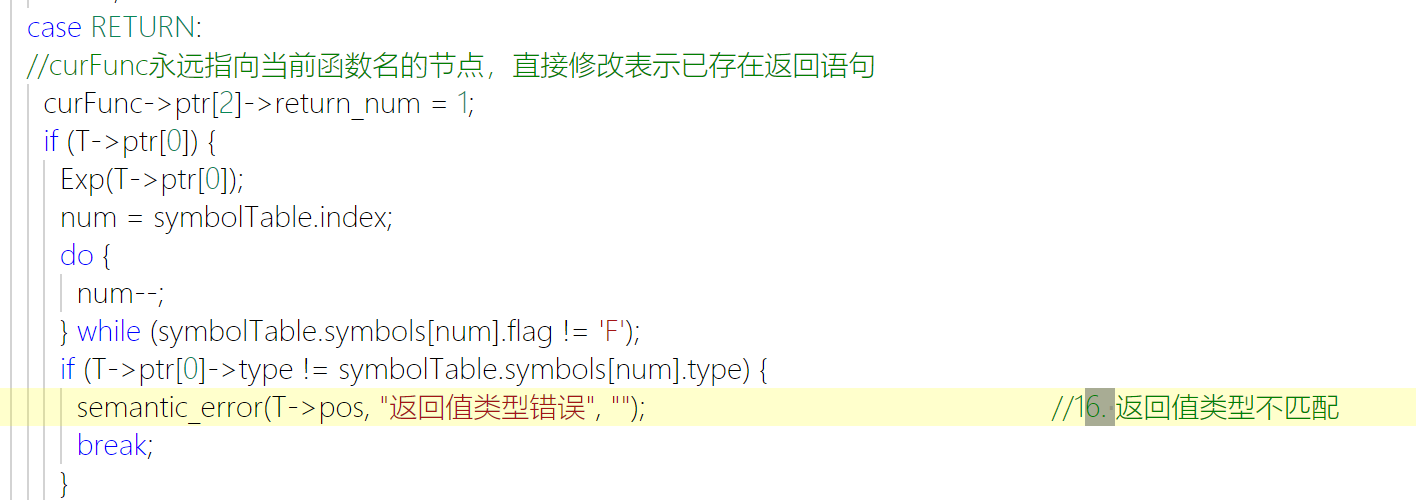


5. 不正确地使用函数名

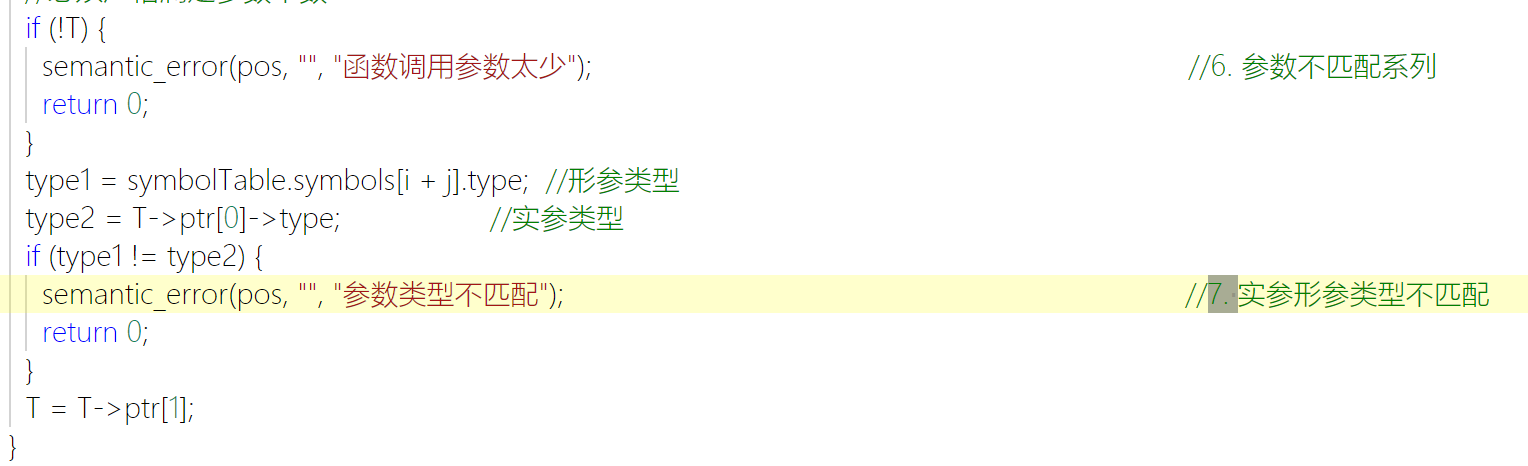




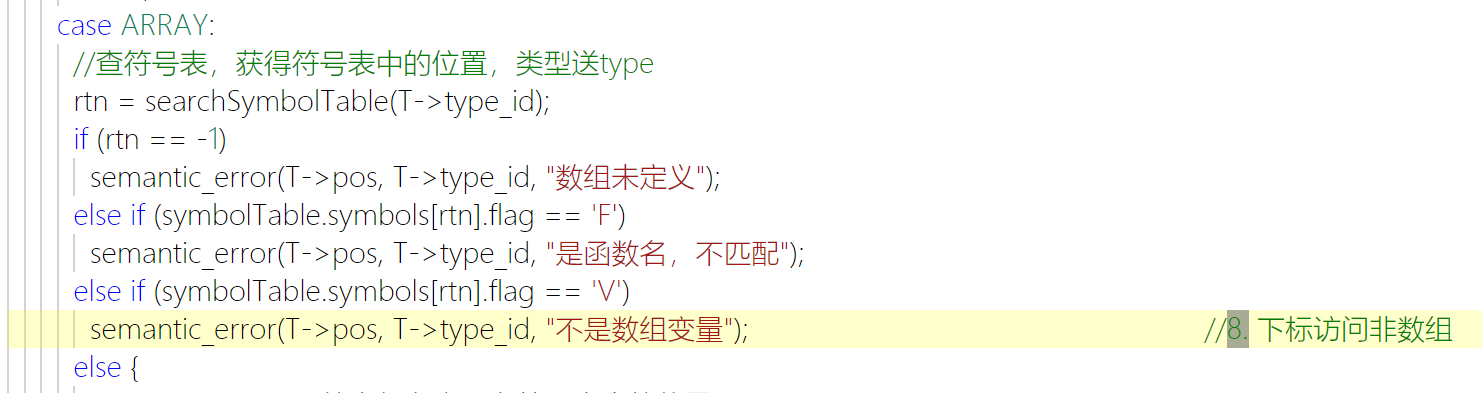
6. 返回值类型不匹配



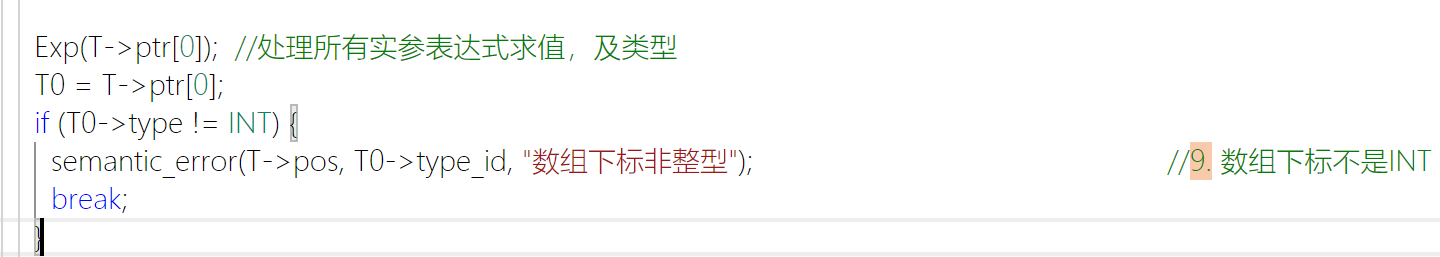
7. 参数类型和形参不匹配



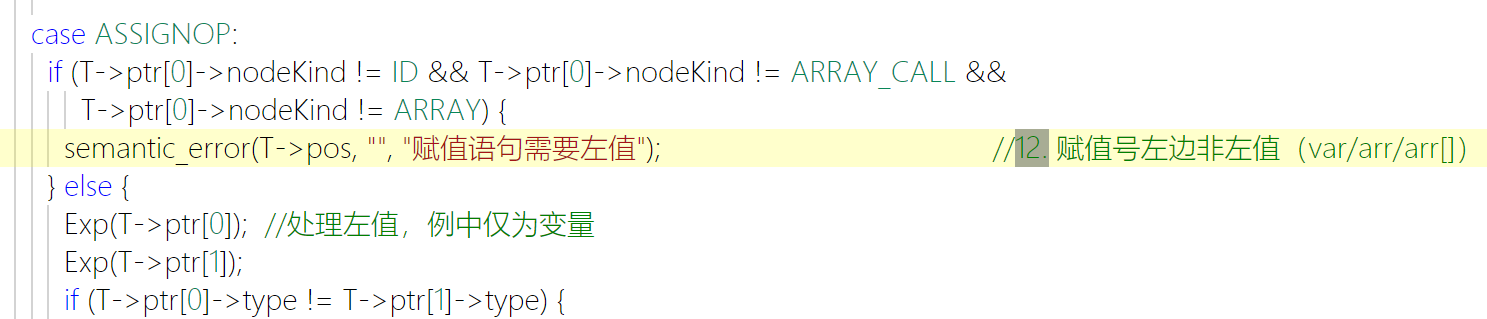
1. 使用下标访问非数组变量



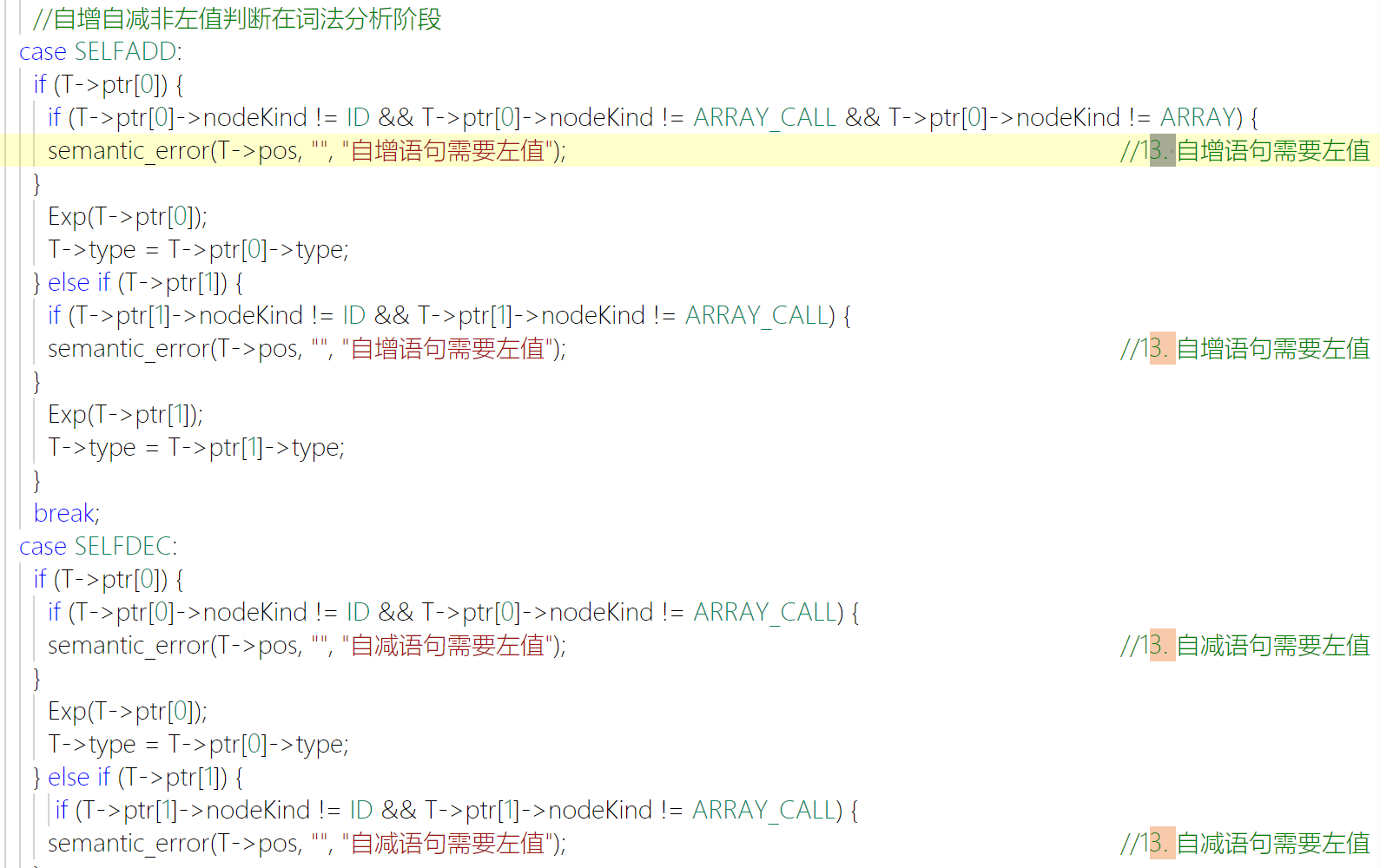
1. 数组下标非INT



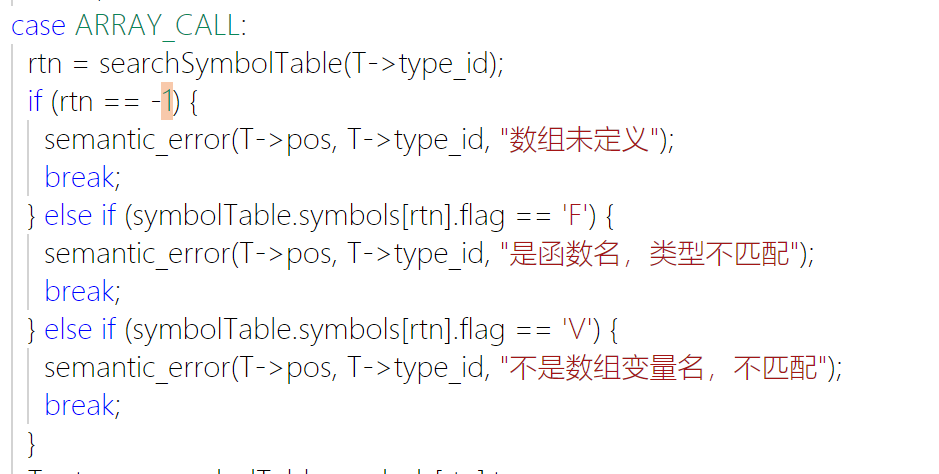
1. 赋值号左边非左值



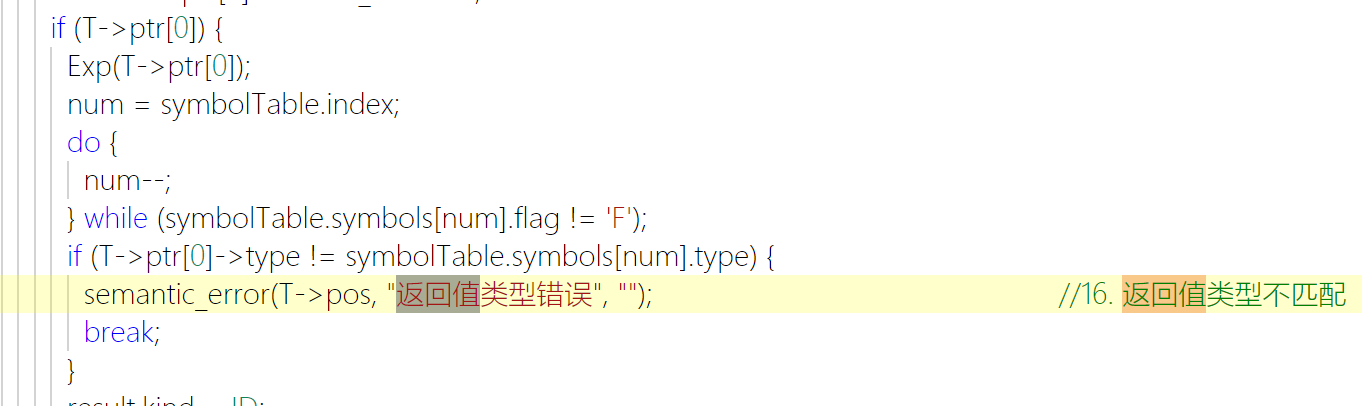
1. 自增自减需要左值



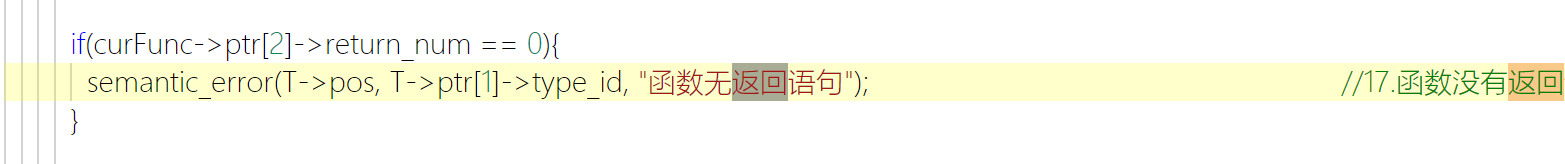
1. 类型不匹配



1. 返回值类型不匹配



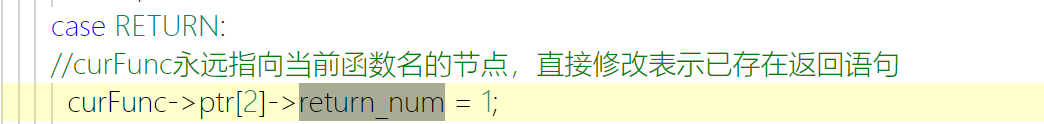
1. 无返回语句



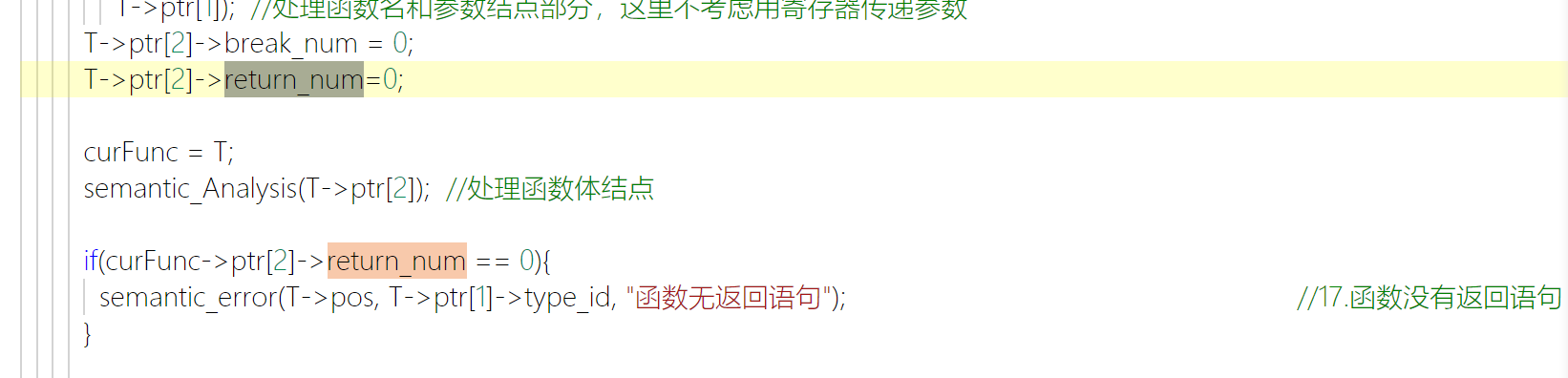
注：这里在ASTnode里定义了return\_num成员，表示当前所在函数是否出现了RETURN语句:



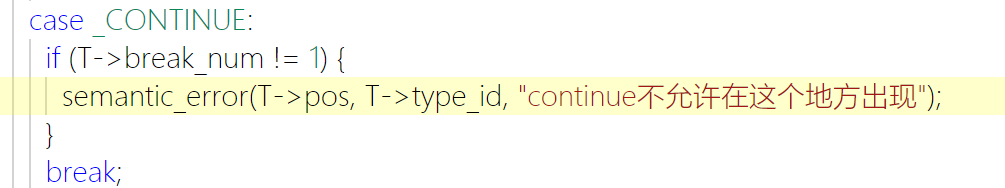
以及一个进入函数式留滞的指针，指向当前函数的根节点



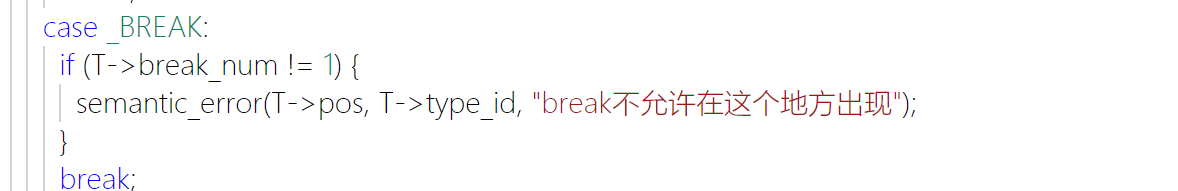
退出函数时判断，没有return则不会在RETURN判断将当前函数的return\_num置0，则可以判断函数没有返回语句。



1. continue位置不合法（break不能出现的位置，continue就不能出现）



1. break位置不合法



break\_num是ASTnode的一个成员变量，只有进入while/for循环之后，才将其置为1，表示可以出现break/continue语句。

## 3.5中间代码生成功能（实验三）

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

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

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

.num 记录子结点的个数，包括自身

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

.offset 记录外部变量在静态数据区中的偏移量以及局部变量和临时变量在活动记录中的偏移量。另外对函数，利用这项保存活动记录的大小。

.width 记录一个结点表示的语法单位中，定义的变量和临时单元所需要占用的字节数，方便计算变量、临时变量在活动记录中偏移量，以及最后计算函数活动记录的大小。

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

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

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

具体中间代码的实现在整个Analysis.c内较为分散，不过我的实现主要也参照了实验指导书上给出来的实现示例：

1. 语句类节点中间代码生成：

|  |  |
| --- | --- |
| **当 前 结 点 类 型** | **翻 译 动 作** |
| COMP\_STM  T1 说明部分子树  T2 语句部分子树 | 访问到T：T2.Snext=T.Snext  访问T的所有子树后：  T.code=T1.code || T2.code |
| IF\_THEN  T1 条件子树  T2 if子句子树 | 访问到T：T1.Etrue=newLabel, T1.Efalse= T2.Snext=T.Snext  访问T的所有子树后：  T.code=T1.code || T1.Etrue ||T2.code |
| IF\_THEN\_ELSE  T1 条件子树  T2 if子句子树  T3 else子句子树 | 访问到T： T1.Etrue=newLabel,T1.Efalse=T.Snext  T1.Snext= T2.Snext=T.Snext  访问T的所有子树后：  T.code=T1.code || T1.Etrue || T2.code || goto T.Enext  || T1.Efalse || T3.code |
| WHILE  T1 条件子树  T2 if子句子树 | 访问到T：T1.Etrue=newLabel, T1.Efalse=T.Snext,  T2.Snext= newLabel;  访问T的所有子树后：  T.code=T2.Snext || T1.code || T1.Etrue || T2.code || goto T2.Snext |
| STM\_LIST  T1 语句1子树  T2 语句2子树（可空） | 访问到T：if(T2非空) {T1.Snext=newLabel , T2.Snext=T.Snext;}  else T1.Snext=S.next;  访问T的所有子树后：  if (T2为空) T.code=T1.code  else T.code=T1.Snext || T1.Snext || T2.code |
| EXP\_STM  T1 表达式子树 | 访问到T：T1.Snext=T.Snext;  访问T的所有子树后：T.code=T1.code |
| RETURN  T1 表达式子树（可空） | 访问到T：T1.Snext=T.Snext;  访问T的所有子树后：  if (T1非空) T.code=T1.code|| return T1.alias  else T.code=T1.code|| return |
|  |  |

1. 基本表达式类节点中间代码生成：

|  |  |
| --- | --- |
| **当 前 结 点 类 型** | **翻 译 动 作** |
| INT  其它如FLOAT类的结点按类似方法处理 | ti=newtemp, ti在符号表的入口赋值给T.place。后续可通过T.alias读取该值。  T.code为： ti = INT的值 |
| ID | ID在符号表中的入口赋值给T.place，代码为空 |
| ASSIGNOP  T1 左值表达式子树  T2 左值表达式子树 | 访问到T： T. palce=T1.place  访问T的所有子树后：T.code=T1.code || T2.code || T1.alias= T2.alias |
| OP 算术运算符。  T1 第一操作数子树  T2 第二操作数子树 | ti=newtemp, ti在符号表的入口赋值给T.place  访问T的所有子树后：  T.code=T1.code || T2.code || ti=T1.alias OP T2.alias |
| UMINUS  T1 操作数子树 | ti=newtemp, ti在符号表的入口赋值给T.place  访问T的所有子树后：T.code=T1.code || ti=- T1.alias |
| RELOP 关系运算符  T1 第一操作数子树  T2 第二操作数子树 | ti=newtemp, ti在符号表的入口赋值给T.place，  Label1=newLabel，Label2=newLabel。  访问T的所有子树后：  T.code=T1.code || T2.code  || if T1.alias RELOP T2.alias goto label1  || ti=0 || goto label2 || label1： || ti=1 || label2： |
| AND  T1 第一操作数子树  T2 第二操作数子树 | ti=newtemp, ti在符号表的入口赋值给T.place，  Label1=newLabel。  访问T的所有子树后：  T.code=T1.code || T2.code || ti=T1.alias \* T2.alias  || if ti==0 goto label1 || ti=1 || label1： |
| OR  T1 第一操作数子树  T2 第二操作数子树 | ti=newtemp, ti在符号表的入口赋值给T.place，  Label1=newLabel，Label2=newLabel。  访问T的所有子树后：  T.code=T1.code || T2.code || ti=0  || if T1.alias ==0 goto label1 || ti=1 || goto label2 || label1：  || if T2.alias ==0 goto label2 || ti=1 || label2： |
| NOT  T1 操作数子树 | ti=newtemp, ti在符号表的入口赋值给T.place，  Label1=newLabel，Label2=newLabel。  访问T的子树后：  T.code= if T1.alias ==0 goto label1  || ti=0 || goto label2 || label1：|| ti=1 || label2： |
| FUNC\_CALL  T1 实参列表子树 | ti=newtemp, ti在符号表的入口赋值给T.place  T.code=T1.code;  访问T的子树，从上至下依次对每个ARGS实参结点T0，完成实参处理。  T.code= T. code || ARG T01.alias  这里T01表示T0的第一个孩子，访问T的子树后：  T.code= T.code || ti=CALL 函数名 |

1. 控制语句Bool表达式语句节点中间代码生成：

|  |  |
| --- | --- |
| **当 前 结 点 类 型** | **翻 译 动 作** |
| INT  其它如FLOAT类的结点按类似方法处理 | if (T.Etrue==””) 按基本表达式处理  else if (INT的值) T.code= goto T.Etrue  else T.code= goto T.Efalse |
| ID | ID在符号表中的入口赋值给T.place  if (T.Etrue==””) 按基本表达式处理  else T.code= if T.alias!=0 goto T.Etrue || goto T.Efalse |
| ASSIGNOP  T1 左值表达式子树  T2 左值表达式子树 | T.palce=T1.place  访问T的所有子树后：  T.code=T1.code || T2.code || T1.alias= T2.alias  if (T.true!=””)  T.code=T.code || if T1.alias!=0 goto T.Etrue || goto T.Efalse |
| OP 算术运算符。  T1 第一操作数子树  T2 第二操作数子树 | ti=newtemp, ti入口赋值给当前结点T.place  访问T的所有子树后：  T.code =T1.code || T2.code || ti=T1.alias OP T2.alias  if (T.true!=””)  T.code = T.code || if ti!=0 goto T.Etrue || goto T.Efalse |
| UMINUS  T1 操作数子树 | ti=newtemp, ti入口赋值给当前结点place  访问T的所有子树后：  T.code = T1.code || ti=- T1.alias  if (T.true!=””)  T.code=T.code || if ti!=0 goto T.Etrue || goto T.Efalse |
| RELOP 关系运算符  T1 第一操作数子树  T2 第二操作数子树 | if (T.true==””) ti=newtemp, ti入口赋值给T.place，  Label1=newLabel，Label2=newLabel。  访问T的所有子树后：  T.code = T1.code || T2.code  if (T.true!=””)  T.code=T.code || if T1.alias RELOP T2.alias goto T.Etrue  || goto T.Efalse  else T.code=T.code || if T1.alias RELOP T2.alias goto label1  || ti=0 || goto label2 || label1：|| ti=1 || label2： |
| AND  T1 第一操作数子树  T2 第二操作数子树 | if (T.Etrue==””) 按基本表达式处理  else T1.Etrue= newLabel, T2.Etrue= T.Etrue,  T1.Efalse= T2.Efalse =T.Efalse;  T.code=T1.code || T1.Etrue || T2.code |
| OR  T1 第一操作数子树  T2 第二操作数子树 | if (T.Etrue==””) 按基本表达式处理  else T1.Etrue=T2.Etrue= T.Etrue,T1.Efalse= newLabel,  T2..Efalse =T.Efalse;  T.code=T1.code || T1.Efalse || T2.code |
| NOT  T1 操作数子树 | f (T.Etrue==””) 按基本表达式处理  else T1.Etrue=T.Efalse, T1.Efalse =T.Etrue;  T.code=T1.code： |
| FUNC\_CALL  T1 实参列表子树 | ti=newtemp, ti入口赋值给T.place  T.code=T1.code;  访问T的子树，从上至下依次对每个ARGS结点T0，完成实参处理。  T.code= T. code || ARG T01.alias  这里T01表示T0的第一个孩子，访问T的子树后：  T.code= T.code || ti=CALL 函数名  f (T.Etrue!=””) T.code=T.code || if ti!=0 goto T.Etrue || goto T.Efalse |

1. 其他类型节点中间代码生成：

|  |  |
| --- | --- |
| **当 前 结 点 类 型** | **翻 译 动 作** |
| FUNC\_DEF  T1 返回值类型  T2 函数名与参数  T3 函数体 | 访问T时：T3.Snext=newLabel  访问T的所有子树后:  T.code=T2.code || T3.code || T3.Snext |
| FUNC\_DEC  T1 参数列表（可空） | 访问T的所有子树后:  T.code=FUNCTION 函数名  if (T1非空) T.code=T.code || T1.code |
| PARAM\_LIST  T1 形参说明子树  T2形参列表子树（可空） | 访问T的所有子树后:  T.code=T1.code  if (T2非空) T.code=T.code || T2.code |
| PARAM\_DEC  T1 形参类型  T2形参名 | 访问T的所有子树后:  T.code=PARAM T2.alias |
| ARGS  T1 实参子树  T2 实参列表子树（可空） | 访问T的所有子树后：  if（T2==NULL）T.code= T1.code  else T.code= T1.code || T2.code |

另外还新增了一些实现，例如对符号表中数组空间的预留和处理，continue和break跳转逻辑实现的中间代码生成以及循环的支持等等，具体见Analysis.c文件及其注释。

## 3.6汇编代码生成功能（实验四）

这部分要完成将TAC指令序列转换成目标代码，目标代码选择为MIPS汇编指令。最终生成的MIPS汇编指令可以在SPIM Simulator上运行。具体内容参见2.6节，代码见提交文件的objectCodeGen.c。

# 4系统测试与评价

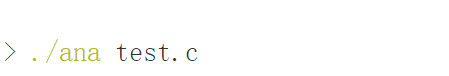
## 4.1 测试用例及测试结果

实验一测试代码如下所示：

|  |
| --- |
| int a,b,c;  float m,n;  int fibo(int a){  if(a==1 ||a==2) return 1;  return fibo(a-1)+fibo(a-2);  }  int main(){  int m,n,i;  m=read();  i = 1;  while(i<=m){  if(i==2) continue;  n = fibo(i);  write(n);  i = i+1;  }  return 1;  } |

测试命令如下：





测试结果如下:

PS C:\Users\caxus\Desktop\Desktop\Univ\_Assenbly\_Principle\aplab1> ./ana test.c

外部定义列表:

外部变量定义:

类型：int

变量名：

ID：a

ID：b

ID：c

外部定义列表:

外部变量定义:

类型：float

变量名：

ID：m

ID：n

外部定义列表:

外部函数定义：

类型：int

函数名：fibo

函数形参：

类型：int，参数名：a

复合语句：

复合语句变量定义：

复合语句语句部分：

IF\_THEN条件语句：

IF条件：

||

==

ID：a

INT：1

==

ID：a

INT：2

THEN子句：(5)

返回语句：

INT：1

IF\_THEN条件语句结束

返回语句：

+

函数调用：

函数名：fibo

函数参数：

-

ID：a

INT：1

函数调用：

函数名：fibo

函数参数：

-

ID：a

INT：2

复合语句结束

外部定义列表:

外部函数定义：

类型：int

函数名：main

无参函数

复合语句：

复合语句变量定义：

局部变量定义列表:

局部变量定义:

类型：int

变量名：

ID：m

ID：n

ID：i

复合语句语句部分：

表达式语句：

=

ID：m

函数调用：

函数名：read

表达式语句：

=

ID：i

INT：1

WHILE循环语句：

WHILE条件：

<=

ID：i

ID：m

WHILE循环体：

复合语句：

复合语句变量定义：

复合语句语句部分：

IF\_THEN条件语句：

IF条件：

==

ID：i

INT：2

THEN子句：(13)

表达式语句：

CONTINUE

IF\_THEN条件语句结束

表达式语句：

=

ID：n

函数调用：

函数名：fibo

函数参数：

ID：i

表达式语句：

函数调用：

函数名：write

函数参数：

ID：n

表达式语句：

=

ID：i

+

ID：i

INT：1

复合语句结束

WHILE循环语句结束

返回语句：

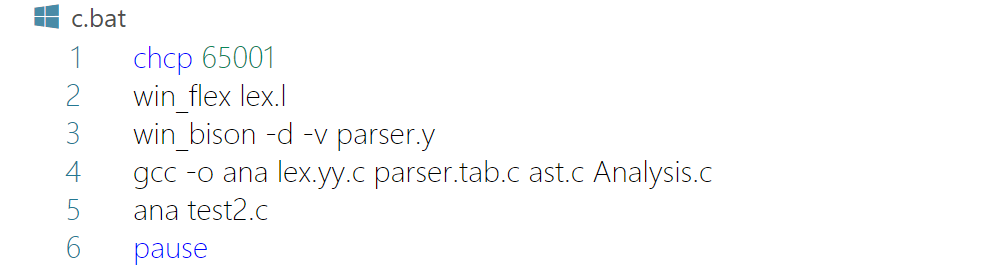
INT：1

复合语句结束

实验二测试代码:

|  |
| --- |
| int a[5];  float b;  char c[6];  int func(int i, int y) { return i; }  int func() { //错误类型3，名称的重复定义  return 0;  }  int func1(){} //错误类型17，函数没有返回语句  int main() {  // this is a test  int i;  char p[10];  float d = 5.5, i; //错误类型3，名称的重复定义  j = i + 1; //错误类型1，使用未定义的变量  inc(i); //错误类型2，调用未定义或未声明的函数  a[i] = 2.2; //错误类型15，赋值号两边的表达式类型不匹配  10 = i; //错误类型12，赋值号左边不是左值表达式  i++;  a = a + 1; //错误类型7，操作数类型不匹配(a是数组变量)  d += 10; //错误类型7，复合运算类型不匹配（d是float型）  func = 3; //错误类型5，对函数名采用非函数调用形式访问 (//同时15，赋值号两边的表达式类型不匹配 )  func();  func(1); //错误类型6，函数调用时参数个数不匹配(过少)  func(2, 3);  func(2, 3.1); //错误类型7, 函数调用时实参和形参类型不匹配  func(1, 2, 3.1); //错误类型6，函数调用时参数个数不匹配(过多)  i[10]; //错误类型8，对非数组变量采用下标变量的形式访问  i(10); //错误类型4， 对非函数名采用函数调用形式  p[1.5] = 10; //错误类型9，数组变量的下标不是整型表达式  p[2] = 'A';  for (a = 1, kk = 1; a < 10; a++) {  int c1 = 0;  c1 = c1++;  // func++; //---------错误类型5，对函数名采用非函数调用形式访问 (在词法分析阶段就会报错  //12++; //---------错误类型13. 自增非左值 (在词法分析阶段就会报错  for (c = 1; c < 5; c++) {  int c2 = 3;  }  continue;  }  continue; //错误类型19，continue位置非法  break; //错误类型18，break位置非法  return d; //错误类型16，返回值类型不匹配  } |

测试脚本命令如下：



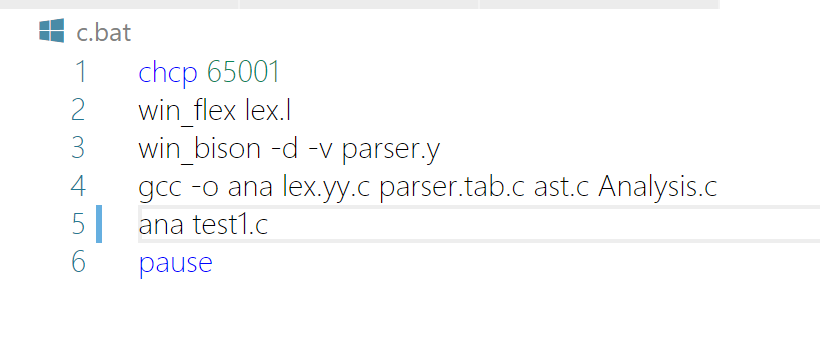
测试结果如下：



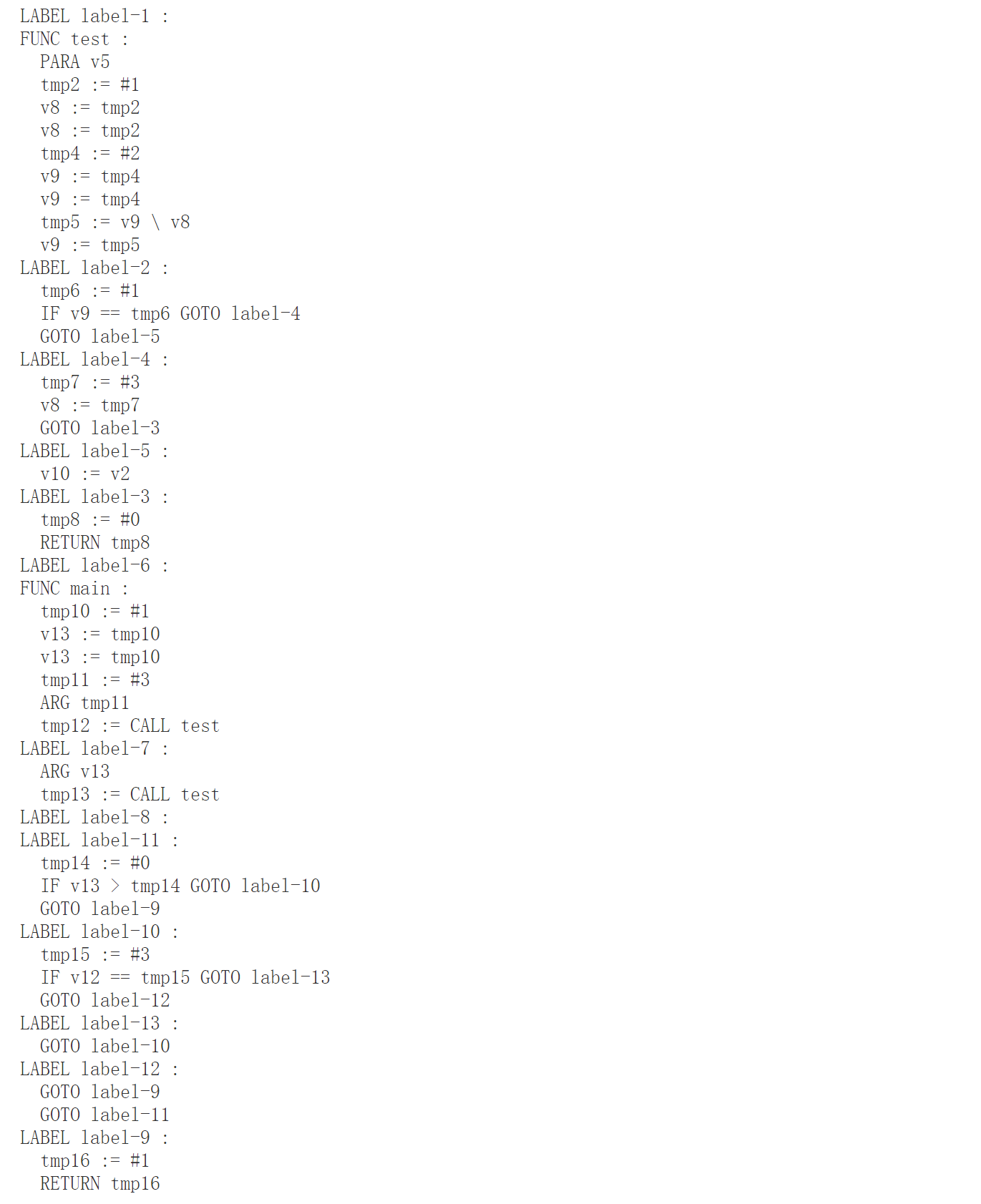
实验三测试代码

|  |
| --- |
| int cjt1,cjt2;  float cjt3;  int test(int a)  {  char cjt4[25];  int a2=1;  int a1=2;  a1=a1/a2;  if(a1==1)  {  a2=3;  }  else  {  int j;  j=cjt2;  }  return 0;  }  int main()  {  int a1,a2=1;  test(3);  test(a2);  while(a2>0){  if(a1==3) continue;  break;  }  return 1;  } |

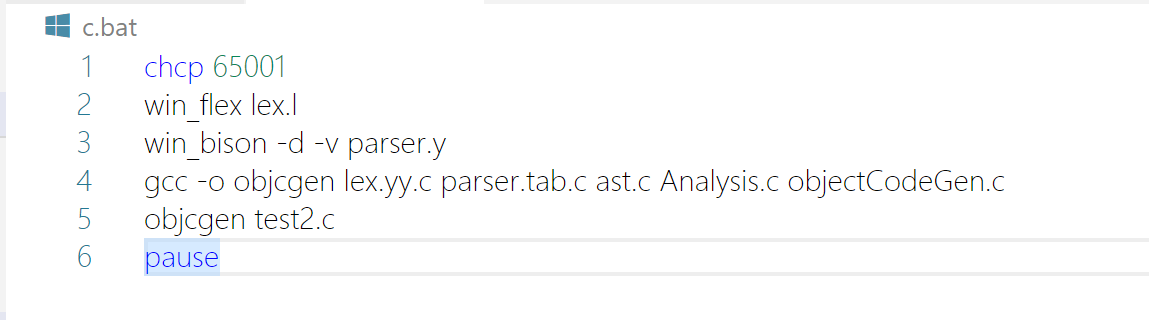
测试命令脚本如下：



测试结果如下：



实验四测试代码与实验三相同，测试脚本如下：



测试生成汇编代码如下：

C:\Users\caxus\Desktop\Desktop\Univ\_Assenbly\_Principle\aplab4>objcgen test2.c

.data

.globl main

.text

label1:

func:

li $t3, 1

sw $t3, 20($sp)

lw $t1, 12($sp)

lw $t2, 20($sp)

blt $t1,$t2,label2

j label3

label2:

lw $t1, 12($sp)

lw $t2, 16($sp)

add $t3,$t1,$t2

sw $t3, 20($sp)

lw $v0, 20($sp)

jr $ra

j label1

label3:

li $t3, 1

sw $t3, 20($sp)

lw $t1, 12($sp)

lw $t2, 20($sp)

sub $t3,$t1,$t2

sw $t3, 24($sp)

li $t3, 1

sw $t3, 28($sp)

lw $t1, 16($sp)

lw $t2, 28($sp)

add $t3,$t1,$t2

sw $t3, 32($sp)

move $t0,$sp

addi $sp, $sp, -40

sw $ra,0($sp)

lw $t1, 24($t0)

move $t3,$t1

sw $t3,12($sp)

lw $t1, 32($t0)

move $t3,$t1

sw $t3,16($sp)

jal func

lw $ra,0($sp)

addi $sp,$sp,40

sw $v0,36($sp)

lw $v0, 36($sp)

jr $ra

label4:

main:

addi $sp, $sp, -0

move $t6, $sp

addi $sp, $sp, -36

li $t3, 10

sw $t3, 0($sp)

lw $t1, 0($sp)

move $t3, $t1

sw $t3, 12($sp)

label5:

li $t3, 20

sw $t3, 0($sp)

lw $t1, 0($sp)

move $t3, $t1

sw $t3, 16($sp)

label6:

move $t0,$sp

addi $sp, $sp, -40

sw $ra,0($sp)

lw $t1, 12($t0)

move $t3,$t1

sw $t3,12($sp)

lw $t1, 16($t0)

move $t3,$t1

sw $t3,16($sp)

jal func

lw $ra,0($sp)

addi $sp,$sp,40

sw $v0,0($sp)

lw $t1, 0($sp)

move $t3, $t1

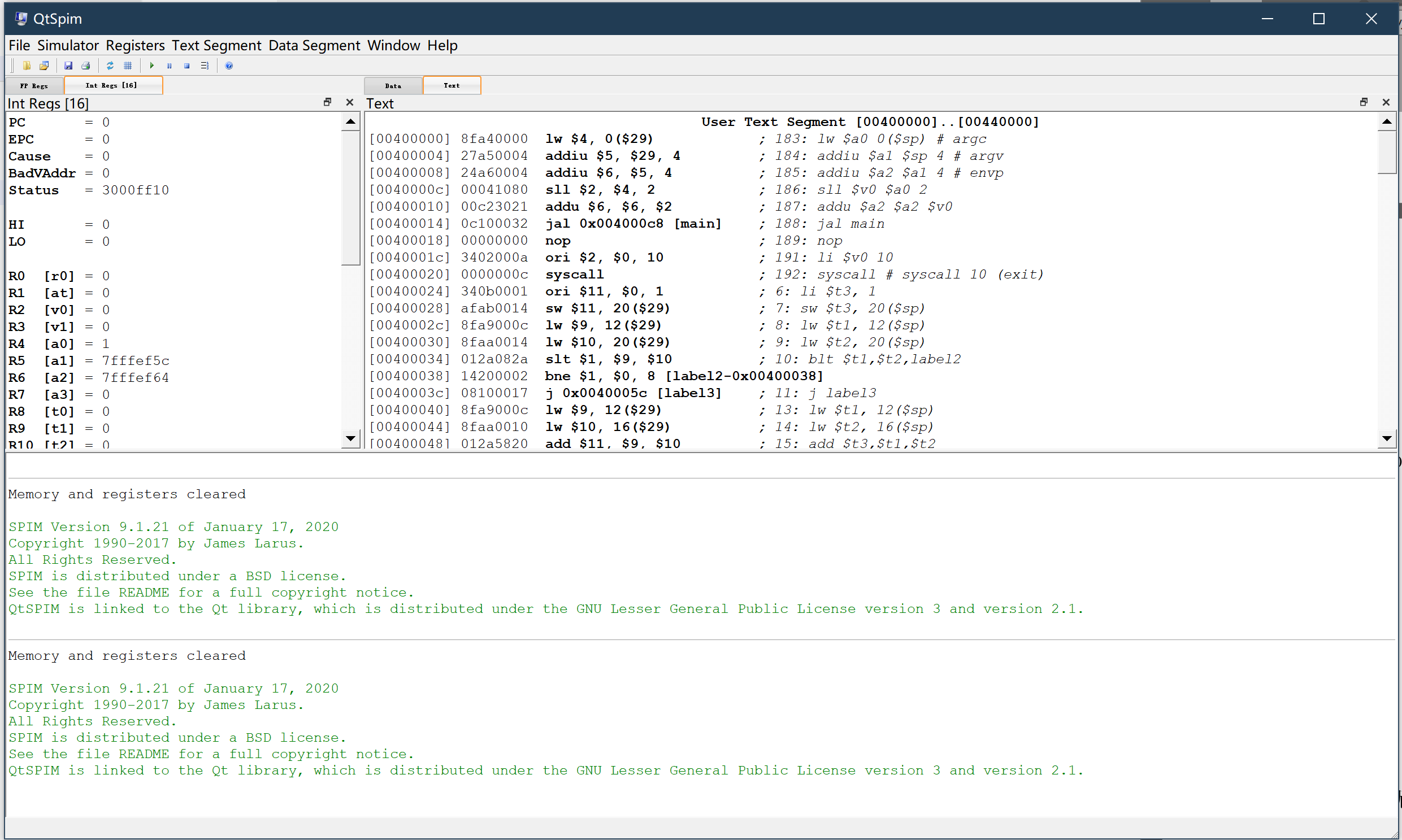
sw $t3, 8($sp)

label7:

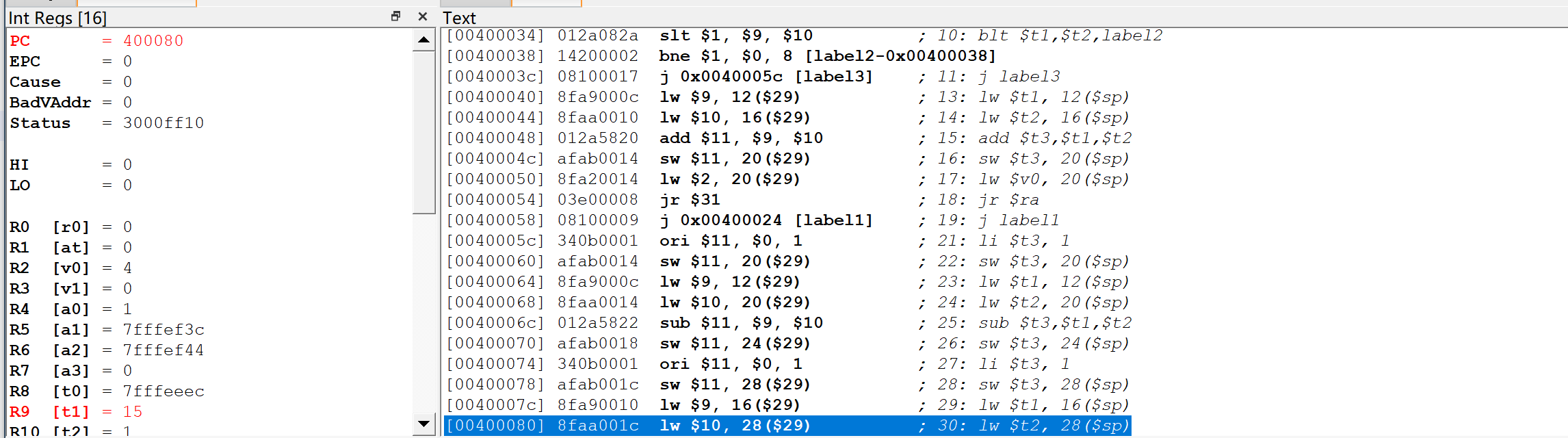
lw $v0, 8($sp)

jr $ra

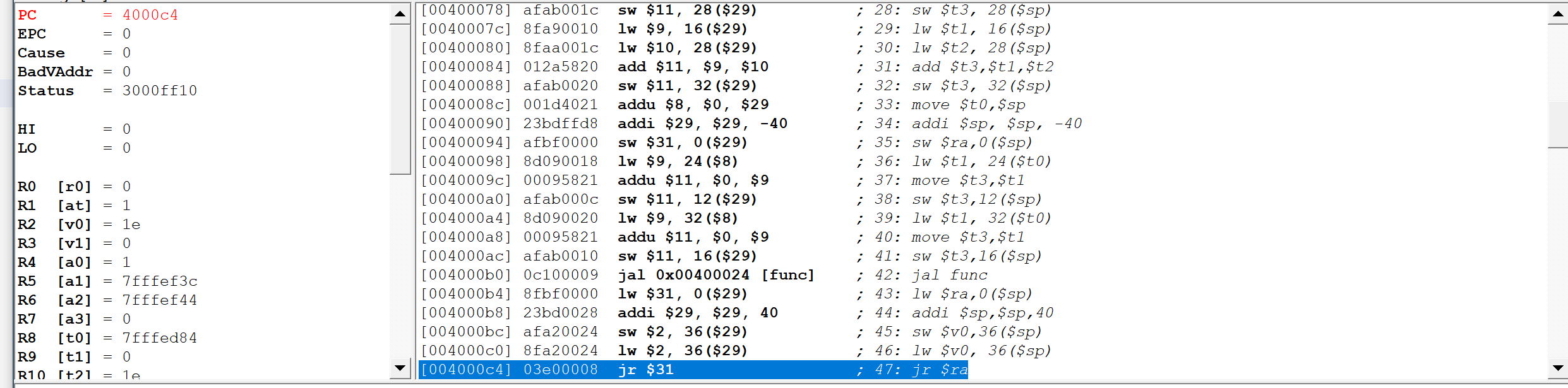
将以上代码存为code.s文件，在QtSpim里打开，按F10单步运行：



可以观察到代码一行一行执行，跳转等操作正常，寄存器值变化正常：



分析实验三测试代码不难看出这是一个递归程序，最终当a减到0时，返回b的值。输入a=10时。相当于最后将返回b+10的结果，即30。现在验证这一结果1，继续单步运行程序直到进入停机逻辑：



此时可以发现R2[v0]寄存器值为0x1e，即30，符合运行预期，程序运行正常，说明中间代码转化出的汇编代码是正确的。

至此，可以看出实验1-4的结果均是正确的，正常的。

## 4.2 系统的优点

系统在普通mini C语言的基础上，还增加了单行注释、多行注释、char类型、continue/break跳转逻辑实现等功能。 能够检测任务书所给的所有我的语言可以识别的错误并给出正确的提示信息。 能够清晰并且详细的展示编译过程中每一个步骤的结果和中间符号表等。

## 4.3 系统的缺点

虽然在前期定义了词法的识别，但是因为技术原因，在后续实验中逐渐削减了这些功能，最终实验三没有实现多维数组、结构体、联合等高级数据结构，实验四也不能正确转化数组相关的中间代码。

# 5实验小结或体会

这次编译原理实验总体上来说难度是偏大的，很考验关于课程所学内容的理解和运用；同时也很考验逻辑思维能力，因为实验一的词法设计和语法定义上如果出现一些冗余和失误，在后续发现了要去修改，或者说为一个不太好的语法去做支持是非常痛苦的。现在看来的确实验一的时候就应该认真构思好语法的实现，做好相关支持的准备。我的数组的实现就因为在实验一里面定义上出现了一些问题，导致后续对多维数组的支持变得很困难。还有就是实验三和实验二都要求对ast节点和code节点的运作方式具有较深的理解和清晰的认识，我在一开始的时候只看了一两遍视频就着手实验，对于codenode里的各个属性和整个中间代码生成流程并不熟悉，导致遇到很多不会做的地方，只能采用参考示例代码和别人的实现，但是和自己的程序也不是完全契合，在这个过程中才慢慢理顺了中间代码生成，属性计算等的实现过程，也算是走了一些弯路。总的来说各种各样的问题的出现都是因为自己对于实验视频所讲的知识和课内知识的结合联系不够紧密，对于知识还是需要多多熟练，才能在运用的时候得心应手。

通过这次实验，我了解了编译程序完整运行的流程和设计过程；详细的体会了词法分析、语法分析、语义分析、中间代码生成和目标代码生成各个步骤的具体实现方法。加深对理论课上所讲的相关概念的理解，同时这次实验的一大特点就是代码结构十分复杂，这次的编码任务也很大地提升了我的代码能力和动手能力。