**课 程 实验 报 告**

**题目： C语言编译器设计与实现**

**课程名称： 编译技术实验**

**专业班级： .........**

**学 号： ..........**

**姓 名： zh**

**指导教师： hwq、zjh、xlp**

**报告日期： ..........**

目录

[1概述 1](#_Toc4650)

[2系统描述 2](#_Toc10390)

[2.1自定义语言概述 2](#_Toc8420)

[2.2单词文法与语言文法 2](#_Toc3061)

[2.3 符号表结构定义 7](#_Toc31428)

[2.4 错误类型码定义 8](#_Toc23170)

[2.5中间代码结构定义 9](#_Toc16949)

[2.6目标代码指令集选择 10](#_Toc5129)

[3系统设计与实现 11](#_Toc27318)

[3.1 词法分析器（详情参考源码:lex.l） 11](#_Toc3428)

[3.2 语法分析器（详情参考源码:parser.y） 11](#_Toc20396)

[3.3 符号表管理 12](#_Toc28303)

[3.4 语义检查 13](#_Toc29348)

[3.5 报错功能 14](#_Toc20062)

[3.6 中间代码生成（实验三） 15](#_Toc9394)

[3.7 代码优化（实验三） 16](#_Toc16604)

[3.8 汇编代码生成（实验四） 16](#_Toc15323)

[4系统测试与评价 19](#_Toc24888)

[5实验小结或体会 47](#_Toc19344)

[参考文献 47](#_Toc29395)

[[1] 王生元 等. 编译原理(第三版). 北京：清华大学出版社，20016 47](#_Toc9177)

[[2] 胡伦俊等. 编译原理(第二版). 北京：电子工业出版社，2005 47](#_Toc20294)

[[3] 王元珍等. 80X86汇编语言程序设计. 武汉：华中科技大学出版社,2005 47](#_Toc14513)

[[4] 王雷等. 编译原理课程设计. 北京：机械工业出版社，2005 47](#_Toc975)

[[5] 曹计昌等. C语言程序设计. 北京：科学出版社，2008 47](#_Toc27909)

[附件： 48](#_Toc28582)

[源代码篇幅较长，已上传至网站https://shixun.educoder.net/ 48](#_Toc4904)

# 1概述

本次实验的目的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。在本次实验中，我选择实现的是**C语言的编译器**。

实验的任务主要是通过对简单编译器的完整实现，加深课程中关键算法的理解，提高系统软件研发技术。

# 2系统描述

### 2.1自定义语言概述

实验中的源语言是一个简化过的C语言，已实现的语言成分如下：

* 数据类型：char类型、int类型、float类型、字符串；
* 基本运算：算术运算、比较运算、自增自减运算和复合赋值运算；
* 控制语句：if语句、while和for和switch语句、break和continue语句；
* 数组：多维数组；
* 注释：支持行注释与块注释。

### 2.2单词文法与语言文法

**2.2.1 单词文法**

为了方便对照，下面展示了lex.l文件中关于单词的正则表达式部分，更详细的内容可以参考源代码中的lex.l文件。

id [A-Za-z][A-Za-z0-9]\*

int [0-9]+

float ([0-9]\*\.[0-9]+)|([0-9]+\.)

char '.'

string \"(\\.|[^\"])\*\"

annotation1 \/\/[^\n]\*

annotation2 \/\\*(\s|.|\r|\n)\*?\\*\/

%%

{annotation1} {printf("%s%s\n","行注释：",yytext); } /\* 行注释 \*/

{annotation2} {printf("%s%s\n","块注释：",yytext); } /\* 块注释 \*/

{int} {yylval.type\_int=atoi(yytext); return INT;} /\* int值 \*/

{float} {yylval.type\_float=atof(yytext); return FLOAT;} /\* float值 \*/

{char} {yylval.type\_char=yytext[1]; return CHAR;} /\* char值 \*/

{string} {strcpy(yylval.type\_string,yytext); return STRING;} /\* string的值

"int" {strcpy(yylval.type\_id, yytext); return TYPE;} /\* int类型 \*/

"float" {strcpy(yylval.type\_id, yytext); return TYPE;} /\* float类型 \*/

"char" {strcpy(yylval.type\_id, yytext); return TYPE;} /\* char类型 \*/

"char \*" {strcpy(yylval.type\_id, yytext); return TYPE;} /\* string类型 \*/

"const" {strcpy(yylval.type\_id, yytext); return CONST;} /\* 常数类型 \*/

"return" { return RETURN;}

"if" { return IF;}

"else" { return ELSE;}

"while" { return WHILE;}

"for" { return FOR;}

"switch" { return SWITCH;}

"case" { return CASE;}

"default" { return DEFAULT;}

"break" { strcpy(yylval.type\_id, yytext); return BREAK;}

"continue" { strcpy(yylval.type\_id, yytext); return CONTINUE;}

{id} { strcpy(yylval.type\_id, yytext); return ID;/\*由于关键字的形式也符合表示符的规则，所以把关键字的处理全部放在标识符的前面，优先识别\*/}

";" { return SEMI;}

":" { return COLON;}

"," { return COMMA;}

">"|"<"|">="|"<="|"=="|"!=" { strcpy(yylval.type\_id, yytext);;return RELOP;}

"=" { return ASSIGNOP;}

"+" { return PLUS;}

"++" { return SELFPLUS;}

"-" { return MINUS;}

"--" { return SELFMINUS;}

"\*" { 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] {}

**2.22 语言文法**

为了方便对照，下面展示了parser.y文件中的一部分内容，为了避免报告内容过于杂乱，这一部分内容省略了语义动作而重点展示了语言文法，更详细的内容可以参考源代码中的parser.y文件。

/\*整个程序\*/

program: ExtDefList

/\* 外部定义列表（函数和变量） \*/

ExtDefList: {$$=NULL;}

| ExtDef ExtDefList

;

/\* 外部定义（变量） \*/

ExtDef: Specifier ExtDecList SEMI

|Specifier FuncDec CompSt

|ConstSpecifier Dec SEMI

| error SEMI ;

/\* 类型 \*/

Specifier: TYPE

;

/\* 常量类型 \*/

ConstSpecifier: CONST TYPE

;

/\* 变量名列表 \*/

ExtDecList: VarDec

| VarDec COMMA ExtDecList

;

/\* 变量（标识符）声明，包括数组 \*/

VarDec: ID

| ID ARRAYLIST

;

/\* 数组索引列表 [2][2]\*/

ARRAYLIST: LB Exp RB

| LB Exp RB ARRAYLIST

;

/\* 函数声明 \*/

FuncDec: ID LP VarList RP

|ID LP RP

| error RC

;

/\* 函数参数声明列表 \*/

VarList: ParamDec

| ParamDec COMMA VarList

| error RC

;

/\* 函数参数声明 \*/

ParamDec: Specifier VarDec

;

/\* 复合语句 \*/

CompSt: LC DefList StmList RC

;

/\* 声明列表 一系列语句或语句块 \*/

StmList: {$$=NULL; }

| Stmt StmList

;

/\* 声明 语句或语句块 \*/

Stmt: Exp SEMI

| CompSt

| RETURN Exp SEMI

| IF LP Exp RP Stmt %prec LOWER\_THEN\_ELSE

| IF LP Exp RP Stmt ELSE Stmt

| WHILE LP Exp RP Stmt

| FOR LP Exp SEMI Exp SEMI Exp RP Stmt

| CASE Exp COLON Stmt

| DEFAULT COLON Stmt

| SWITCH LP Exp RP Stmt

| SKIP SEMI

;

/\* 跳转语句 \*/

SKIP: BREAK

| CONTINUE

;

/\* 局部变量定义列表 \*/

DefList: {$$=NULL; }

| Def DefList

| error SEMI

;

/\* 局部变量定义 \*/

Def: Specifier DecList SEMI

;

/\*局部变量名列表\*/

DecList: Dec

| Dec COMMA DecList

;

/\* 赋值语句（用Exp，包含变量和常量）\*/

Dec: VarDec

| VarDec ASSIGNOP Exp

;

/\* 表达式 \*/

Exp: Exp ASSIGNOP Exp

| Exp AND Exp

| Exp OR Exp

| Exp RELOP Exp

| Exp PLUS Exp

| Exp SELFPLUS

| SELFPLUS Exp

| Exp MINUS Exp

| Exp SELFMINUS

| SELFMINUS Exp

| Exp STAR Exp

| Exp DIV Exp

| LP Exp RP

| MINUS Exp %prec UMINUS

| NOT Exp

| DPLUS Exp

| Exp DPLUS

| ID LP Args RP

| ID LP RP

| ID

| INT

| FLOAT

| CHAR

| STRING

| ID ARRAYLIST

| Exp PLUS ASSIGNOP Exp

| Exp MINUS ASSIGNOP Exp

| Exp STAR ASSIGNOP Exp

| Exp DIV ASSIGNOP Exp

;

Args: Exp COMMA Args

| Exp

;

### 2.3 符号表结构定义

本实验采用的符号表结构是**顺序表，**结构体设计如下：

//一个符号表项

struct symbol { //这里只列出了一个符号表项的部分属性，没考虑属性间的互斥

char name[33]; //变量或函数名

int level; //层号

int type; //变量类型或函数返回值类型

int paramnum; //对函数适用，记录形式参数个数 个人理解为对数组也适用， 数组的每个值就相当于

int dimension[2]; //对数组适用，记录数组维的大小或者索引 int数组的默认值是 0，代表没有这个维

int dimensionNum; //对数组适用，记录数组维的数目

char alias[10]; //别名，为解决嵌套层次使用

char flag; //符号标记，函数：'F' 变量：'V' 参数：'P' 临时变量：'T' 常 量： 'C' 数组：'A' 数组项：'I'

char offset; //外部变量和局部变量在其静态数据区或活动记录中的偏移量，

//或记录函数活动记录大小，目标代码生成时使用

};

//符号表

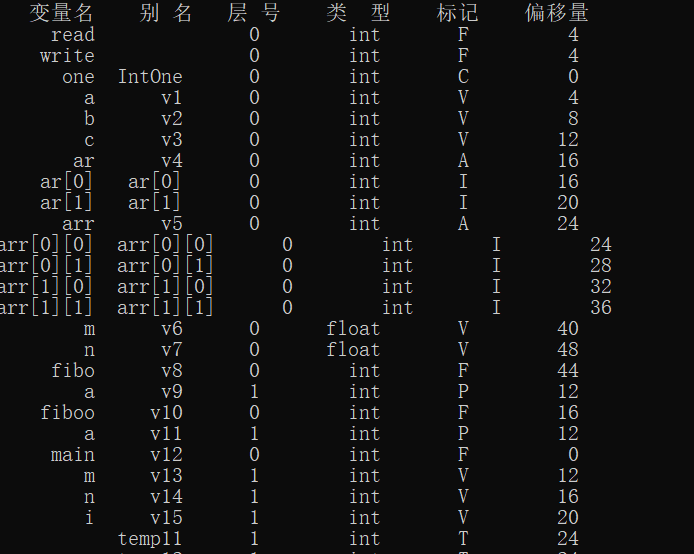
struct symboltable{

struct symbol symbols[MAXLENGTH];//符号数组

int index;//符号位置索引

} symbolTable;

符号表symbolTable是一个顺序栈，栈顶指针index初值为0，每次填写符号时，将新的符号填写到栈顶位置，再栈顶指针加1。符号表的具体情形如下图：



**各个数据类型的宽度：**

Int ： 4

Float ： 8

Char ：1

String ： 10（默认大小）

数组 ： 维数之积\*数据类型宽度

**标记的含义：**

1. F 函数
2. P 函数参数
3. V 变量
4. C 常量
5. A数组名
6. I 具体的数组项
7. T 临时变量

**个性化设计的数组在符号表中的记录：**

1. 数组名和第一个数组项对应同一个地址（符合C语言的习惯）；
2. 数组名的标记是A，数组项的标记是I；
3. 将每个数组项插入符号表，便于按索引调用时的查找以和越界判断。

### 2.4 错误类型码定义

实验中共识别了17项语义错误，依次展示如下：

1. 定义：变量重复定义

2. 定义：函数重复定义

3. 定义：使用未定义的变量

4. 定义：调用未定义或未声明的函数

5. 数组：数组下标类型错误

6. 数组：对普通变量采用下标变量的形式访问

7. 数组：对函数使用下标访问

8. 数组：对常量使用下标访问

9. 函数：对非函数名采用函数调用形式

10. 函数：对函数名采用非函数调用形式（变量）访问

11. 函数：函数返回值类型错误

12. 函数：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

13. 函数：函数调用时实参和形参类型不匹配

14. 赋值表达式：赋值号左边不是左值表达式

15. 运算表达式：对非左值表达式进行自增、自减运算

16. 跳转语句：循环外使用break

17. 跳转语句：循环外使用continue

### 2.5中间代码结构定义

采用三地址代码TAC作为中间语言，中间语言代码的定义如表1所示。

**表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 |  | y |
| 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 | x |
| RETURN x | 返回语句 | RETURN |  |  | x |
| ARG x | 传实参x | ARG |  |  | x |
| x:=CALL f | 调用函数(有返回值) | CALL | f |  | x |
| CALL f | 调用函数(无返回值) | CALL | f |  |  |
| PARAM x | 函数形参 | PARAM |  |  | x |

三地址中间代码TAC是一个4元组，逻辑上包含（op、opn1、opn2、result），其中op表示操作类型说明，opn1和opn2表示2个操作数，result表示运算结果。

### 2.6目标代码指令集选择

我选择的目标语言是MIPS32指令序列， TAC指令和MIPS32指令的对应关系如表2所示。其中reg(x)表示变量x所分配的寄存器。

**表2 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 |

# 3系统设计与实现

1. **词法分析器**（详情参考源码:lex.l）

基本思路：根据语言的词法规则，按Flex的格式，编辑Lex.l文件，主要是在规则部分用正则表达式识别单词并附加相应的动作。使用Flex编译后即可得到词法分析源程序Lex.yy.c，通过调用yylex（）进行词法分析，每当识别出一个单词，将该单词的（单词种类码，单词的自身值）输出给语法分析程序。

1. **语法分析器**（详情参考源码:parser.y）

基本思路：根据语言的语法规则，按Bison的格式，编辑Parser.y文件，主要是编写语言文法并附加相应的语义动作，以及解决一些移进~归约冲突（by 设置优先级），用Bison编译后即可得语法分析源程序Parser.tab.c，调用parser（）进行语法分析。

基于简单和效率方面的考虑，在清楚了词法、语法分析算法原理的基础上，我选择通过联合使用2个工具（Flex和Bison）来构造词法、语法分析程序，语法正确后生成抽象语法树。

词法和语法分析器联合在一起完成词法与语法分析时，要求统一单词的种类编码，这时可将各个单词在parser.y中逐个以标识符的形式，通过%token罗列出来。在Parser.y文件中，这些标识符在语法规则部分，作为语法规则的终结符；同时用Bison编译后，可生成一个文件Parser.tab.h，该文件中将这些标识符定义为枚举常量，每一个就对应一个（类）单词，这些枚举常量提供给Lex.l使用，每当识别出一类单词时，就可返回对应的种类码（枚举常量）。联合使用FLEX和Bison构造词法、语法分析器的工作流程图示意图如下所示。



1. **符号表管理**

基本思路：通过先根遍历AST完成符号表的管理:在语义分析过程中，访问到了说明部分的结点时，在符号表中添加新的内容；在中间代码生成阶段，会生成一些临时变量并且插入符号表中。

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0；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C语言中允许有复合语句，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，是否有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结点COM\_STM的所有子树，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（如何修改：应该是要用这个起点+该作用域的宽度），同时LEV减一，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，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（默认假设：靠近栈顶的作用域是距离当前作用域最近的作用域，因此从栈顶开始查询），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）根据变量名查找的是一个函数。等等，需要做出各种检查。

符号表的操作函数如下图所示：



1. **语义检查**

基本思路：通过先根遍历AST完成静态语义分析:访问到执行语句部分时，如果执行语句涉及变量（或函数），则根据访问的变量（或函数）名称查询符号表。如果执行语句不涉及变量和函数，进行一些其他的检查（如：对常量进行自增、自减就是未涉及变量和函数的错误执行语句）。

在实验二中，总共识别了17种语义错误：

1. 定义：变量重复定义

2. 定义：函数重复定义

3. 定义：使用未定义的变量

4. 定义：调用未定义或未声明的函数

5. 数组：数组下标类型错误

6. 数组：对普通变量采用下标变量的形式访问

7. 数组：对函数使用下标访问

8. 数组：对常量使用下标访问

9. 函数：对非函数名采用函数调用形式

10. 函数：对函数名采用非函数调用形式（变量）访问

11. 函数：函数返回值类型错误

12. 函数：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

13. 函数：函数调用时实参和形参类型不匹配

14. 赋值表达式：赋值号左边不是左值表达式

15. 运算表达式：对非左值表达式进行自增、自减运算

16. 跳转语句：循环外使用break

17. 跳转语句：循环外使用continue

1. **报错功能**

报错可能发生在三个阶段：

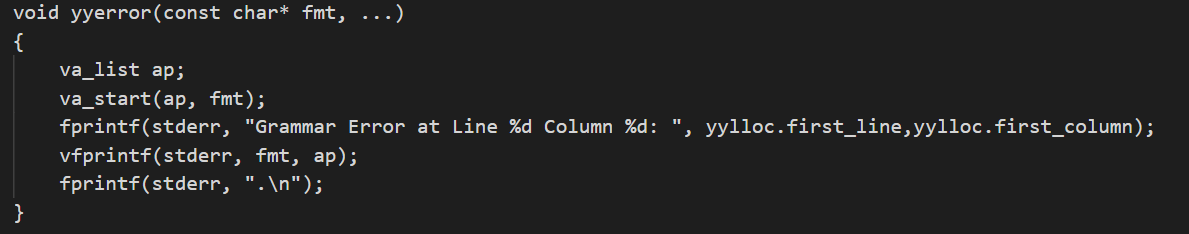
**词法分析阶段**：词法分析过程中，一旦遇到识别不了的单词，需要进行报错。处理方法很简单，在FLEX文件的规则部分，前面是能识别出来的所有单词的正则式，最后一个条就是一个符号”.”表示的正则式，表示前面不能识别出的单词形式（即：错误单词形式），这时结合变量yylineno给出错误信息即可。



**语法分析阶段**：由BISON文件中的yyerror函数负责完成，当由词法分析程序得到的单词序列不能满足parser.y文件中的语言文法时，结合yylineno给出错误信息，报告错误调用的函数如下图所示。由于待编译的源程序可能会有多个语法错误，这时，需要有容错的功能，跳过错误的代码段，继续向后进行语法分析。这时可通过跳过一段源程序代码段到指定的符号，再接着进行语法分析。例如：

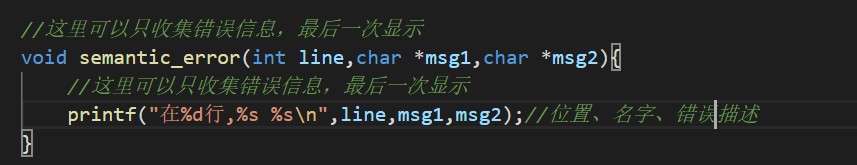
Stm →error SEMI

表示对语句进行语法分析时，一旦有错，跳过分号（SEMI），继续向后进行语法分析。可在parser.y文件中多处设置这种同步操作。



**静态语义检查阶段**：在静态语义检查过程中，如果发现了语义错误就通过semantic\_error()函数输出：错误位置、变量名、错误描述。

semantic\_error()函数如下图：



1. **中间代码生成**

基本思路：通过对AST进行遍历，计算相关的继承属性和综合属性的值，利用符号表，生成以三地址代码TAC作为中间语言的中间语言代码序列。

考虑到目标代码的生成要以中间代码为基础，因此需要在中间代码生成时获取一些信息。需要特别注意的是偏移量，因为目标代码要根据偏移量来取变量的值，一旦偏移量出错就会导致目标代码取出来的值出错。

三地址中间代码TAC是一个4元组，逻辑上包含（op、opn1、opn2、result），其中op表示操作类型说明，opn1和opn2表示2个操作数，result表示运算结果。对于不同的C语言语句，操作数个数不定，实验过程中需要针对不同的C语言语句，适当地使用op、opn1、opn2和result。

TAC在头文件里的结构组成：

struct opn{

int kind; //标识联合成员的属性

int type; //标识操作数的数据类型

union {

int const\_int; //整常数值，立即数

float const\_float; //浮点常数值，立即数

char const\_char; //字符常数值，立即数

char string[50]; //字符串数值，立即数

char id[33]; //变量或临时变量的别名或标号字 符串

};

int level; //变量的层号，0表示是全局变量，数据保存在静 态数据区

int offset; //偏移量，目标代码生成时用

};

struct codenode { //三地址TAC代码结点,采用单链表存放中间语 言代码

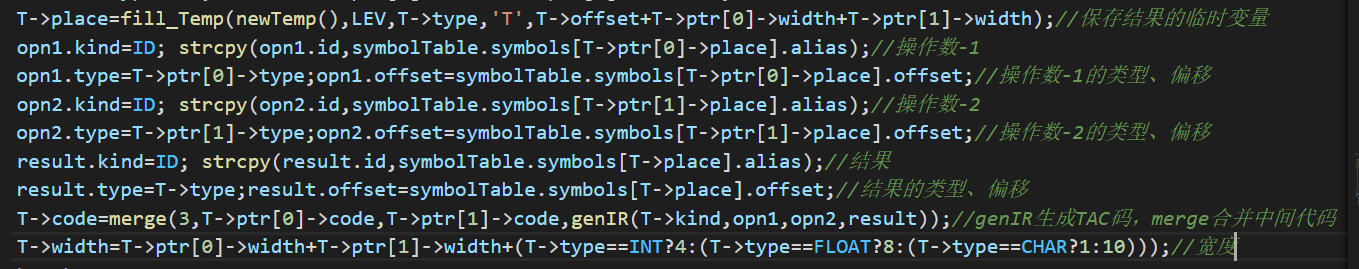
int op;

struct opn opn1,opn2,result;

struct codenode \*next,\*prior;

};

下面给出一个注释过的源码截图，展示对于TAC四元组的具体操作：



关于中间代码生成的程序设计，值得一提的是多维数组调用的中间代码生成。按照前文提到的符号表管理的自定义设计（3.3节），在定义数组的时候，就已经根据数组的维数，将所有的数组项（数组项在符号表里的名字=数组名+对应索引）一一插入了符号表。所以在识别到数组调用时，在将数组名和索引拼接之后，就可以按照与标识符（即：ID）非常类似的步骤生成其中间代码了。

需要注意的是，在中间代码生成阶段，生成了许多临时变量。这些临时变量用于临时保存运算或者赋值的结果，这些临时变量的值最终往往会赋值给之前在符号表中已经保存了的变量或者函数参数或者数组项。

1. **代码优化**

未进行代码优化。

1. **汇编代码生成**

基本思路：选择MIPS32位指令作为目标语言，采用朴素寄存器分配。根据TAC码结点（struct codenode）的op属性判断该TAC码对应的操作类型，然后根据不同的操作类型进行不同的目标代码生成方式。

在这个阶段对操作数取值是一个很重要的工作，大体上分为对立即数取值和对变量取值，通过中间代码生成阶段保存的opn1或者opn2的const\_int属性可以直接获取立即数的值，通过中间代码生成阶段保存的offset属性可以得到操作数的地址和结果的保存地址，进而取值或存值。

当选择朴素的寄存器分配方案后，目标代码生成时，每当运算操作时，都需要将操作数读入到寄存器中，运算结束后将结果写到对应的单元。由于选择朴素的寄存器分配，只会用到几个寄存器，这里约定操作数使用$t1和$t2，运算结果使用$t3，翻译的方法如表3所示。

**表3 朴素寄存器分配的翻译**

|  |  |
| --- | --- |
| **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 |  |

对于函数调用X:=CALL f，需要完成开辟活动记录的空间、参数的传递和保存返回地址等，函数调用返回后，需要恢复返回地址，读取函数返回值以及释放活动记录空间。具体步骤如下：

1. 首先根据符号表中函数定义项得到该函数活动记录的大小，开辟活动记录空间和保存返回地址。
2. 根据函数定义中的参数个数paramnum，即在X:=CALL f之前有paramnum个ARG形式的中间代码，可获得各个实参值所存放的单元，取出后送到形式参数的单元中。

（3） 使用jal f 转到函数f处

（4） 释放活动记录空间和恢复返回地址。

需要说明一下的是，在目标代码生成阶段，我对源语言（C）作了一定的限制。由于float和char类型的寄存器使用和int类型存在差异，加之我对MIPS汇编的熟悉程度还不够，因此在此阶段将源语言限制为int类型，但除此之外的其他功能基本上都实现了，这点可以参考后面给出的测试用例。

# 4系统测试与评价

**4.1 测试用例**

（说明：以下只给出正确测试的测试用例，测试结果在4.2节中，错误测试在4.3节中单独给出）

**4.1.1 实验一用例**

int intNum;//测试:int类型

float floNum;//测试:float类型

char c; // 测试:char类型

char \*s;// 测试:字符串类型

int array[4][4];//测试:数组类型

int total;//运算结果

int index;//数组下标索引

const float cNum=6.6;//测试:常量类型

/\*

\* fun函数说明

\*/

int fun (int a, float b ,char cc,char \*ss)

{

int m=100;

//break;//语法错误展示

if (a>b) {

m=a;

intNum=666666;

floNum=0.999999;

c='a';

s="abc";

total=a+b;//测试:基本运算

total=a-b;

total=a\*b;

total=a/b;

total+=1;//测试:复合运算

total-=1;

total\*=1;

total/=1;

index=total;

floNum=222+666;//测试:自动类型转换

floNum=0.222+0.666;

floNum=0.222+666;

floNum=222+0.666;

array[0][1]=1;//测试:数组赋值

}

else m=10;

while (a<10)

{

if (a==5)

{

a=a+2;

}else

{

//2++;//语法错误展示

break;

}

}

for (a=0;b<10;b++)

{

b=b+1;

continue;

}

switch (a)

{

case 1:

{

a=a+1;

break;

}

case 2:

{

a=a+2;

break;

}

default:

{

a\*=3;

break;

}

}

return m;

}

**4.1.2 实验二用例**

int a;

int array[3][5];

char c;

char c;//错误：变量重复定义

char \*str;

float m,n;

const int ci=1;

int fibo(int a){

if (a==1||a==2)

return 1;

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

}

int fibo(int a){

//错误：函数重复定义

}

char fiboc(char cc){

array['a'][0]=1;//错误：非法类型的数组下标

fibo[0][0]=1;//错误：对函数使用下标访问

m[0][0]=2;//错误：对非数组变量使用下标访问

fun(a);//错误：使用未定义函数

fibo();//错误：函数调用参数过少

fibo(c);//错误：函数调用参数类型不匹配

fibo(a,c);//错误：函数调用参数过多

m();//错误：误用变量名作函数名

a=fibo;//错误：误用函数作为普通变量

2++;//错误：对常量使用自增、自减

return 'a';

}

int main(){

int m,n,i;

m=read();

i=1;

xxx=2;//错误：使用未定义变量

2++=1;//错误：赋值表达式左值

2=1;//错误：赋值表达式左值

break;//错误：循环外的break

continue;//错误：循环外的continue

while (i<m)

{

//continue;

n=fibo(i);

write(n);

i=i+1;

break;

}

return 'a';//函数返回值类型错误

}

**4.1.3 实验三用例**

int a;

int ar[2];

int arr[2][2];

char c;

char \*str;

float m,n;

const int ci=1;

int fibo(int a,int b){

if (a==1||a==2) //测试：包含多个条件的条件转移语句的中间代码

return 1;

return fibo(a-1,1)+fibo(a-2,1);//测试：多个函数一起调用的中间代码

}

int main(){

int m,n,i;

ar[1]=1;//测试：一维数组调用的中间代码

arr[0][1]=2;//测试：二维数组调用的中间代码

m=read();

i=1;

i++;//测试：一元+1和二元+1的中间代码

i=i+1;//测试：一元+1和二元+1的中间代码

while (i<m)//测试：循环的中间代码

{

n=fibo(i,n);

write(n);

if(i<3)

continue;//测试：continue跳转

i=i+1;

if(i>3)

break;//测试：break跳转

}

return m;

}

**4.1.4 实验四用例**

int a,b,c;

int ar[2];

int arr[2][2];

float m,n;

int fibo(int a){

if (a==1||a==2)

return 1;

return fibo(a-1)+fibo(a-2);//测试：递归函数

}

int fiboo(int a){

return a;

}

int main(){

int m,n,i;

ar[1]=1;//测试：一维数组调用

arr[0][1]=2;//测试：二维数组调用

write(ar[1]+arr[0][1]);//测试：多个数组之间的运算 + 函数调用

m=read();//输入m的值 Enter an intege

i=1;

while (i<m)

{

if (i==2)

{

i++;

continue;//测试: continue ok i==2时就会执行continue直接进入下一轮循环==> 2不会被打印显示

}

n=fiboo(i);

write(n);//输出n的值

if (i==8)

{

break;//测试: break ok i==8时就会执行break退出循环 ==> 8之后的数不会被打印显示

}

i=i+1;

}

return 1;

}

**4.2 正确性测试**

**4.2.1 实验一测试结果**

TYPE：(261 , int)

ID：(259 , a)

COMOA：(276 , ,)

ID：(259 , b)

COMOA：(276 , ,)

ID：(259 , c)

SEMI：(275 , ;)

TYPE：(261 , float)

ID：(259 , m)

COMOA：(276 , ,)

ID：(259 , n)

SEMI：(275 , ;)

TYPE：(261 , const)

TYPE：(261 , int)

ID：(259 , ci)

ASSIGNOP：(285 , =)

INT：(258 , 1)

SEMI：(275 , ;)

TYPE：(261 , int)

ID：(259 , fibo)

LP：(269 , ()

TYPE：(261 , int)

ID：(259 , a)

RP：(270 , ))

LC：(273 , {)

IF：(293 , if)

LP：(269 , ()

ID：(259 , a)

RELOP：(260 , ==)

INT：(258 , 1)

OR：(291 , ||)

ID：(259 , a)

RELOP：(260 , ==)

INT：(258 , 2)

RP：(270 , ))

RETURN：(296 , return)

INT：(258 , 1)

SEMI：(275 , ;)

RETURN：(296 , return)

ID：(259 , fibo)

LP：(269 , ()

ID：(259 , a)

MINUS：(281 , -)

INT：(258 , 1)

RP：(270 , ))

PLUS：(279 , +)

ID：(259 , fibo)

LP：(269 , ()

ID：(259 , a)

MINUS：(281 , -)

INT：(258 , 2)

RP：(270 , ))

SEMI：(275 , ;)

RC：(274 , })

TYPE：(261 , int)

ID：(259 , main)

LP：(269 , ()

RP：(270 , ))

LC：(273 , {)

TYPE：(261 , int)

ID：(259 , m)

COMOA：(276 , ,)

ID：(259 , n)

COMOA：(276 , ,)

ID：(259 , i)

COMOA：(276 , ,)

ID：(259 , array)

LB：(271 , [)

INT：(258 , 10)

RB：(272 , ])

LB：(271 , [)

INT：(258 , 10)

RB：(272 , ])

SEMI：(275 , ;)

ID：(259 , m)

ASSIGNOP：(285 , =)

ID：(259 , read)

LP：(269 , ()

RP：(270 , ))

SEMI：(275 , ;)

ID：(259 , i)

ASSIGNOP：(285 , =)

INT：(258 , 1)

SEMI：(275 , ;)

WHILE：(295 , while)

LP：(269 , ()

ID：(259 , i)

RELOP：(260 , <)

ID：(259 , m)

RP：(270 , ))

LC：(273 , {)

ID：(259 , n)

ASSIGNOP：(285 , =)

ID：(259 , fibo)

LP：(269 , ()

ID：(259 , i)

RP：(270 , ))

SEMI：(275 , ;)

ID：(259 , write)

LP：(269 , ()

ID：(259 , n)

RP：(270 , ))

SEMI：(275 , ;)

ID：(259 , i)

ASSIGNOP：(285 , =)

ID：(259 , i)

PLUS：(279 , +)

INT：(258 , 1)

SEMI：(275 , ;)

ID：(259 , array)

LB：(271 , [)

INT：(258 , 0)

RB：(272 , ])

LB：(271 , [)

INT：(258 , 0)

RB：(272 , ])

ASSIGNOP：(285 , =)

INT：(258 , 1)

SEMI：(275 , ;)

RC：(274 , })

RETURN：(296 , return)

INT：(258 , 1)

SEMI：(275 , ;)

RC：(274 , })

外部变量定义：(1)

类型： int

变量名：

ID： a

ID： b

ID： c

外部变量定义：(2)

类型： float

变量名：

ID： m

ID： n

常量定义：(3)

常量类型： int

声明赋值：

ASSIGNOP

ID： ci

INT：1

函数定义：(8)

类型： int

函数名：fibo

函数形参：

类型：int, 参数名：a

复合语句：(8)

复合语句的变量定义部分：

复合语句的语句部分：

条件语句(IF\_THEN)：(7)

条件：

OR

==

ID： a

INT：1

==

ID： a

INT：2

IF子句：(7)

返回语句：(6)

INT：1

返回语句：(7)

PLUS

函数调用：(7)

函数名：fibo

第1个实际参数表达式：

MINUS

ID： a

INT：1

函数调用：(7)

函数名：fibo

第1个实际参数表达式：

MINUS

ID： a

INT：2

函数定义：(21)

类型： int

函数名：main

无参函数

复合语句：(21)

复合语句的变量定义部分：

局部变量定义：(10)

类型： int

变量名：

m

n

i

array

复合语句的语句部分：

表达式语句：(11)

ASSIGNOP

ID： m

函数调用：(11)

函数名：read

表达式语句：(12)

ASSIGNOP

ID： i

INT：1

WHILE循环语句：(19)

循环条件：

<

ID： i

ID： m

循环体：(19)

复合语句：(19)

复合语句的变量定义部分：

复合语句的语句部分：

表达式语句：(15)

ASSIGNOP

ID： n

函数调用：(15)

函数名：fibo

第1个实际参数表达式：

ID： i

表达式语句：(16)

函数调用：(16)

函数名：write

第1个实际参数表达式：

ID： n

表达式语句：(17)

ASSIGNOP

ID： i

PLUS

ID： i

INT：1

表达式语句：(18)

ASSIGNOP

ID： array[0][0]

INT：1

返回语句：(20)

INT：1

**4.2.2 实验二测试结果**

在4行,c 变量重复定义

变量名 别 名 层 号 类 型 标记 偏移量

read 0 int F 4

write 0 int F 4

x 1 int P 12

a v1 0 int V 0

array v2 0 int A 4

c v3 0 char V 64

str v5 0 char \* V 66

m v6 0 float V 76

n v7 0 float V 84

ci v8 0 int C 92

fibo v9 0 int F 0

a v10 1 int P 12

temp1 1 int T 16

temp2 1 int T 16

temp3 1 int T 16

temp4 1 int T 16

temp5 1 int T 20

temp6 1 int T 24

temp7 1 int T 28

temp8 1 int T 32

temp9 1 int T 36

temp10 1 int T 40

在13行,fibo 函数重复定义

变量名 别 名 层 号 类 型 标记 偏移量

read 0 int F 4

write 0 int F 4

x 1 int P 12

a v1 0 int V 0

array v2 0 int A 4

c v3 0 char V 64

str v5 0 char \* V 66

m v6 0 float V 76

n v7 0 float V 84

ci v8 0 int C 92

fibo v9 0 int F 44

a v10 1 int P 12

在17行,array 数组变量的下标不是整型表达式

在18行,fibo 对函数使用下标访问

在19行,m 对非数组变量使用下标访问

在20行,fun 函数未定义

在21行, 函数调用参数太少!

在22行, 函数参数类型不匹配!

在23行, 函数调用参数太多!

在24行,m 这是变量而不是函数，类型不匹配

在25行,fibo 这是函数而不是变量，类型不匹配

在26行,对常量进行自增自减

变量名 别 名 层 号 类 型 标记 偏移量

read 0 int F 12

write 0 int F 4

x 1 int P 12

a v1 0 int V 0

array v2 0 int A 4

c v3 0 char V 64

str v5 0 char \* V 66

m v6 0 float V 76

n v7 0 float V 84

ci v8 0 int C 92

fibo v9 0 int F 44

a v10 1 int P 12

fiboc v12 0 char F 0

cc v13 1 char P 12

temp11 1 int T 13

temp12 1 int T 13

temp13 1 int T 13

temp14 1 int T 13

temp15 1 int T 13

temp16 1 int T 13

在33行,xxx 变量未定义

在34行, 赋值语句需要左值

在35行, 赋值语句需要左值

在36行,在循环外非法使用break

在37行,在循环外非法使用continue

变量名 别 名 层 号 类 型 标记 偏移量

read 0 int F 12

write 0 int F 4

x 1 int P 12

a v1 0 int V 0

array v2 0 int A 4

c v3 0 char V 64

str v5 0 char \* V 66

m v6 0 float V 76

n v7 0 float V 84

ci v8 0 int C 92

fibo v9 0 int F 44

a v10 1 int P 12

fiboc v12 0 char F 17

cc v13 1 char P 12

main v14 0 int F 0

m v15 1 int V 12

n v16 1 int V 16

i v17 1 int V 20

temp17 1 int T 24

temp18 1 int T 24

temp19 1 int T 24

temp20 2 int T 24

temp21 2 int T 24

temp22 2 int T 24

temp23 2 int T 28

在46行,a 返回值与函数返回值的类型不匹配

变量名 别 名 层 号 类 型 标记 偏移量

read 0 int F 12

write 0 int F 4

x 1 int P 12

a v1 0 int V 0

array v2 0 int A 4

c v3 0 char V 64

str v5 0 char \* V 66

m v6 0 float V 76

n v7 0 float V 84

ci v8 0 int C 92

fibo v9 0 int F 44

a v10 1 int P 12

fiboc v12 0 char F 17

cc v13 1 char P 12

main v14 0 int F 0

m v15 1 int V 12

n v16 1 int V 16

i v17 1 int V 20

temp17 1 int T 24

temp18 1 int T 24

temp19 1 int T 24

**4.2.3 实验三测试结果**

（仅展示中间代码）

FUNCTION fibo :

PARAM v10

PARAM v11

temp1 := #1

IF v10 == temp1 GOTO label3

GOTO label4

LABEL label4 :

temp2 := #2

IF v10 == temp2 GOTO label3

GOTO label2

LABEL label3 :

temp3 := #1

RETURN temp3

LABEL label2 :

temp4 := #1

temp5 := v10 - temp4

temp6 := #1

ARG temp5

ARG temp6

temp7 := CALL fibo

temp8 := #2

temp9 := v10 - temp8

temp10 := #1

ARG temp9

ARG temp10

temp11 := CALL fibo

temp12 := temp7 + temp11

RETURN temp12

LABEL label1 :

FUNCTION main :

temp13 := #1

ar[1] := temp13

temp14 := #2

arr[0][1] := temp14

temp15 := CALL read

v13 := temp15

temp16 := #1

v15 := temp16

temp17 := v15 + IntOne

v15 := temp17

temp18 := #1

temp19 := v15 + temp18

v15 := temp19

LABEL label14 :

IF v15 < v13 GOTO label13

GOTO label12

LABEL label13 :

ARG v15

ARG v14

temp20 := CALL fibo

v14 := temp20

ARG v14

CALL write

temp22 := #3

IF v15 < temp22 GOTO label18

GOTO label17

LABEL label18 :

GOTO label14

LABEL label17 :

temp23 := #1

temp24 := v15 + temp23

v15 := temp24

temp25 := #3

IF v15 > temp25 GOTO label20

GOTO label14

LABEL label20 :

GOTO label12

GOTO label14

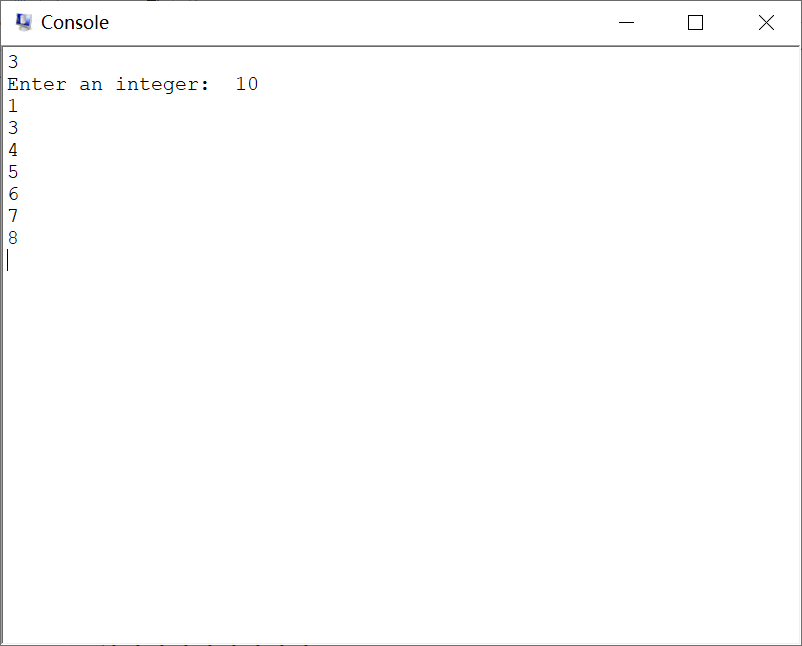
LABEL label12 :

RETURN v13

LABEL label5 :

**4.2.4 实验四测试结果**

（仅展示在QtSpim上的运行结果和目标代码）



运行结果说明：

参考4.1.4节中给出的测试代码，可知

上图中的3：

是调用多维数组项进行计算后的输出结果。

上图中的Enter an Interger：10后面的输出：

是while循环中的输出结果，其中2没有被输出是continue的作用，8后面的数字没有输出是break的作用。

.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

write:

li $v0,1

syscall

li $v0,4

la $a0,\_ret

syscall

move $v0,$0

jr $ra

fibo:

li $t3, 1

sw $t3, 16($sp)

lw $t1, 12($sp)

lw $t2, 16($sp)

beq $t1,$t2,label3

j label4

label4:

li $t3, 2

sw $t3, 16($sp)

lw $t1, 12($sp)

lw $t2, 16($sp)

beq $t1,$t2,label3

j label2

label3:

li $t3, 1

sw $t3, 16($sp)

lw $v0,16($sp)

jr $ra

label2:

li $t3, 1

sw $t3, 16($sp)

lw $t1, 12($sp)

lw $t2, 16($sp)

sub $t3,$t1,$t2

sw $t3, 20($sp)

move $t0,$sp

addi $sp, $sp, -44

sw $ra,0($sp)

lw $t1, 20($t0)

move $t3,$t1

sw $t3,12($sp)

jal fibo

lw $ra,0($sp)

addi $sp,$sp,44

sw $v0,24($sp)

li $t3, 2

sw $t3, 28($sp)

lw $t1, 12($sp)

lw $t2, 28($sp)

sub $t3,$t1,$t2

sw $t3, 32($sp)

move $t0,$sp

addi $sp, $sp, -44

sw $ra,0($sp)

lw $t1, 32($t0)

move $t3,$t1

sw $t3,12($sp)

jal fibo

lw $ra,0($sp)

addi $sp,$sp,44

sw $v0,36($sp)

lw $t1, 24($sp)

lw $t2, 36($sp)

add $t3,$t1,$t2

sw $t3, 40($sp)

lw $v0,40($sp)

jr $ra

label1:

fiboo:

lw $v0,12($sp)

jr $ra

label5:

main:

addi $sp, $sp, -32

li $t3, 1

sw $t3, 24($sp)

lw $t1, 24($sp)

move $t3, $t1

sw $t3, 20($sp)

li $t3, 2

sw $t3, 24($sp)

lw $t1, 24($sp)

move $t3, $t1

sw $t3, 28($sp)

lw $t1, 20($sp)

lw $t2, 28($sp)

add $t3,$t1,$t2

sw $t3, 24($sp)

lw $a0, 24($sp)

addi $sp, $sp, -4

sw $ra,0($sp)

jal write

lw $ra,0($sp)

addi $sp, $sp, 4

addi $sp, $sp, -4

sw $ra,0($sp)

jal read

lw $ra,0($sp)

addi $sp, $sp, 4

sw $v0, 24($sp)

lw $t1, 24($sp)

move $t3, $t1

sw $t3, 12($sp)

li $t3, 1

sw $t3, 24($sp)

lw $t1, 24($sp)

move $t3, $t1

sw $t3, 20($sp)

label14:

lw $t1, 20($sp)

lw $t2, 12($sp)

blt $t1,$t2,label13

j label12

label13:

li $t3, 2

sw $t3, 24($sp)

lw $t1, 20($sp)

lw $t2, 24($sp)

beq $t1,$t2,label16

j label15

label16:

lw $t1, 20($sp)

li $t2, 1

add $t3,$t1,$t2

sw $t3, 24($sp)

lw $t1, 24($sp)

move $t3, $t1

sw $t3, 20($sp)

j label14

label15:

move $t0,$sp

addi $sp, $sp, -16

sw $ra,0($sp)

lw $t1, 20($t0)

move $t3,$t1

sw $t3,12($sp)

jal fiboo

lw $ra,0($sp)

addi $sp,$sp,16

sw $v0,24($sp)

lw $t1, 24($sp)

move $t3, $t1

sw $t3, 16($sp)

lw $a0, 16($sp)

addi $sp, $sp, -4

sw $ra,0($sp)

jal write

lw $ra,0($sp)

addi $sp, $sp, 4

li $t3, 8

sw $t3, 24($sp)

lw $t1, 20($sp)

lw $t2, 24($sp)

beq $t1,$t2,label21

j label20

label21:

j label12

label20:

li $t3, 1

sw $t3, 24($sp)

lw $t1, 20($sp)

lw $t2, 24($sp)

add $t3,$t1,$t2

sw $t3, 28($sp)

lw $t1, 28($sp)

move $t3, $t1

sw $t3, 20($sp)

j label14

label12:

li $t3, 1

sw $t3, 24($sp)

lw $v0,24($sp)

jr $ra

label6:

**4.3 报错功能测试**

**4.3.1 词法分析时报错**

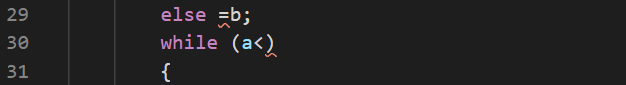
如图：当字符的单引号缺失一个时，由于无法识别字符，会在词法分析时产生报错。

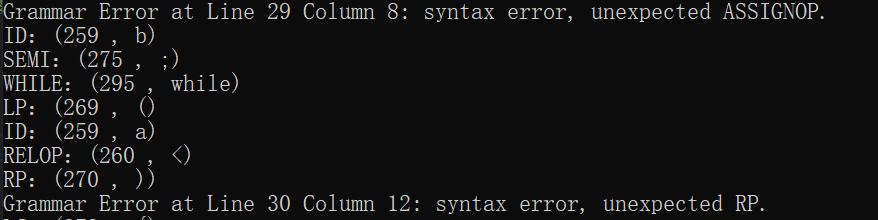




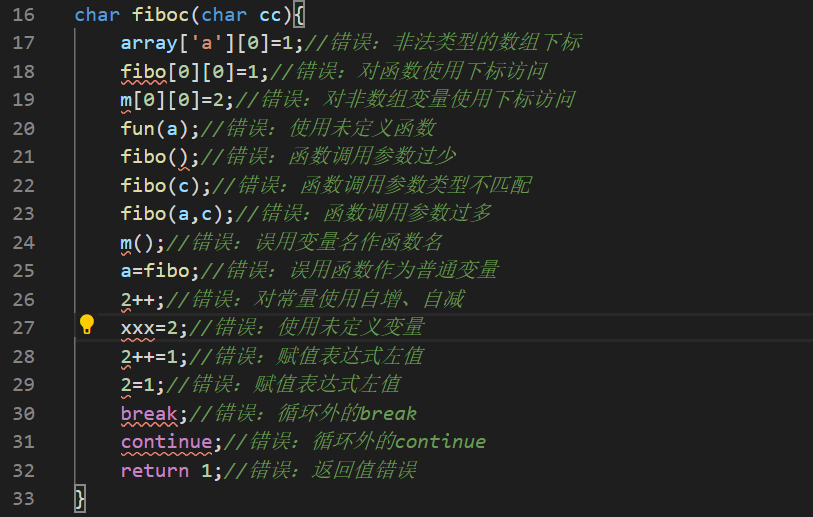
**4.3.2 语法分析时报错**

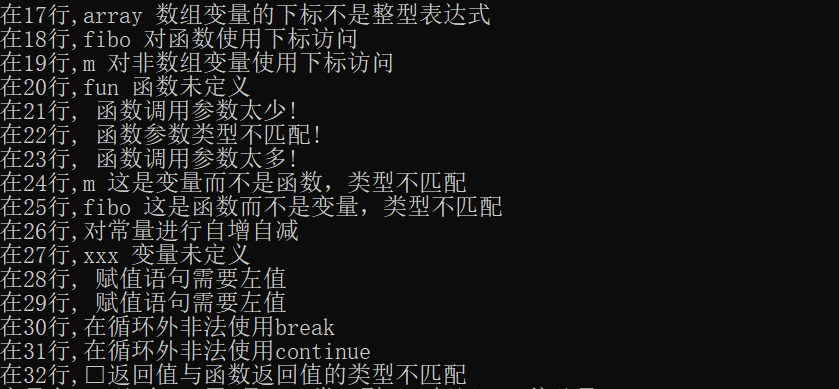
如图：29行的赋值表达式出错、30行的条件表达式出错，所以连续报了两个语法错误。





**4.3.3 语义分析时报错**





**4.4 系统的优点**

该编译器基本上完成了的C语言编译器的功能，可完成对基本C语言代码的编译工作。

**4.5 系统的缺点**

对于C语言中的一些高级功能（如：指针和结构体），该编译器还未实现。而且，在目标代码生成阶段，将源语言的基本数据类型限制为了int。

# 5实验小结或体会

通过此次编译原理实验，我大致了解了一个编译器的基本设计思路并且进一步巩固了课本上的知识。通过实验一，我熟悉了正则表达式的运用；通过实验二，我熟悉了语言文法的设计；通过实验三，我熟悉了如何将AST转化为TAC码；通过实验四，我了解了MIPS32位指令并且熟悉了如何将TAC码转化为汇编代码。除此之外，由于本实验设计的是C语言编译器，所以在实验过程中更进一步加深了我对C语言的理解。

**参考文献**

[1] 王生元 等. 编译原理(第三版). 北京：清华大学出版社，20016

[2] 胡伦俊等. 编译原理(第二版). 北京：电子工业出版社，2005

[3] 王元珍等. 80X86汇编语言程序设计. 武汉：华中科技大学出版社,2005

[4] 王雷等. 编译原理课程设计. 北京：机械工业出版社，2005

[5] 曹计昌等. C语言程序设计. 北京：科学出版社，2008