

**实验报告**

**题目： miniC - Compiler**

**课程名称： 编译原理**

**专业班级： ACM1601**

**学 号： U201614756**

**姓 名： 江易星**

**指导教师： 徐丽萍**

**报告日期： 2019/06/18**

**计算机科学与技术学院**

**目录**

[1选题背景 1](#_Toc12201945)

[2系统关键定义 2](#_Toc12201946)

[2.1单词文法描述 2](#_Toc12201947)

[2.2语句文法描述 3](#_Toc12201948)

[2.3 符号表结构描述 7](#_Toc12201949)

[2.4 错误类型码描述 7](#_Toc12201950)

[2.5中间代码描述 7](#_Toc12201951)

[2.6 目标代码描述 8](#_Toc12201952)

[3系统设计与实现 12](#_Toc12201953)

[3.1编译程序符号表结构 12](#_Toc12201954)

[3.2编译程序报错功能 12](#_Toc12201955)

[3.3词法语法分析器（实验一） 13](#_Toc12201956)

[3.3.1词法分析器 13](#_Toc12201957)

[3.3.2语法分析器 14](#_Toc12201958)

[3.4语义分析（实验二） 18](#_Toc12201959)

[3.5中间代码生成功能（实验三） 19](#_Toc12201960)

[3.6汇编代码生成功能（实验四） 20](#_Toc12201961)

[4系统测试与评价 21](#_Toc12201962)

[4.1 测试用例 21](#_Toc12201963)

[4.2 正确性测试 23](#_Toc12201964)

[4.3 报错功能测试 26](#_Toc12201965)

[4.4 系统的优点 26](#_Toc12201966)

[4.5 系统的缺点 26](#_Toc12201967)

[5实验小结或体会 27](#_Toc12201968)

[**参考文献** 28](#_Toc12201969)

# 1选题背景

本次课程设计是构造一个高级语言的子集的编译器，目标代码可以是汇编语言也可以是其他形式的机器语言。按照任务书，实现的方案可以有很多种选择。

可以根据自己对编程语言的定义选择实现语言的特定功能。建议大家选用decaf语言。

编译器的语法和词法分析采用课程的课堂实验的结果，重点在语义分析、符号表结构设计、中间代码、目标代码存储结构设计、代码优化等阶段的实现。

课设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，提高自己对系统软件编写的兴趣。

# 2系统关键定义

## 2.1单词文法描述

mini C中的单词可以分为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定义如表2.1所示：

表2.1：单词的文法定义

|  |  |  |
| --- | --- | --- |
| 单词符号说明 | 单词种类码 | 正则表达式 |
| {char} | CHAR | '[A-Za-z0-9]' |
| {int} | INT | [0-9]+ |
| {float} | FLOAT | ([0-9]\*\.[0-9]+)|([0-9]+\.) |
| {string} | STRING | \"[A-Za-z0-9]\*\" |
| “char” | TYPE |  |
| “int” | TYPE |  |
| “float” | TYPE |  |
| "string" | TYPE |  |
| "struct" | TYPE |  |
| 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 |  |
| “&&” | AND |  |
| “||” | OR |  |
| "." | DOT |  |
| “!” | NOT |  |
| “(” | LP |  |
| “)” | RP |  |
| "[" | LB |  |
| "]" | RB |  |
| “{” | LC |  |
| “}” | RC |  |
| "//"[^\n]\* | 注释，无种类码 |  |
| "/\*"([^\\*]|(\\*)\*[^\\*/])\*(\\*)\*"\*/" | 注释，无种类码 |  |

## 2.2语句文法描述

对语句的文法定义如下代码所示：

|  |
| --- |
| program: ExtDefList {semantic\_Analysis0($1);} /\*显示语法树,语义分析: { display($1,0); semantic\_Analysis0($1);} \*/  ;  ExtDefList: {$$=NULL;}  | ExtDef ExtDefList {$$=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  ;  ExtDef: Specifier ExtDecList SEMI {$$=mknode(EXT\_VAR\_DEF,$1,$2,NULL,yylineno);} //该结点对应一个外部变量声明  | StructSpecifier SEMI {$$=mknode(EXT\_STRUCT\_DEF,$1,NULL,NULL,yylineno);}  | Specifier FuncDec CompSt {$$=mknode(FUNC\_DEF,$1,$2,$3,yylineno);} //该结点对应一个函数定义  | error SEMI {$$=NULL;}  ;  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);}  ;  Specifier: TYPE {$$=mknode(TYPE,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);if(!strcmp($1, "int"))$$->type=INT;if(!strcmp($1, "float"))$$->type=FLOAT;if(!strcmp($1, "char"))$$->type=CHAR;if(!strcmp($1, "string"))$$->type=STRING;}  | StructSpecifier {$$=$1;}  ;  StructSpecifier: STRUCT OptTag LC DefList RC {$$=mknode(STRUCT\_DEF,$2,$4,NULL,yylineno);}  | STRUCT OptTag {$$=mknode(STRUCT\_DEC,$2,NULL,NULL,yylineno);}  ;  OptTag: ID {$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}  ;  VarDec: ID {$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);} //ID结点，标识符符号串存放结点的type\_id  | VarDec LB INT RB {struct node \*temp=mknode(INT,NULL,NULL,NULL,yylineno);temp->type\_int=$3;$$=mknode(ARRAY\_DEC, $1, temp, NULL,yylineno);}  ;  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  ;  VarList: ParamDec {$$=mknode(PARAM\_LIST,$1,NULL,NULL,yylineno);}  | ParamDec COMMA VarList {$$=mknode(PARAM\_LIST,$1,$3,NULL,yylineno);}  ;  ParamDec: Specifier VarDec {$$=mknode(PARAM\_DEC,$1,$2,NULL,yylineno);}  ;  // 定义部分和执行部分  CompSt: LC DefList StmList RC {$$=mknode(COMP\_STM,$2,$3,NULL,yylineno);}  ;  StmList: {$$=NULL; }  | Stmt StmList {$$=mknode(STM\_LIST,$1,$2,NULL,yylineno);}  ;  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 ForDec RP Stmt {$$=mknode(FOR,$3,$5,NULL,yylineno);}  ;  ForDec: Exp SEMI Exp SEMI Exp {$$=mknode(FOR\_DEC,$1,$3,$5,yylineno);}  | SEMI Exp SEMI {$$=mknode(FOR\_DEC,NULL,$2,NULL,yylineno);}  ;  DefList: {$$=NULL; }  | Def DefList {$$=mknode(DEF\_LIST,$1,$2,NULL,yylineno);}  ;  Def: Specifier DecList SEMI {$$=mknode(VAR\_DEF,$1,$2,NULL,yylineno);}  ;  DecList: Dec {$$=mknode(DEC\_LIST,$1,NULL,NULL,yylineno);}  | Dec COMMA DecList {$$=mknode(DEC\_LIST,$1,$3,NULL,yylineno);}  ;  Dec: VarDec {$$=$1;}  | VarDec ASSIGNOP Exp {$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,"ASSIGNOP");}  ;  Exp: Exp ASSIGNOP Exp {$$=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 PLUS PLUS {$$=mknode(PLUS\_ONE,$1,NULL,NULL,yylineno);strcpy($$->type\_id,"PLUS\_ONE");}  | Exp PLUS ASSIGNOP Exp {$$=mknode(ASSIGNOP\_PLUS,$1,$4,NULL,yylineno);strcpy($$->type\_id,"ASSIGNOP\_PLUS");}  | Exp MINUS Exp {$$=mknode(MINUS,$1,$3,NULL,yylineno);strcpy($$->type\_id,"MINUS");}  | Exp MINUS MINUS {$$=mknode(MINUS\_ONE,$1,NULL,NULL,yylineno);strcpy($$->type\_id,"MINUS\_ONE");}  | Exp MINUS ASSIGNOP Exp {$$=mknode(ASSIGNOP\_MINUS,$1,$4,NULL,yylineno);strcpy($$->type\_id,"ASSIGNOP\_MINUS");}  | Exp STAR Exp {$$=mknode(STAR,$1,$3,NULL,yylineno);strcpy($$->type\_id,"STAR");}  | Exp STAR ASSIGNOP Exp {$$=mknode(ASSIGNOP\_STAR,$1,$4,NULL,yylineno);strcpy($$->type\_id,"ASSIGNOP\_STAR");}  | Exp DIV Exp {$$=mknode(DIV,$1,$3,NULL,yylineno);strcpy($$->type\_id,"DIV");}  | Exp DIV ASSIGNOP Exp {$$=mknode(ASSIGNOP\_DIV,$1,$4,NULL,yylineno);strcpy($$->type\_id,"ASSIGNOP\_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($$->type\_id,"NOT");}  | ID LP Args RP {$$=mknode(FUNC\_CALL,$3,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}  | ID LP RP {$$=mknode(FUNC\_CALL,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}  | Exp LB Exp RB {$$=mknode(EXP\_ARRAY,$1,$3,NULL,yylineno);}  | Exp DOT ID {struct node \*temp=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$3);$$=mknode(EXP\_ELE,$1,temp,NULL,yylineno);}  | ID {$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}  | INT {$$=mknode(INT,NULL,NULL,NULL,yylineno);$$->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;$$->type=CHAR;}  | STRING {$$=mknode(STRING,NULL,NULL,NULL,yylineno);strcpy($$->type\_string,$1);$$->type=STRING;}  ;  Args: Exp COMMA Args {$$=mknode(ARGS,$1,$3,NULL,yylineno);}  | Exp {$$=mknode(ARGS,$1,NULL,NULL,yylineno);}  ; |

## 2.3 符号表结构描述

这里采用的是单表组织形式来实现符号表，用一个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，则将其属性压栈到符号栈中，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、标记以及偏移量。别名在后续的实验步骤中将会用到。层号是用来记录符号所在的作用域的，每当程序进入一个复合语句时，层号就加一，退出时层号就减一。类型记录变量的数据类型或者是函数的返回值类型。偏移量记录变量的偏移地址。

## 2.4 错误类型码描述

词法分析由工具flex实现，该阶段的错误由flex自行处理。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）变量重复定义

（2）函数重复定义

（3）赋值语句错误

（4）函数引用错误

（5）函数参数不匹配

（6）返回类型错误

（7）函数未定义

## 2.5中间代码描述

选用四元式作为中间代码的形式，各种定义如表2.2所示：

表2.2：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 |

## 2.6 目标代码描述

选用MIPS作为对应的目标代码，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：

表2.3：中间代码与目标代码的对应关系

|  |  |
| --- | --- |
| 中间代码节点类型 | 目标代码形式 |
| LABEL | Label\_name: #直接打印标号名 |
| Return\_op | lw $t0,4($sp) #从栈中取出返回值  sw $t0,12($fp) #将返回值存到活动记录指定位置  add $t0,$ra,$zero #从ra寄存器取出本函数返回地址  lw $ra,8($fp) #恢复调用者函数的返回值  addi $sp,$fp,8 #恢复sp的值  lw $fp,4($fp) #恢复帧指针的值  jr $t0 返回 |
| GOTO\_BREAK | J end\_label\_num #跳到语句块结束标记处 |
| JUDGE | lw $t0,4($sp) #将布尔值从栈中取出到t0寄存器  addi $sp,$sp,4 #将布尔值出栈  beqz $t0 end\_label #布尔值为假跳转 |
| GOTO\_JUDGE | J judge\_lebel #跳转到语句块的判断标号处 |
| ASSIGN\_OP | lw $t0,8($sp) #从栈中取出要赋的值  addi $sp,$sp,8 #将临时值出栈  sw $t0, 变量偏移($fp) #将值存到要赋的变量 |
| AND\_NO | lw $t1,8($sp) #取出操作数1  lw $t2,4($sp) #取出操作数2  addi $sp,$sp,8 #将操作数出栈  and $t0,$t1,$t2 #计算与的结果  sw $t0,($sp) #结果压栈  addi $sp,$sp,-4 |
| OR\_NO | lw $t1,8($sp) #取出操作数1  lw $t2,4($sp) #取出操作数2  addi $sp,$sp,8 #将操作数出栈  or $t0,$t1,$t2 #计算或的结果  sw $t0,($sp) #结果压栈  addi $sp,$sp,-4 |
| GR\_NO | lw $t1,8($sp) #取出操作数1  lw $t2,4($sp) #取出操作数2  addi $sp,$sp,8 #将操作数出栈  addi $t3,$zero,0 #将t3置零  addi $t4,$zero,1 #将t4置1  sub $t0,$t1,$t2 #操作数1-操作数2的结果存t0  bgtz $t0, label\_x #结果大于0跳转到label\_x  sw $t3,($sp) #将0压栈，表示布尔值为false  j label\_y #跳到结束label处。  Label\_x:  sw $t4,($sp) #将1压栈，表示布尔值为真  label\_y:  addi $sp,$sp,-4 #栈向低地址空间移动 |
| LS\_NO | lw $t1,8($sp) #取出操作数1  lw $t2,4($sp) #取出操作数2  addi $sp,$sp,8 #将操作数出栈  addi $t3,$zero,0 #将t3置零  addi $t4,$zero,1 #将t4置1  sub $t0,$t1,$t2 #操作数1-操作数2的结果存t0  bltz $t0, label\_x #结果小于0跳转到label\_x  sw $t3,($sp) #将0压栈，表示布尔值为false  j label\_y #跳到结束label处。  Label\_x:  sw $t4,($sp) #将1压栈，表示布尔值为真  label\_y:  addi $sp,$sp,-4 #栈向低地址空间移动 |
| GE | lw $t1,8($sp) #取出操作数1  lw $t2,4($sp) #取出操作数2  addi $sp,$sp,8 #将操作数出栈  addi $t3,$zero,0 #将t3置零  addi $t4,$zero,1 #将t4置1  sub $t0,$t1,$t2 #操作数1-操作数2的结果存t0  bgez $t0, label\_x #结果大于等于0跳转到label\_x  sw $t3,($sp) #将0压栈，表示布尔值为false  j label\_y #跳到结束label处。  Label\_x:  sw $t4,($sp) #将1压栈，表示布尔值为真  label\_y:  addi $sp,$sp,-4 #栈向低地址空间移动 |
| LE | lw $t1,8($sp) #取出操作数1  lw $t2,4($sp) #取出操作数2  addi $sp,$sp,8 #将操作数出栈  addi $t3,$zero,0 #将t3置零  addi $t4,$zero,1 #将t4置1  sub $t0,$t1,$t2 #操作数1-操作数2的结果存t0  blez $t0, label\_x #结果大于0跳转到label\_x  sw $t3,($sp) #将0压栈，表示布尔值为false  j label\_y #跳到结束label处。  Label\_x:  sw $t4,($sp) #将1压栈，表示布尔值为真  label\_y:  addi $sp,$sp,-4 #栈向低地址空间移动 |
| EQ\_NO | lw $t1,8($sp) #取出操作数1  lw $t2,4($sp) #取出操作数2  addi $sp,$sp,8 #将操作数出栈  addi $t3,$zero,0 #将t3置零  addi $t4,$zero,1 #将t4置1  beq $t1, $t2,label\_x #结果大于0跳转到label\_x  sw $t3,($sp) #将0压栈，表示布尔值为false  j label\_y #跳到结束label处。  Label\_x:  sw $t4,($sp) #将1压栈，表示布尔值为真  label\_y:  addi $sp,$sp,-4 #栈向低地址空间移动 |
| NE\_NO | lw $t1,8($sp) #取出操作数1  lw $t2,4($sp) #取出操作数2  addi $sp,$sp,8 #将操作数出栈  addi $t3,$zero,0 #将t3置零  addi $t4,$zero,1 #将t4置1  bne $t1, $t2,label\_x #结果大于0跳转到label\_x  sw $t3,($sp) #将0压栈，表示布尔值为false  j label\_y #跳到结束label处。  Label\_x:  sw $t4,($sp) #将1压栈，表示布尔值为真  label\_y:  addi $sp,$sp,-4 #栈向低地址空间移动 |
| ADD | lw $t1,8($sp) #取操作数1  lw $t2,4($sp) #取操作数2  addi $sp,$sp,8 #操作数出栈  add $t0,$t1,$t2 #操作数相加存到t0中  sw $t0,($sp) #计算结果压栈  addi $sp,$sp,-4 |
| MINUS | lw $t1,8($sp) #取操作数1  lw $t2,4($sp) #取操作数2  addi $sp,$sp,8 #操作数出栈  sub $t0,$t1,$t2 #操作数相减存到t0中  sw $t0,($sp) #计算结果压栈  addi $sp,$sp,-4 |
| MUL | lw $t1,8($sp) #取操作数1  lw $t2,4($sp) #取操作数2  addi $sp,$sp,8 #操作数出栈  mul $t0,$t1,$t2 #操作数相乘存到t0中  sw $t0,($sp) #计算结果压栈  addi $sp,$sp,-4 |
| DIV | lw $t1,8($sp) #取操作数1  lw $t2,4($sp) #取操作数2  addi $sp,$sp,8 #操作数出栈  div $t1,$t2 #操作数相除  mflo $t0 #商存到t0  sw $t0,($sp) #计算结果压栈  addi $sp,$sp,-4 |
| UMINUS\_NO | lw $t1,4($sp) #取操作数  addi $sp,$sp,4 #操作数出栈  sub $t0,$zero,$t1 #计算负值  sw $t0,($sp) #结果如栈  addi $sp,$sp,-4 |
| NOT | lw $t1,4($sp) #取操作数  addi $sp,$sp,4 #操作数出栈  beq $t1,$zero,label\_x  sw $zero,($sp) #操作数是1，取非后将0入栈  j label-y  label\_x:  addi $t1,$zero,1  sw $t1,($sp) #操作数是0，将取非后的1入栈  label\_y:  addi $sp,$sp,-4 |
| INT\_NO | addi $t0,$zero,num #立即数存到t0  sw $t0,($sp) $t0压栈  addi $sp,$sp,-4 |
| ID\_NO | lw $t0,-变量偏移($fp) #取出变量的值  sw $t0,($sp) #变量值压栈  addi $sp,$sp,-4 |
| FUN\_PUBLIC | addi $sp,$sp,-12 #预留控制区空间  sw $ra,8($sp) #保存当前函数返回地址  sw $fp,4($sp) #保存当前帧地址 |
| FUN\_CALL | addi $fp,$sp,实参所占空间大小 #调整帧指针  addi $sp,$fp,-活动记录大小 #调整栈指针  jal x #跳转到函数x |

# 3系统设计与实现

## 3.1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：

|  |
| --- |
| struct symbol  {  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'  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  //或函数活动记录大小，目标代码生成时使用  //其它...  }; |

其中各个字段的含义如注释所示。

整个符号表的定义如下所示：

|  |
| --- |
| 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。

## 3.2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。

## 3.3词法语法分析器（实验一）

### 3.3.1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。

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;}  "string" {strcpy(yylval.type\_id, yytext);return TYPE;}  "struct" {return STRUCT;}  "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 ('[A-Za-z0-9]')  string (\"[A-Za-z0-9]\*\")  {int} {yylval.type\_int=atoi(yytext);return INT;}  {float} {yylval.type\_float=atof(yytext);return FLOAT;}  {char} {strcpy(yylval.type\_char,yytext);return CHAR;}  {string} {strcpy(yylval.type\_string,yytext);return STRING;} |

（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 AND;}  "||" {return OR;} |

（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

|  |
| --- |
| "//"[^\n]\* {/\* 注释 \*/}  "/\*"([^\\*]|(\\*)\*[^\\*/])\*(\\*)\*"\*/" {/\* 注释 \*/} |

### 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);} /\*显示语法树,语义分析: { display($1,0); semantic\_Analysis0($1);} \*/  ;  ExtDefList: {$$=NULL;}  | ExtDef ExtDefList {$$=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  ;  ExtDef: Specifier ExtDecList SEMI {$$=mknode(EXT\_VAR\_DEF,$1,$2,NULL,yylineno);} //该结点对应一个外部变量声明  | Specifier SEMI  | Specifier FuncDec CompSt {$$=mknode(FUNC\_DEF,$1,$2,$3,yylineno);} //该结点对应一个函数定义  | error SEMI {$$=NULL;}  ;  ExtDecList: VarDec {$$=$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 {$$=mknode(EXT\_DEC\_LIST,$1,$3,NULL,yylineno);}  ;  Specifier: TYPE {$$=mknode(TYPE,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);if($1=="int")$$->type=INT;if($1=="float")$$->type=FLOAT;if($1=="char")$$->type=CHAR;if($1=="string")$$->type=STRING;}  | StructSpecifier {}  ;  StructSpecifier: STRUCT OptTag LC DefList RC {$$=mknode(STRUCT\_DEF,$2,$4,NULL,yylineno);}  | STRUCT Tag {$$=mknode(STRUCT\_DEC,$2,NULL,NULL,yylineno);}  ;  OptTag: {$$=NULL;}  | ID {$$=mknode(STRUCT\_TAG,NULL,NULL,NULL,yylineno);strcpy($$->struct\_name,$1);}  ;  Tag: ID {$$=mknode(STRUCT\_TAG,NULL,NULL,NULL,yylineno);strcpy($$->struct\_name,$1);}  ;  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  | VarDec LB INT RB {$$=mknode(ARRAY\_DEC,$1,$3,NULL,yylineno);}  ;  FuncDec: ID LP VarList RP {$$=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$$->type\_id  ;  VarList: ParamDec {$$=mknode(PARAM\_LIST,$1,NULL,NULL,yylineno);}  | ParamDec COMMA VarList {$$=mknode(PARAM\_LIST,$1,$3,NULL,yylineno);}  ;  ParamDec: Specifier VarDec {$$=mknode(PARAM\_DEC,$1,$2,NULL,yylineno);}  ;  // 定义部分和执行部分  CompSt: LC DefList StmList RC {$$=mknode(COMP\_STM,$2,$3,NULL,yylineno);}  ;  StmList: {$$=NULL; }  | Stmt StmList {$$=mknode(STM\_LIST,$1,$2,NULL,yylineno);}  ;  Stmt: Exp SEMI {$$=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 {$$=mknode(WHILE,$3,$5,NULL,yylineno);}  | FOR LP ForDec RP Stmt {$$=mknode(FOR,$3,$5,NULL,yylineno);}  ;  ForDec: Exp SEMI Exp SEMI Exp {$$=mknode(FOR\_DEC,$1,$3,$5,yylineno);}  | SEMI Exp SEMI {$$=mknode(FOR\_DEC,NULL,$2,NULL,yylineno);}  ;  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(DEC\_LIST,$1,NULL,NULL,yylineno);}  | Dec COMMA DecList {$$=mknode(DEC\_LIST,$1,$3,NULL,yylineno);}  ;  Dec: VarDec {$$=$1;}  | VarDec ASSIGNOP Exp {$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type\_id,"ASSIGNOP");}  ;  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($$->type\_id,"PLUS");}  | Exp PLUS PLUS {$$=mknode(PLUS\_ONE,$1,NULL,NULL,yylineno);strcpy($$->type\_id,"PLUS\_ONE");}  | Exp PLUS ASSIGNOP Exp {$$=mknode(ASSIGNOP\_PLUS,$1,$4,NULL,yylineno);strcpy($$->type\_id,"ASSIGNOP\_PLUS");}  | Exp MINUS Exp {$$=mknode(MINUS,$1,$3,NULL,yylineno);strcpy($$->type\_id,"MINUS");}  | Exp MINUS MINUS {$$=mknode(MINUS\_ONE,$1,NULL,NULL,yylineno);strcpy($$->type\_id,"MINUS\_ONE");}  | Exp MINUS ASSIGNOP Exp {$$=mknode(ASSIGNOP\_MINUS,$1,$4,NULL,yylineno);strcpy($$->type\_id,"ASSIGNOP\_MINUS");}  | Exp STAR Exp {$$=mknode(STAR,$1,$3,NULL,yylineno);strcpy($$->type\_id,"STAR");}  | Exp STAR ASSIGNOP Exp {$$=mknode(ASSIGNOP\_STAR,$1,$4,NULL,yylineno);strcpy($$->type\_id,"ASSIGNOP\_STAR");}  | Exp DIV Exp {$$=mknode(DIV,$1,$3,NULL,yylineno);strcpy($$->type\_id,"DIV");}  | Exp DIV ASSIGNOP Exp {$$=mknode(ASSIGNOP\_DIV,$1,$4,NULL,yylineno);strcpy($$->type\_id,"ASSIGNOP\_DIV");}  | LP Exp RP {$$=$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");}  | 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);}  | Exp LB Exp RB {$$=mknode(EXP\_ARRAY,$1,$3,NULL,yylineno);}  | Exp DOT ID {$$=mknode(EXP\_ELE,$1,$3,NULL,yylineno);}  | ID {$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type\_id,$1);}  | INT {$$=mknode(INT,NULL,NULL,NULL,yylineno);$$->type\_int=$1;$$->type=INT;}  | FLOAT {$$=mknode(FLOAT,NULL,NULL,NULL,yylineno);$$->type\_float=$1;$$->type=FLOAT;}  | CHAR {$$=mknode(CHAR,NULL,NULL,NULL,yylineno);strcpy(yylval.type\_char,$1);$$->type=CHAR;}  | STRING {$$=mknode(STRING,NULL,NULL,NULL,yylineno);strcpy(yylval.type\_string,$1);$$->type=STRING;}  ;  Args: Exp COMMA Args {$$=mknode(ARGS,$1,$3,NULL,yylineno);}  | Exp {$$=mknode(ARGS,$1,NULL,NULL,yylineno);}  ; |

## 3.4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。

## 3.5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。

.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，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）。

## 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。具体内容参见前面2.6节。

# 4系统测试与评价

## 4.1 测试用例

实验一测试代码如下所示：

|  |
| --- |
| int a, b, c;  float m, n;  char fuck;  struct TS  {  int t;  float s;  } testTs2;  // 斐波拉契  int fibo(int a)  {  if (a == 1 || a == 2)  return 1;  return fibo(a - 1) + fibo(a - 2);  }  /\* 主函数 \*/  int main()  {  int m, n, i;  char hello;  int ar[2][3];  struct TS testTs1;  string hello2;  m = read();  hello = 'a';  hello2 = "dasfa";  ar[0][1] = 1;  i = 1;  testTs1.t = 1;  while (i <= m)  {  n = fibo(i);  write(n);  i++;  }  for (i = 1; i <= m; i = i + 1)  {  i = i + 1;  }  i += 1;  return 1;  } |

实验二测试代码

|  |
| --- |
| int a,b,c;  float m,n;  char xxxx;  // struct TS  // {  // int t;  // float s;  // } testTs2;  int fibo(int a)  {  if (a == 1 || a == 2) return (a-1);  return fibo(a-1)+fibo(a-2);  }  int main()  {  int m,n,i;  // int c[10];  char ch;  float f;  string str;  m = read();  i=1;  //错误  jojo = 12;///未定义变量  3 = m;  m = add(3,4);  //错误  while(i<=m)  {  n = fibo(i);  write(n);  i=i+1;  }  return 1;  } |

实验三测试代码

|  |
| --- |
| 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)  {  n = fibo(i);  write(n);  i=i+1;  }  return 1;  } |

## 4.2 正确性测试

实验一的测试结果如图4.1所示：

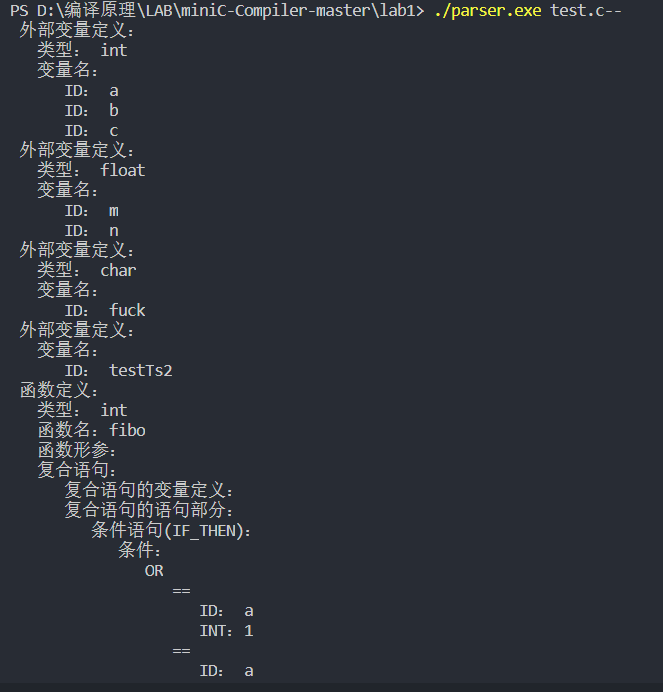


图4.1：实验一测试结果

实验二的测试结果如图4.2所示：

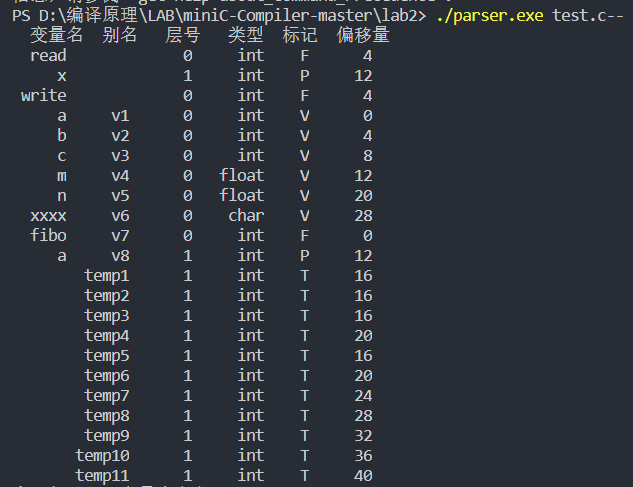


图4.2：实验二测试结果

实验三测试结果如图4.3所示：

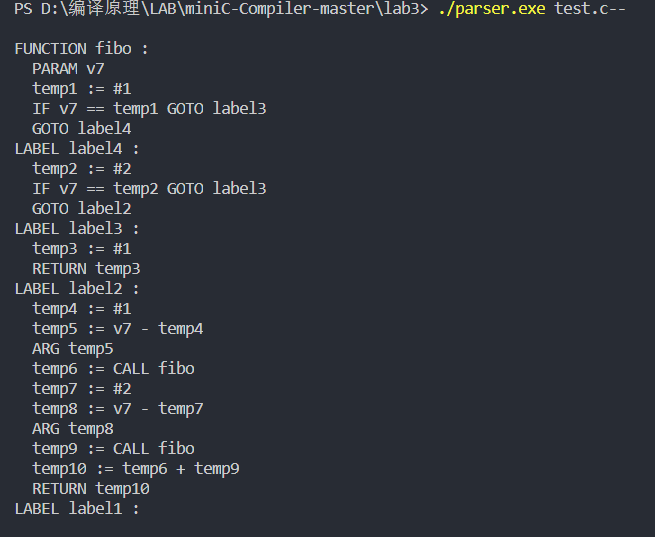


图4.3：实验三测试结果

## 4.3 报错功能测试

实验二的测试代码中添加了几个错误，在测试代码中已经用注释给出。错误的报错情况如图4.4所示：



图4.4：报错

## 4.4 系统的优点

系统在miniC的基础上，还增加了单行注释、多行注意、char类型、string类型等多种功能。能够检测多种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。

## 4.5 系统的缺点

没有实现数组、结构体、联合等高级的数据结构和操作。

# 5实验小结或体会

通过这次实验，了解了编译程序整个的流程，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的整个流程。加深对理论课上所讲的概念的理解，同时提高了自己的动手能力。

**参考文献**

[1] 吕映芝等. 编译原理(第二版). 北京：清华大学出版社，2005

[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