南開大學
Nankai University

计算机学院

编译系统原理期末实验报告

简易 SysY 语言编译器的实现

小组：B440

小组成员：刘荟文 孙蕗

学号：2114019 2112060

专业：计算机科学与技术 信息安全

报告作者：孙蕗

指导教师：王刚老师

2024 年 1 月 16 日

目录

1 摘要

此次编译器的实现包含了词法分析、语法分析、类型检查及中间代码生成和目标代码生成 5 个模块。完成了实验的基本要求，还实现了数组、break、continue 控制流等进阶要求。本实验两个同学相互合作，共同完成本学期的编译原理实验。在实验的过程中进一步深化对理论的认识，也因为理论知识不足导致后续实验完成进展缓慢，进行较多错误的尝试，实现也变得十分繁复。经过一个学期地不懈努力最终通过 146 个测试样例。

关键字：词法分析 语法分析 类型检查 中间代码生成 目标代码生成

2 分工

- 词法分析阶段

  经由交流讨论，小组成员完成了对关键字、标识符、基本符号、多进制下整数和浮点数、库函数的识别。

- 语法分析阶段

  经由交流讨论，小组成员完成了上下无关文法的设计和语法树的构建工作，包含对标识符定义和初始化，基本算数表达式，数组，分支循环，函数定义及调用，作用域等的设计。

- 类型检查（语义分析）

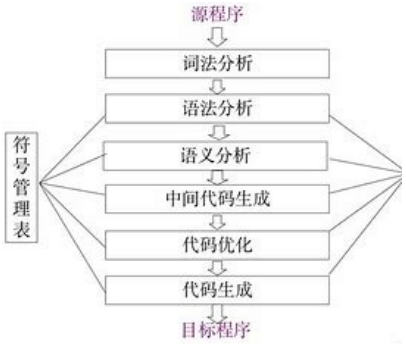
  经由交流讨论，小组成员完成了表达式类型检查，函数返回值及类型检查、常量计算和变量初始化等工作。

- 中间代码生成

  经由交流讨论，小组成员完成了 IR 指令的生成，为浮点数和数组提供了相关指令的设计。

- 目标代码生成

  经由交流讨论，小组成员完成了汇编代码的生成，包含与数组、浮点数相关的汇编代码生成。



得分99.00 最后一次提交时间:2024-01-15 12:49:28

Runtime Error

Summary		
RE	Score(Functional Test)	99
	Time(Performance Test)	-
Git Clone Command		
Last Commit At	-	

Detail

3 词法分析

3.1 实验内容

词法分析阶段需要借助 Flex，输入 SysY 语言源程序，输出每一个文法单元的类别、词素、行号、列号、以及必要的属性（DECIMAL 的属性为数值，ID 的属性为符号表的指针）。比如 DECIMAL 有对应属于它的“数值”属性，ID 有它对应的符号表项。完成关键字、标识符、符号、库函数等的识别，采用正则表达式，对相关进制的整数和浮点数进行识别。

3.2 实验效果

图 3.1: example.sy 截图

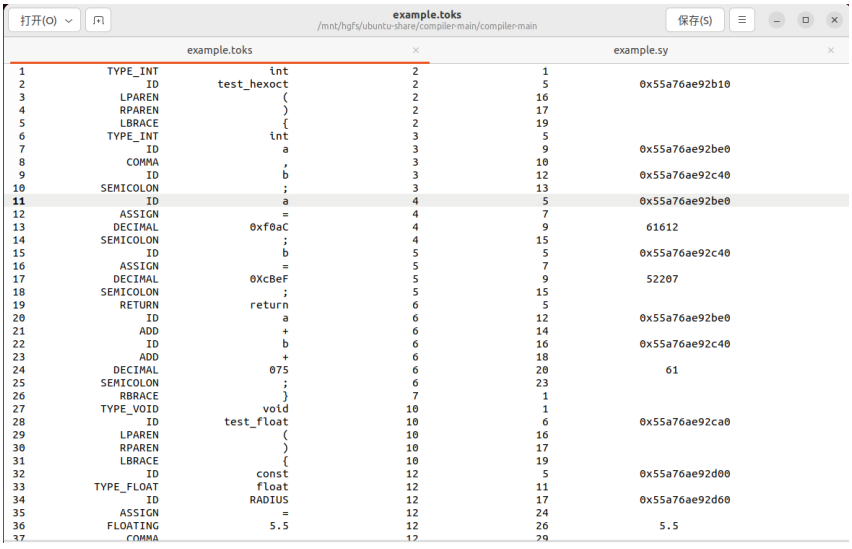


图 3.2: example.sy 词法分析后 tokens 截图

3.3 代码设计

主要需要设计规则、定义、以及符号表部分

3.3.1 定义部分

文字块内可以直接书写 C 代码，这里文字块内 C 代码根据文法单元的类别进行不同的输出。

```
%{
    /* Your code here, if desired (lab3). */
    int offset = 1;

    void DEBUG_FOR_LAB3(std::string token, std::string lexeme){
        fprintf(yyout, "%20s%20s%20d%20d\n",token.c_str(), lexeme.c_str(), yylineno, offset);
    }

    void DEBUG_FOR_LAB3_DECIMAL(std::string token, std::string lexeme, std::string value){
        fprintf(yyout, "%20s%20s%20d%20d%20s\n",token.c_str(), lexeme.c_str(), yylineno, offset, value.c_str());
    }

    void DEBUG_FOR_LAB3_ID(std::string token, std::string lexeme, double* p){
        fprintf(yyout, "%20s%20s%20d%20d\t%20p\n",token.c_str(), lexeme.c_str(), yylineno, offset, p);
    }

%}
```

下面完成了对十进制、八进制、十六进制、浮点型常量、十六进制浮点数等的规则进行定义. 对浮点数进行定义时还考虑到了指数的情况。比如合法的指数部分可以是“E9”，“E+9”，“E-9”，“e9”等等情况。除此之外，如果有指数部分，小数部分还可以是整型而不是浮点型。比如“1e9”这种情况。

定义部分声明一些起始状态用来限制特定规则的作用范围，比如处理注释段。注释段中的数字字母等元素不应该作为正常的元素识别。此处完成了单行注释和多行注释的词法分析。%x 定义了注释状态，表示进入此状态后，只有此状态下的模式可以匹配。INITIAL 为默认状态。%% 分隔开每一个部分。词法分析器接受源程序作为输入流，当词法分析器匹配到定义的模式后，返回相应的词法单元（TOKEN）给语法分析器（将源程序拆解成词法单元返回给语法分析器）。

```
INTEGER ([1-9][0-9]*|0)
OCTAL (0[1-7][0-7]*|0)
HEXAL (0(x|X)[1-9a-fA-F][0-9a-fA-F]*|0)
FLOATING_DEC ((([0-9]*[.][0-9]*([eE][+-]?[0-9]+)?)|([0-9]+[eE][+-]?[0-9]+))[fLlL]?)
FLOATING_HEX (((0(x|X)[0-9a-fA-F]*[.][0-9a-fA-F]*([eE][+-]?[0-9a-fA-F]+)?)|([0-9a-fA-F]+[eE][+-]?[0-9a-fA-F]+))[fLlL]?)
ID [[:alpha:]_][[:alpha:][:digit:]_]*
EOL (\r\n|\n|\r)
WHITE [\t ]
COMMENT (\/\/[^\n]*)
commentbegin "/*"
commentelement .
commentline \n
commentend "*/"
%x COMMENT
%%
{commentbegin} {BEGIN COMMENT;}
<COMMENT>{commentelement} {}
<COMMENT>{commentline} {yylineno++;}
<COMMENT>{commentend} {BEGIN INITIAL;}
```

3.3.2 规则部分

规则部分包含模式行和 C 代码，支持 [ ]、*、+、?、|、( )、{ }、/等元字符，还可以指定匹配某个字符之外的字符、重复某个规则的若干次。如符号 [...] 表示方括号内包含的项可被重复 0 次或 1 次；符号 {...} 表示花括号内包含的项可被重复 0 次或多次.

```
135 "-" {
136         DEBUG_FOR_LAB3("SUB", "-");
137         offset+=strlen(yytext);
138 }
139 "*" {
140         DEBUG_FOR_LAB3("MUL", "*");
141         offset+=strlen(yytext);
142 }
143 "/" {
144         DEBUG_FOR_LAB3("DIV", "/");
145         offset+=strlen(yytext);
146 }
147 "%" {
148         DEBUG_FOR_LAB3("MOD", "%");
149         offset+=strlen(yytext);
150 }
151 "||" {
152         DEBUG_FOR_LAB3("OR", "||");
153         offset+=strlen(yytext);
154 }
155 "&&" {
156         DEBUG_FOR_LAB3("AND", "&&");
157         offset+=strlen(yytext);
158 }
159 "!" {
160         DEBUG_FOR_LAB3("NOT", "!");
161         offset+=strlen(yytext);
162 }
163 "," {
164         DEBUG_FOR_LAB3("COMMA", ",");
165         offset+=strlen(yytext);
```

在上面完成了对十进制、八进制、十六进制、浮点型常量、十六进制浮点数等的规则进行定义后，

下面还实现了将八进制和十六进制数保存为十进制的输出，将浮点数常量保存为 float 类型的输出。


```
{INTEGER} {
        DEBUG_FOR_LAB3_DECIMAL("INTEGER", ""+std::string(yytext), ""+std::string(yytext));
        offset+=strlen(yytext);
}
{FLOATING_DEC} {
        DEBUG_FOR_LAB3_DECIMAL("FLOATING", ""+std::string(yytext), ""+std::string(yytext));
        offset+=strlen(yytext);
}
{FLOATING_HEX} {
        //char* stop;
        //float dec = strtol(yytext, &stop, 16);
        //DEBUG_FOR_LAB3_DECIMAL("DECIMAL", ""+std::string(yytext), ""+to_string(dec));
        DEBUG_FOR_LAB3_DECIMAL("FLOATING", ""+std::string(yytext), ""+std::string(yytext));
        offset+=strlen(yytext);
}
{ID} {
        string str = yytext;
        DEBUG_FOR_LAB3_ID("ID",""+std::string(yytext), mysymTable.getAddress(str,1));
        offset+=strlen(yytext);
}
{OCTAL} {
        char* stop;
        int dec = strtol(yytext, &stop, 8);
        DEBUG_FOR_LAB3_DECIMAL("DECIMAL", ""+std::string(yytext), ""+to_string(dec));
        offset+=strlen(yytext);
}
{HEXAL} {
        char* stop;
        int dec = strtol(yytext, &stop, 16);
        DEBUG_FOR_LAB3_DECIMAL("DECIMAL", ""+std::string(yytext), ""+to_string(dec));
        offset+=strlen(yytext);
}
{EOL} yylineno++,offset=1;
{WHITE} {
    if(yytext[0] == 32)
        offset+=1;
    else
```

3.3.3 符号表

本次实验中实现了简单的符号表。对于标识符（ID），它的属性是符号表项，同名标识符在相同作用域可能指向相同的符号表项，也可能因为在不同作用域重新声明而指向不同的符号表项，词法分析程序需要对这些情况进行区分。此次实验的符号表可实现简单的插入、查找、获取符号表项指针的功能。


```
#ifndef MYSYMBOLTABLE_H
#define MYSYMBOLTABLE_H

#include <string>
#include <unordered_map>

class MySymbolTable {
    std::unordered_map<std::string, double*> mysymbolTable;
public:
    bool lookup(std::string identifier);// 查找标识符是否存在于符号表中
    void insert(std::string identifier, double value);// 插入或更新标识符及其对应的值到符号表中
    double* getAddress(std::string identifier, double value);
};
#endif //SYMBOLTABLE_H
```

```
#include <iostream>
#include <cfloat>
#include "MySymbolTable.h"

bool MySymbolTable::lookup(std::string identifier)
{
    if(mysymbolTable.find(identifier) == mysymbolTable.end()){// 使用 find() 函数在符号表中查找标识符
        return false;//不存在返回false
    }
    return true;//存在返回true
}

void MySymbolTable::insert(std::string identifier, double value)
{
    double* addr = new double(value); // 创建一个新的 double 类型的变量，并将其地址存储在符号表中
    auto res = mysymbolTable.insert(std::pair<std::string, double*>(identifier, addr));
    if(res.second){}
}

double* MySymbolTable::getAddress(std::string identifier,double value) {
    if (mysymbolTable.find(identifier) == mysymbolTable.end()) {
        insert(identifier, value);
        getAddress(identifier, value);
    }
    return mysymbolTable[identifier]; // 返回标识符对应的地址
}
```

4 语法分析

4.1 实验内容

基于词法分析的 token 序列构建一颗抽象语法树，每一行可以理解为语法树上的一个结点，每个结点有其自身的类型、属性、以及数个子节点。构建出树后，之后的操作可以理解为对该树进行一次遍历。根据作用域的层级关系将识别到的标识符加入对应层级的符号表，对于表达式，根据子表达式类型初步确定父表达式的类型，完成变量的初始化赋值操作和数组维度的表示工作。

4.2 实验效果

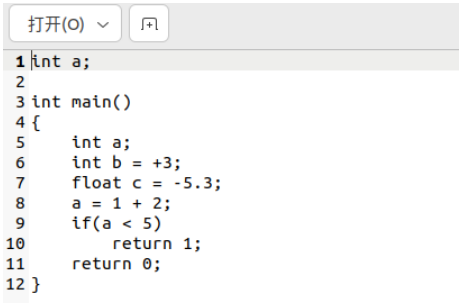


图 4.3: example.sy 截图

图 4.4: example.sy 语法分析后 ast 截图

4.3 代码设计

针对 SysY 语法特性设计对应的上下文无关文法。SysY 语言的语法是递归的，可以递归解析它。扫描并检查第一个令牌是一个数字。如果不是，则返回错误。获取下一个单词。如果到达输入的末尾，则返回结果。否则进行递归解析。

4.3.1 if-else 配对

为解决悬空-else 文法的二义性问题，使用 Bison 中的优先级功能，通过手动设置 then 语句和 else 语句的优先级，来实现让 else 和最近的 if 配对。

设置 then 和 else 语句的优先级 (else 的优先级高于 then)，然后在 if 的上下文无关文法中，使用下面两个符号。产生式优先级和右部最后一个终结符的优先级相同，即产生式 stmt->if expr then stmt 的优先级和终结符 then 的优先级相同。发生移进/规约冲突时，可以通过比较向前看符号和产生式的优先级解决冲突。若向前看符号的优先级更高，则进行移入，若产生式的优先级更高，则进行规约。else 优先级更高，因此会将 else 移入。

```
17
18 %union {
19     int itype;
20     double ftype;
21     char* strtype;
22     StmtNode* stmttype;
23     ExprNode* exprtype;
24     Type* type;
25 }
26
27 %start Program
28 %token <strtype> ID
29 %token <itype> INTEGER
30 %token <ftype> FLOATING
31 %token CONST
32 %token TYPE_INT TYPE_FLOAT TYPE_VOID
33 %token IF ELSE WHILE BREAK CONTINUE RETURN
34 %token LPAREN RPAREN LBRACKET RBRACKET LBRACE RBRACE COMMA SEMICOLON
35 %token ADD SUB MUL DIV MOD AND OR NOT LESS LESSEQ GREAT GREATEQ EQ NEQ ASSIGN
36
37 %type <stmttype> Stmts Stmt AssignStmt ExpStmt BlockStmt IfStmt WhileStmt BreakStmt ContinueStmt ReturnStmt
38 %type <stmttype> DeclStmt ConstDefList ConstDef ConstInitVal VarDefList VarDef VarInitVal FuncDef FuncParams FuncParam FuncRParams
39 %type <stmttype> ArrConstIndices ArrValIndices ConstInitValList VarInitValList
40 %type <exprtype> Exp ConstExp AddExp MulExp UnaryExp PrimaryExp LVal Cond LOrExp LAndExp EqExp RelExp
41 %type <type> Type
42
43 %precedence THEN
44 %precedence ELSE
45 %%
```

```
150 // if语句，带和不带else的情况
151 IfStmt
152     :   IF LPAREN Cond RPAREN Stmt %prec THEN {
153             $$ = new IfStmt($3, $5);
154         }
155     |   IF LPAREN Cond RPAREN Stmt ELSE Stmt {
156             $$ = new IfElseStmt($3, $5, $7);
157         }
158     ;
159
```

4.3.2 语句

语句包含赋值语句、表达式语句、空语句、语句块、if 语句、while 语句、break 语句、continue 语句、return 语句等。

大部分语句很容易理解，需要注意的是赋值语句需要包含左值和右值，这需要在 AssignStmt 中分别存储，其中左值需要用单独的容器存储，其中包括变量名称和数组维度信息。

Listing 1: 语句

```
1  // 程序
2  Program
3      :    Stmts{
4               ast.setRoot($1);
5           }
6      ;
7
8  // 语句序列
9  Stmts
10     :    Stmts Stmt{
11              SeqNode* node = (SeqNode*)$1;
12              node->addNext((StmtNode*)$2);
```

```
13              $$ = (StmtNode*) node;
14          }
15      |   Stmt{
16              SeqNode* node = new SeqNode();
17              node->addNext((StmtNode*)$1);
18              $$ = (StmtNode*) node;
19          }
20      ;
21
22  // 语句
23  Stmt
24      :   AssignStmt {$$=$1;} //赋值语句
25      |   ExpStmt SEMICOLON{$$=$1;}   //表达式语句
26      |   BlockStmt {$$=$1;}  //语句块
27      |   IfStmt {$$=$1;} //if语句
28      |   WhileStmt {$$=$1;}  //while语句
29      |   BreakStmt {$$=$1;}  //break语句
30      |   ContinueStmt {$$=$1;}   //continue语句
31      |   ReturnStmt {$$=$1;} //return语句
32      |   DeclStmt {$$=$1;}   //
33      |   FuncDef {$$=$1;}    //
34      |   SEMICOLON {$$ = new EmptyStmt();}   //
35      ;
36
37  // 赋值操作的左值
38  LVal
39      :   ID {    //标识符
40              SymbolEntry *se;
41              se = identifiers->lookup($1);
42              if(se == nullptr)
43              {
44                  fprintf(stderr, "identifier \"%s\" is undefined\n", (char*)$1);
45                  delete [](char*)$1;
46                  assert(se != nullptr);
47              }
48              $$ = new Id(se);
49              delete []$1;
50          }
51      |   ID ArrValIndices {  //数组元素
52              SymbolEntry *se;
53              se = identifiers->lookup($1);
54              if(se == nullptr)
55              {
56                  fprintf(stderr, "identifier \"%s\" is undefined\n", (char*)$1);
57                  delete [](char*)$1;
58                  assert(se != nullptr);
59              }
60              Id* newId = new Id(se);
61              newId->addIndices((ExprStmtNode*)$2);
62              $$ =  newId;
```

```
63              delete []$1;
64          }
65      ;
66
67  // 赋值语句
68  AssignStmt
69      :   LVal ASSIGN Exp SEMICOLON {
70              $$ = new AssignStmt($1, $3);
71          }
72      ;
73
74  // 表达式语句
75  ExpStmt
76      :   ExpStmt COMMA Exp { //多个表达式
77              ExprStmtNode* node = (ExprStmtNode*)$1;
78              node->addNext($3);
79              $$ = node;
80          }
81      |   Exp {   //单个表达式
82              ExprStmtNode* node = new ExprStmtNode();
83              node->addNext($1);
84              $$ = node;
85          }
86      ;
87
88  // 语句块, 由{}包围
89  BlockStmt
90      :   LBRACE {
91              identifiers = new SymbolTable(identifiers);
92          }
93          Stmts RBRACE {
94              $$ = new CompoundStmt($3);
95              SymbolTable *top = identifiers;
96              identifiers = identifiers->getPrev();
97              delete top;
98          }
99      |   LBRACE RBRACE {
100             $$ = new CompoundStmt(nullptr);
101         }
102     ;
103
104 // if语句, 带和不带else的情况
105 IfStmt
106     :   IF LPAREN Cond RPAREN Stmt %prec THEN {
107             $$ = new IfStmt($3, $5);
108         }
109     |   IF LPAREN Cond RPAREN Stmt ELSE Stmt {
110             $$ = new IfElseStmt($3, $5, $7);
111         }
112     ;
```

```
113
114 //while 语句
115 WhileStmt
116     :   WHILE LPAREN Cond RPAREN Stmt {
117             $$ = new WhileStmt($3,$5);
118         }
119     ;
120
121 //break 语句
122 BreakStmt
123     :   BREAK SEMICOLON {
124             $$ = new BreakStmt();
125         }
126     ;
127
128 //continue 语句
129 ContinueStmt
130     :   CONTINUE SEMICOLON{
131             $$ = new ContinueStmt();
132         }
133     ;
134
135 // return 语句, 带返回值和不带返回值
136 ReturnStmt
137     :   RETURN Exp SEMICOLON {
138             $$ = new ReturnStmt($2);
139         }
140     |   RETURN SEMICOLON {
141             $$ = new ReturnStmt(nullptr);
142         }
143     ;
```

4.3.3 算术运算结点

算术运算包含一元运算和二元运算。在语法分析阶段，先对结点类型进行判断，选择运算节点中类型优先级高的节点。比如两个 int 类型的结点运算结果才能是 int 类型，有一个是 float 类型的运算结果就应该是 float 类型。

Listing 2: 算术运算

```
1 // 变量表达式
2 Exp
3     :   AddExp {
4             $$ = $1;
5         }
6     ;
7
8 // 常量表达式
9 ConstExp
```

```
    :   AddExp {
            $$ = $1;
        }
    ;

// 加法级表达式，加法，减法
AddExp
    :   MulExp {
            $$ = $1;
        }
    |   AddExp ADD MulExp {
            SymbolEntry *se;
            if($1->getType()->isInt() && $3->getType()->isInt()){
                se = new TemporarySymbolEntry(TypeSystem::intType,
                    SymbolTable::getLabel());
            }
            else{
                se = new TemporarySymbolEntry(TypeSystem::floatType,
                    SymbolTable::getLabel());
            }
            $$ = new BinaryExpr(se, BinaryExpr::ADD, $1, $3);
        }
    |   AddExp SUB MulExp {
            SymbolEntry *se;
            if($1->getType()->isInt() && $3->getType()->isInt()){
                se = new TemporarySymbolEntry(TypeSystem::intType,
                    SymbolTable::getLabel());
            }
            else{
                se = new TemporarySymbolEntry(TypeSystem::floatType,
                    SymbolTable::getLabel());
            }
            $$ = new BinaryExpr(se, BinaryExpr::SUB, $1, $3);
        }
    ;

// 乘法级表达式，乘法，除法
MulExp
    :   UnaryExp {
            $$ = $1;
        }
    |   MulExp MUL UnaryExp {
            SymbolEntry *se;
            if($1->getType()->isInt() && $3->getType()->isInt()){
                se = new TemporarySymbolEntry(TypeSystem::intType,
                    SymbolTable::getLabel());
            }
            else{
                se = new TemporarySymbolEntry(TypeSystem::floatType,
                    SymbolTable::getLabel());
```

```
54              }
55              $$ = new BinaryExpr(se, BinaryExpr::MUL, $1, $3);
56          }
57      |   MulExp DIV UnaryExp {
58              SymbolEntry *se;
59              if($1->getType()->isInt() && $3->getType()->isInt()){
60                  se = new TemporarySymbolEntry(TypeSystem::intType,
                        SymbolTable::getLabel());
61              }
62              else{
63                  se = new TemporarySymbolEntry(TypeSystem::floatType,
                        SymbolTable::getLabel());
64              }
65              $$ = new BinaryExpr(se, BinaryExpr::DIV, $1, $3);
66          }
67      |   MulExp MOD UnaryExp {
68              SymbolEntry *se;
69              if($1->getType()->isInt() && $3->getType()->isInt()){
70                  se = new TemporarySymbolEntry(TypeSystem::intType,
                        SymbolTable::getLabel());
71              }
72              else{
73                  se = new TemporarySymbolEntry(TypeSystem::floatType,
                        SymbolTable::getLabel());
74              }
75              $$ = new BinaryExpr(se, BinaryExpr::MOD, $1, $3);
76          }
77      ;
78
79  // 一元表达式，单目运算符
80  UnaryExp
81      :   PrimaryExp {
82              $$ = $1;
83          }
84      |   ID LPAREN FuncRParams RPAREN {
85              SymbolEntry *se;
86              se = identifiers->lookup($1);
87              if(se == nullptr)
88              {
89                  fprintf(stderr, "identifier \"%s\" is undefined\n", (char*)$1);
90                  delete [](char*)$1;
91                  assert(se != nullptr);
92              }
93              SymbolEntry *tmp = new TemporarySymbolEntry(se->getType(),
                    SymbolTable::getLabel());
94              $$ = new FuncCallNode(tmp, new Id(se), (FuncCallParamsNode*)$3);
95          }
96      |   ADD UnaryExp {
97              //$$ = $2;
98              SymbolEntry *tmp = new TemporarySymbolEntry($2->getType(),
```

```
                SymbolTable::getLabel());
99              $$ = new OneOpExpr(tmp, OneOpExpr::ADD, $2);
100         }
101     |   SUB UnaryExp {
102             SymbolEntry *tmp = new TemporarySymbolEntry($2->getType(),
                    SymbolTable::getLabel());
103             $$ = new OneOpExpr(tmp, OneOpExpr::SUB, $2);
104         }
105     |   NOT UnaryExp {
106             SymbolEntry *tmp = new TemporarySymbolEntry($2->getType(),
                    SymbolTable::getLabel());
107             $$ = new OneOpExpr(tmp, OneOpExpr::NOT, $2);
108         }
109     ;
110 }
```

4.3.4 条件表达式

条件表达式使用了递归和运算符优先级的方式来构建对应的语法树节点，其中或的优先级最低

4.3.5 关系表达式

需要注意的是不管运算节点类型是什么，关系表达式的运算结果节点是 bool 类型。

4.3.6 数组

数组的常量和变量下标表示需要考虑到一维数组和多维数组的多种情况，数组的大小可由常量表达式或变量表达式决定。

4.3.7 常量

常量有可能以列表形式定义，DelStmt 节点用以表示包含多个常量定义的语句。单独的常量定义定义 ConstDef, 一种是常量标识符结点，还可能是常量初始化值的结点。此处还考虑到了数组常量初始化的可能情况。

```
// 关系表达式
RelExp
    :   AddExp {
            $$ = $1;
        }
    |   RelExp LESS AddExp {
            SymbolEntry *se = new TemporarySymbolEntry(TypeSystem::boolType, SymbolTable::getLabel());
            $$ = new BinaryExpr(se, BinaryExpr::LESS, $1, $3);
        }
    |   RelExp LESSEQ AddExp {
            SymbolEntry *se = new TemporarySymbolEntry(TypeSystem::boolType, SymbolTable::getLabel());
            $$ = new BinaryExpr(se, BinaryExpr::LESSEQ, $1, $3);
        }
    |   RelExp GREAT AddExp {
            SymbolEntry *se = new TemporarySymbolEntry(TypeSystem::boolType, SymbolTable::getLabel());
            $$ = new BinaryExpr(se, BinaryExpr::GREAT, $1, $3);
        }
    |   RelExp GREATEQ AddExp {
            SymbolEntry *se = new TemporarySymbolEntry(TypeSystem::boolType, SymbolTable::getLabel());
            $$ = new BinaryExpr(se, BinaryExpr::GREATEQ, $1, $3);
        }
    ;
```

```
// 数组的常量下标表示
ArrConstIndices
    :   ArrConstIndices LBRACKET ConstExp RBRACKET {
            ExprStmtNode* node = (ExprStmtNode*)$1;
            node->addNext($3);
            $$ = node;
        }
    |   LBRACKET ConstExp RBRACKET {
            ExprStmtNode* node = new ExprStmtNode();
            node->addNext($2);
            $$ = node;
        }
    ;

// 数组的变量下标表示
ArrValIndices
    :   ArrValIndices LBRACKET Exp RBRACKET {
            ExprStmtNode* node = (ExprStmtNode*)$1;
            node->addNext($3);
            $$ = node;
        }
    |   LBRACKET Exp RBRACKET {
            ExprStmtNode* node = new ExprStmtNode();
            node->addNext($2);
            $$ = node;
        }
    ;
```

常量定义在第一步规约的时候，使用第二个文法。在第二个文法对应的语义动作中，可以申请一个 ConstDefList 节点，并将当前的 ConstDef 节点作为子节点插入到 ConstDefList 节点的子节点数组中。然后后续就会使用第一个文法进行规约，说明已经执行完了第二个文法对应的语义动作，ConstDefList 节点已经有了内存空间，并且前序的子节点已经插入到了 ConstDefList 节点的子节点中，此时只需要把新的 ConstDef 节点从后面插入即可，然后再把这个 ConstDefList 节点赋值给根节点。

Listing 3: 常量

```
// 常量定义列表
ConstDefList
    :   ConstDefList COMMA ConstDef {
            DeclStmt* node = (DeclStmt*) $1;
            node->addNext((DefNode*)$3);
            $$ = node;
        }
    |   ConstDef {
            DeclStmt* node = new DeclStmt(true);
            node->addNext((DefNode*)$1);
            $$ = node;
        }
    ;

// 常量定义
```

```
ConstDef
    :   ID ASSIGN ConstExp {
            Type* type;
            if(currentType->isInt()){
                type = TypeSystem::constIntType;
            }
            else{
                type = TypeSystem::constFloatType;
            }
            SymbolEntry *se;
            se = new IdentifierSymbolEntry(type, $1, identifiers->getLevel());
            identifiers->install($1, se);
            $$ = new DefNode(new Id(se), (Node*)$3, true, false);//类型向上转换
        }
    |   ID ArrConstIndices ASSIGN ConstInitVal{
            Type* type;
            if(currentType->isInt()){
                type = new ConstIntArrayType();
            }
            else{
                type = new ConstFloatArrayType();
            }
            SymbolEntry *se;
            se = new IdentifierSymbolEntry(type, $1, identifiers->getLevel());
            identifiers->install($1, se);
            Id* id = new Id(se);
            id->addIndices((ExprStmtNode*)$2);
            $$ = new DefNode(id, (Node*)$4, true, true);//类型向上转换
        }
    ;

// 常量的初始化值的可能形式
ConstInitVal
    :   ConstExp {
            InitValNode* newNode = new InitValNode(true);
            newNode->setLeafNode((ExprNode*)$1);
            $$ = newNode;
        }
    |   LBRACE ConstInitValList RBRACE{
            $$ = $2;
        }
    |   LBRACE RBRACE{
            $$ = new InitValNode(true);
    }
    ;

// 数组常量初始化列表
ConstInitValList
    :   ConstInitValList COMMA ConstInitVal{
            InitValNode* node = (InitValNode*)$1;
```

```
66              node->addNext((InitValNode*)$3);
67              $$ = node;
68          }
69      |   ConstInitVal{
70              InitValNode* newNode = new InitValNode(true);
71              newNode->addNext((InitValNode*)$1);
72              $$ = newNode;
73          }
74      ;
```

4.3.8 变量

变量有可能以列表形式定义，DelStmt 节点用以表示包含多个变量定义的语句。单独的变量定义定义 VarDef, 一种是变量标识符结点，还可能是变量初始化值的结点。此处还考虑到了数组变量初始化的可能情况。

Listing 4: 常量

```
1  // 变量定义列表
2  VarDefList
3      :   VarDefList COMMA VarDef {    //多个用逗号分隔
4              DeclStmt* node = (DeclStmt*) $1;
5              node->addNext((DefNode*)$3);
6              $$ = node;
7          }
8      |   VarDef {
9              DeclStmt* node = new DeclStmt(true);
10             node->addNext((DefNode*)$1);
11             $$ = node;
12         }
13     ;
14
15 // 变量定义
16 VarDef
17     :   ID {
18             Type* type;
19             if(currentType->isInt()){
20                 type = TypeSystem::intType;
21             }
22             else{
23                 type = TypeSystem::floatType;
24             }
25             SymbolEntry *se;
26             se = new IdentifierSymbolEntry(type, $1, identifiers->getLevel());
27             identifiers->install($1, se);
28             $$ = new DefNode(new Id(se), nullptr, false, false);
29         }
30     |   ID ASSIGN Exp {
31             Type* type;
```

```
32              if(currentType->isInt()){
33                  type = TypeSystem::intType;
34              }
35              else{
36                  type = TypeSystem::floatType;
37              }
38              SymbolEntry *se;
39              se = new IdentifierSymbolEntry(type, $1, identifiers->getLevel());
40              identifiers->install($1, se);
41              $$ = new DefNode(new Id(se), (Node*)$3, false, false);//类型向上转换
42          }
43      |   ID ArrConstIndices {
44              Type* type;
45              if(currentType->isInt()){
46                  type = new IntArrayType();
47              }
48              else{
49                  type = new FloatArrayType();
50              }
51              SymbolEntry *se;
52              se = new IdentifierSymbolEntry(type, $1, identifiers->getLevel());
53              identifiers->install($1, se);
54              Id* id = new Id(se);
55              id->addIndices((ExprStmtNode*)$2);
56              $$ = new DefNode(id, nullptr, false, true);//类型向上转换
57          }
58      |   ID ArrConstIndices ASSIGN VarInitVal{
59              Type* type;
60              if(currentType->isInt()){
61                  type = new IntArrayType();
62              }
63              else{
64                  type = new FloatArrayType();
65              }
66              SymbolEntry *se;
67              se = new IdentifierSymbolEntry(type, $1, identifiers->getLevel());
68              identifiers->install($1, se);
69              Id* id = new Id(se);
70              id->addIndices((ExprStmtNode*)$2);
71              $$ = new DefNode(id, (Node*)$4, false, true);//类型向上转换
72          }
73      ;
74
75  // 变量初始化值
76  VarInitVal
77      :   Exp {
78              InitValNode* node = new InitValNode(false);
79              node->setLeafNode((ExprNode*)$1);
80              $$ = node;
81          }
```

```
82      |   LBRACE VarInitValList RBRACE{    //数组变量初始化
83              $$ = $2;
84          }
85      |   LBRACE RBRACE{
86              $$ = new InitValNode(false);
87      }
88      ;
89
90  // 数组变量初始化列表
91  VarInitValList
92      :   VarInitValList COMMA VarInitVal{
93              InitValNode* node = (InitValNode*)$1;
94              node->addNext((InitValNode*)$3);
95              $$ = node;
96          }
97      |   VarInitVal{
98              InitValNode* newNode = new InitValNode(false);
99              newNode->addNext((InitValNode*)$1);
100             $$ = newNode;
101         }
102     ;
```

4.3.9 函数

FuntionType 中维护了函数参数的类型，在函数参数语法树构建完成后，需要将函数参数的类型写到函数类型对应的域当中。函数参数考虑多个函数参数组成的函数参数和单个函数参数 FuncParam，函数参数包含参数类型和参数名，此外还要考虑数组类型的参数还包括数组的维度信息。

Listing 5: 函数

```
1   // 函数定义，函数名、参数列表和函数体
2   FuncDef
3       :   Type ID {
4               Type *funcType;
5               funcType = new FunctionType($1,{});
6               SymbolEntry *se = new IdentifierSymbolEntry(funcType, $2,
                    identifiers->getLevel());
7               identifiers->install($2, se);
8               identifiers = new SymbolTable(identifiers);
9           }
10          LPAREN FuncParams{
11              SymbolEntry *se;
12              se = identifiers->lookup($2);
13              assert(se != nullptr);
14              if($5!=nullptr){
15                  //将函数参数类型写入符号表
16                  ((FunctionType*)(se->getType()))->setparamsType(((FuncDefParamsNode*)$5)->getParams
17              }
18          }
```

```
        RPAREN BlockStmt {
            SymbolEntry *se;
            se = identifiers->lookup($2);
            assert(se != nullptr);
            $$ = new FunctionDef(se, (FuncDefParamsNode*)$5, $8);
            SymbolTable *top = identifiers;
            identifiers = identifiers->getPrev();
            delete top;
            delete []$2;
        }
    ;

// 函数参数列表
FuncParams
    :   FuncParams COMMA FuncParam {
            FuncDefParamsNode* node = (FuncDefParamsNode*)$1;
            node->addNext(((DefNode*)$3)->getId());
            $$ = node;
        }
    |   FuncParam {
            FuncDefParamsNode* node = new FuncDefParamsNode();
            node->addNext(((DefNode*)$1)->getId());
            $$ = node;
        }
    |   %empty {
            $$ = nullptr;
        }
    ;

// 函数参数, 参数类型和参数名, 数组类型的参数还包括数组的维度信息
FuncParam
    :   Type ID {
            SymbolEntry *se = new IdentifierSymbolEntry($1, $2,
                identifiers->getLevel());
            identifiers->install($2, se);
            $$ = new DefNode(new Id(se), nullptr, false, false);
        }
    //数组函数参数
    |   Type ID LBRACKET RBRACKET ArrConstIndices{
            if($1==TypeSystem::intType){
                Type* arrayType;
                arrayType = new IntArrayType();
                ((IntArrayType*)arrayType)->pushBackDimension(-1);
                //最高维未指定, 记为默认值-1
                SymbolEntry *se = new IdentifierSymbolEntry(arrayType, $2,
                    identifiers->getLevel());
                identifiers->install($2, se);
                Id* id = new Id(se);
                id->addIndices((ExprStmtNode*)$5);
                $$ = new DefNode(id, nullptr, false, true);
```

```
67              }
68              else if($1==TypeSystem::floatType){
69                  Type* arrayType;
70                  arrayType = new FloatArrayType();
71                  ((FloatArrayType*)arrayType)->pushBackDimension(-1);
72                  //最高维未指定，记为默认值-1
73                  SymbolEntry *se = new IdentifierSymbolEntry(arrayType, $2,
                        identifiers->getLevel());
74                  identifiers->install($2, se);
75                  Id* id = new Id(se);
76                  id->addIndices((ExprStmtNode*)$5);
77                  $$ = new DefNode(id, nullptr, false, true);
78              }
79          }
80      |   Type ID LBRACKET RBRACKET{
81              if($1==TypeSystem::intType){
82                  Type* arrayType;
83                  arrayType = new IntArrayType();
84                  ((IntArrayType*)arrayType)->pushBackDimension(-1);
85                  //最高维未指定，记为默认值-1
86                  SymbolEntry *se = new IdentifierSymbolEntry(arrayType, $2,
                        identifiers->getLevel());
87                  identifiers->install($2, se);
88                  Id* id = new Id(se);
89                  $$ = new DefNode(id, nullptr, false, true);
90              }
91              else if($1==TypeSystem::floatType){
92                  Type* arrayType;
93                  arrayType = new FloatArrayType();
94                  ((FloatArrayType*)arrayType)->pushBackDimension(-1);
95                  //最高维未指定，记为默认值-1
96                  SymbolEntry *se = new IdentifierSymbolEntry(arrayType, $2,
                        identifiers->getLevel());
97                  identifiers->install($2, se);
98                  Id* id = new Id(se);
99                  $$ = new DefNode(id, nullptr, false, true);
100             }
101         }
102     ;
```

5 类型检查及中间代码生成

5.1 实验内容

类型检查是编译过程的重要一步，以确保操作对象与操作符相匹配。每一个表达式都有相关联的类型，如关系运算表达式的类型为布尔型，而计算表达式的类型一般为整型或浮点型等。类型检查的目的在于找出源代码中不符合类型表达式规定的代码，在最终的代码生成之前报错，使得程序员根据错误信息对源代码进行修正。本学期实验中类型检查阶段完成了对于未声明变量及同一作用域下重复声

明的变量的检查；实现条件判断表达式 int 至 bool 类型的隐式类型转换；检查数值运算表达式运算数类型是否正确（如返回值为 void 的函数调用结果是否参与了其他表达式的运算）；检查未声明函数及函数形参是否与实参类型及数目匹配；检查 return 语句操作数和函数声明的返回值类型是否匹配；对 break、while 语句进行静态检查，判断是否仅出现在 while 语句中。

中间代码生成是在词法分析语法分析的基础上，将 SysY 源代码翻译成中间代码。中间代码位于源代码和目标代码直接，是一种抽象的、中间层次的编程语言表示，通常比源代码更接近底层的机器代码，但比目标代码更抽象。中间代码生成主要包含对数据流和控制流两种类型的翻译，数据流包括表达式运算、变量声明与赋值等，控制流包括 if、while、break、continue 等语句。

5.2 实验效果

example.sy

```
1 int main()
2 {
3     int a;
4     int b;
5     int min;
6     a = 1 + 2 + 3;
7     b = 2 + 3 + 4;
8     if (a < b)
9         min = a;
10    else
11        min = b;
12    return min;
13 }
```

图 5.5: example.sy 截图

example.ll

```
1 declare i32 @getint()
2 declare void @putint(i32)
3 declare i32 @getch()
4 declare void @putch(i32)
5 declare void @putf(i32)
6
7 define i32 @main() {
8 B17:
9   %t20 = alloca i32, align 4
10  %t19 = alloca i32, align 4
11  %t18 = alloca i32, align 4
12  %t4 = add i32 1, 2
13  %t5 = add i32 %t4, 3
14  store i32 %t5, i32* %t18, align 4
15  %t7 = add i32 2, 3
16  %t8 = add i32 %t7, 4
17  store i32 %t8, i32* %t19, align 4
18  %t9 = load i32, i32* %t18, align 4
19  %t10 = load i32, i32* %t19, align 4
20  %t24 = icmp slt i32 %t9, %t10
21  br i1 %t24, label %B21, label %B22
22 B21:                                ; preds = %B17
23  %t13 = load i32, i32* %t18, align 4
24  store i32 %t13, i32* %t20, align 4
25  br label %B23
26 B22:                                ; preds = %B17
27  %t15 = load i32, i32* %t19, align 4
28  store i32 %t15, i32* %t20, align 4
29  br label %B23
30 B23:                                ; preds = %B21, %B22
31  %t16 = load i32, i32* %t20, align 4
32  ret i32 %t16
33 }
```

图 5.6: example.sy 类型检查及中间代码生成后截图

5.3 代码设计-类型检查

类型检查最简单的实现方式是在建立语法树的过程中进行相应的识别和处理，也可以在建树完成后，自底向上遍历语法树进行类型检查。类型检查过程中，父结点需要检查孩子结点的类型，并根据孩子结点类型确定自身类型。有一些表达式可以在语法制导翻译时就确定类型，比如整数就是整型，这些表达式通常是语法树的叶结点。而有些表达式则依赖其子表达式确定类型，这些表达式则是语法树中的内部结点。

5.3.1 对于未声明变量及同一作用域下重复声明的变量的检查

符号表中 lookup 函数，通过遍历当前符号表及其上层符号表，直到到达符号表的顶层，检查给定名字的符号表项是否存在。如果在所有符号表中都未找到，则返回 nullptr，表示未声明的变量。

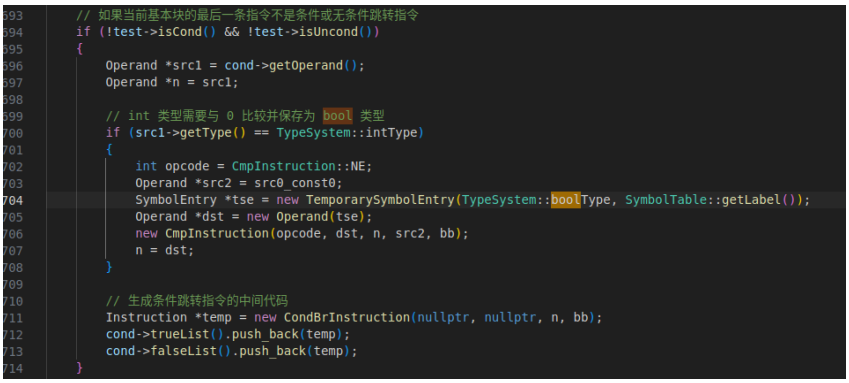
符号表中 install 函数，当要插入一个新的符号表项时，首先检查是否存在同名函数的重定义，如果是，则将新的函数符号表项链接到已存在函数符号表项的末尾。

```
/*
    Description: lookup the symbol entry of an identifier in the symbol table
    Parameters:
        name: identifier name
    Return: pointer to the symbol entry of the identifier

    hint:
    1. The symbol table is a stack. The top of the stack contains symbol entries in
        the current scope.
    2. Search the entry in the current symbol table at first.
    3. If it's not in the current table, search it in previous ones(along the 'prev'
        link).
    4. If you find the entry, return it.
    5. If you can't find it in all symbol tables, return nullptr.
*/
SymbolEntry *SymbolTable::lookup(std::string name)
{
    SymbolTable *current = identifiers; // 指向当前符号表
    // 在当前符号表以及其上层符号表中查找名为name的符号表项
    while (current != nullptr)
        // symbolTable为map类型的成员变量
        // 如果在当前符号表中找到了名为name的符号表项，则返回该符号表项
        if (current->symbolTable.find(name) != current->symbolTable.end())
            return current->symbolTable[name];
        else
            // 否则，继续向上层符号表查找
            current = current->prev;
    return nullptr;// 都没找到
}

// install the entry into current symbol table.
void SymbolTable::install(std::string name, SymbolEntry *entry)
{
    // 同时检查是否有函数的重定义,如果有同名的函数，链入符号表项
    if (symbolTable.find(name) != symbolTable.end() &&
        symbolTable[name]->getType()->isFunc())
    {
        // 如果发现同名函数，则将新的函数符号表项链到已存在函数符号表项的末尾
        symbolTable[name]->setNext(entry);
    }
    else
    {
        // 否则，直接将符号表项插入当前符号表
```

```
41          symbolTable[name] = entry;
42      }
43  }
```

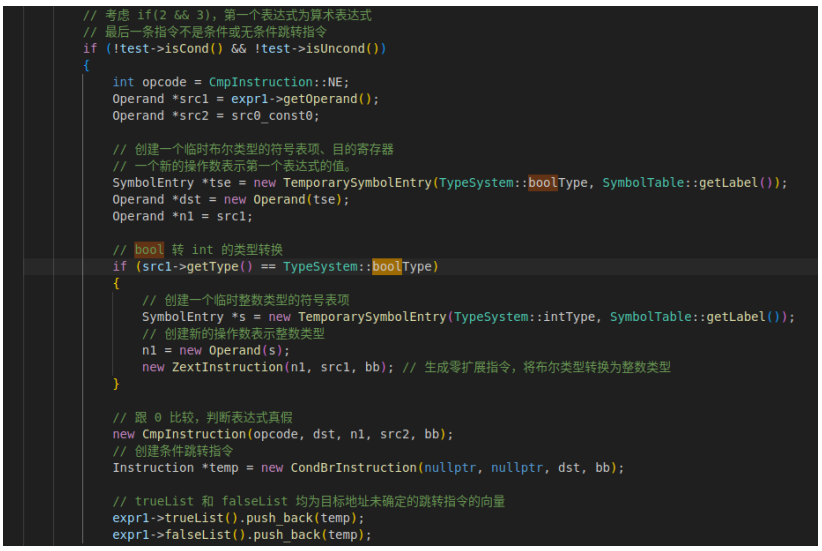
5.3.2 实现条件判断表达式 int 至 bool 类型的隐式类型转换

int 类型需要与 0 比较并保存为 bool 类型



```
// 如果当前基本块的最后一条指令不是条件或无条件跳转指令
if (!test->isCond() && !test->isUncond())
{
    Operand *src1 = cond->getOperand();
    Operand *n = src1;

    // int 类型需要与 0 比较并保存为 bool 类型
    if (src1->getType() == TypeSystem::intType)
    {
        int opcode = CmpInstruction::NE;
        Operand *src2 = src0_const0;
        SymbolEntry *tse = new TemporarySymbolEntry(TypeSystem::boolType, SymbolTable::getLabel());
        Operand *dst = new Operand(tse);
        new CmpInstruction(opcode, dst, n, src2, bb);
        n = dst;
    }

    // 生成条件跳转指令的中间代码
    Instruction *temp = new CondBrInstruction(nullptr, nullptr, n, bb);
    cond->trueList().push_back(temp);
    cond->falseList().push_back(temp);
}
```

创建一个临时整数类型的符号表项, 创建新的操作数表示整数类型，通过生成零扩展指令，将布尔类型转换为整数类型



```
// 考虑 if(2 && 3)，第一个表达式为算术表达式
// 最后一条指令不是条件或无条件跳转指令
if (!test->isCond() && !test->isUncond())
{
    int opcode = CmpInstruction::NE;
    Operand *src1 = expr1->getOperand();
    Operand *src2 = src0_const0;

    // 创建一个临时布尔类型的符号表项、目的寄存器
    // 一个新的操作数表示第一个表达式的值。
    SymbolEntry *tse = new TemporarySymbolEntry(TypeSystem::boolType, SymbolTable::getLabel());
    Operand *dst = new Operand(tse);
    Operand *n1 = src1;

    // bool 转 int 的类型转换
    if (src1->getType() == TypeSystem::boolType)
    {
        // 创建一个临时整数类型的符号表项
        SymbolEntry *s = new TemporarySymbolEntry(TypeSystem::intType, SymbolTable::getLabel());
        // 创建新的操作数表示整数类型
        n1 = new Operand(s);
        new ZextInstruction(n1, src1, bb); // 生成零扩展指令，将布尔类型转换为整数类型
    }

    // 跟 0 比较，判断表达式真假
    new CmpInstruction(opcode, dst, n1, src2, bb);
    // 创建条件跳转指令
    Instruction *temp = new CondBrInstruction(nullptr, nullptr, dst, bb);

    // trueList 和 falseList 均为目标地址未确定的跳转指令的向量
    expr1->trueList().push_back(temp);
    expr1->falseList().push_back(temp);
```

5.3.3 检查数值运算表达式运算数类型是否正确（如返回值为 void 的函数调用结果是否参与了其他表达式的运算）

对于运算表达式结点，检查参与运算的表达式的结果，如果参与运算的表达式运算的结果是空，则表示 void 参与了运算，报错。

```
452         // 类型检查
453         if (expr1->getOperand() == nullptr)
454         {
455             fprintf(stderr, "BinaryExpr can't be void type\n");
456             assert(expr1->getOperand() != nullptr);
457         }
458         if (expr2->getOperand() == nullptr)
459         {
460             fprintf(stderr, "BinaryExpr can't be void type\n");
461             assert(expr2->getOperand() != nullptr);
462         }
```

5.3.4 检查未声明函数及函数形参是否与实参类型及数目匹配

在 FuncCallExpr::typeCheck 函数中，首先进行了函数调用的准备工作，包括获取实参的数量、获取函数声明的参数类型列表等。然后，通过遍历同名的函数列表，考虑同名重载函数。在循环中，获取当前函数声明的参数类型列表，并比较实参和形参的参数数目。尝试找到匹配的函数。如果找到了匹配的函数（参数数量相同），则设置 symbolEntry 为当前函数的符号表项，退出循环。如果找到了匹配的函数，获取该函数的返回值类型，再返回值不是 void 的情况下创建临时符号表项和相应的操作数寄存器，用于保存函数调用的返回值。通过遍历函数声明的参数类型列表和实参列表，对比它们的类型和数量是否匹配。如果不匹配，输出相应的错误信息

```
1  // 函数调用表达式，判断函数调用时形参及实参类型或数目是否一致
2  // 涉及到非重载函数的重复声明的判断
3  FuncCallExpr::FuncCallExpr(SymbolEntry *se, ExprNode *param)
4      : ExprNode(se), param(param)
5  {
6      // 存储函数调用返回值
7      dst = nullptr;
8
9      // 统计实参计数
10     int actualParamCount = 0;
11     ExprNode *temp1 = param;
12     while (temp1)
13     {
14         actualParamCount++;
15         temp1 = (ExprNode *)(temp1->getNext());
16     }
17
18     SymbolEntry *temp2 = se;
19
20     // 遍历同名的函数列表（考虑同名重载函数）
21     while (temp2)
22     {
23         // 获取当前函数声明的参数类型列表
24         std::vector<Type *> params = ((FunctionType
               *)temp2->getType())->getParamsType();
25
26         // 比较实参和形参的参数数目，尝试找到正确的函数对应的符号表项
27         if ((long unsigned int)actualParamCount == params.size())
28         {
29             // 设置该函数名对应的符号表项
30             this->symbolEntry = temp2;
31             break;
```

```
32          }
33
34          // 给符号表项都加了一个next成员变量,专用于处理同名函数
35          temp2 = temp2->getNext();
36      }
37
38      if (symbolEntry)//如果找到了匹配的函数
39      {
40          // 获取函数返回值的类型
41          this->type = ((FunctionType *)symbolEntry->getType())->getRetType();
42
43          // 返回值类型不是void, 构造目标寄存器的保存操作数
44          if (this->type != TypeSystem::voidType)
45          {
46              // 创建临时符号表项, 保存函数调用的返回值
47              SymbolEntry *se = new TemporarySymbolEntry(this->type,
                    SymbolTable::getLabel());
48              dst = new Operand(se);// 创建目标操作数, 指向临时符号表项
49          }
50
51          // 获取参数的类型列表
52          std::vector<Type *> params = ((FunctionType
                *)symbolEntry->getType())->getParamsType();
53
54          // temp初始化等于实参param
55          ExprNode *temp = param;
56          //遍历函数声明的参数类型列表
57          for (auto it = params.begin(); it != params.end(); it++)
58          {
59              if (temp == nullptr)//判断实参数目是否匹配
60              {
61                  fprintf(stderr, "actual Param numbers %d not match in function %s\n",
                        actualParamCount, symbolEntry->toStr().c_str());
62                  assert(temp != nullptr);
63              }
64              else if ((*it)->getKind() != Type::INT)// 判断实参类型是否为INT
65              {
66                  fprintf(stderr, "arguments type not match in function %s\n",
                        symbolEntry->toStr().c_str());
67                  assert(temp != nullptr);
68              }
69              temp = (ExprNode *)(temp->getNext());//移动到下一实参
70          }
71          if (temp != nullptr)//判断实参数目是否匹配
72          {
73              fprintf(stderr, "actual param numbers %d not match in function %s\n",
                    actualParamCount, symbolEntry->toStr().c_str());
74              assert(temp != nullptr);
75          }
76      }
```

```
};
```

5.3.5 检查 return 语句操作数和函数声明的返回值类型是否匹配

检查 return 语句操作数和函数声明的返回值类型是否匹配通过 checkRet 函数实现。判断没有返回值的情况，如果没有返回值，返回值类型应该是 void，如果返回值类型是其他类型，则存在类型错误。如果有返回值，int 和 constint 兼容，如果返回值是函数类型且该函数的返回值是 int，则类型正确，否则存在类型错误。

```
// 实现checkRet函数的主要的判断
// 判断return 语句操作数和函数声明的返回值类型是否匹配
bool ReturnStmt::checkRet(Type *retType)
{
    // 先判断没有返回值的情况
    if (!retValue)
    {
        // 返回值类型为void，则正确，否则均有类型错误
        if (retType != TypeSystem::voidType)
        {
            fprintf(stderr, "function has no return statement or function type
                doesn't match return type\n");
            assert(retType == TypeSystem::voidType);
            return false;
        }
        else
        {
            return true;
        }
    }
    else
    {
        // 获取返回值类型
        Type *valueType = retValue->getSymPtr()->getType();

        //  考虑constINT和INT两种类型的兼融
        if ((retType == TypeSystem::constIntType || retType == TypeSystem::intType) &&
            (valueType == TypeSystem::constIntType || valueType ==
                TypeSystem::intType))
        {
            return true;
        }
        else
        {
            // 考虑返回值为函数类型的情况，如果其返回类型是int
            if (valueType->isFunc() && ((FunctionType *)valueType)->getRetType() ==
                TypeSystem::intType)
            {
                return true;
            }
```

```
38
39              fprintf(stderr, "function type doesn't match return type\n");
40              return false;
41          }
42      }
43  }
```

5.4 代码设计-中间代码生成

5.4.1 数据流语句的翻译

- 表达式运算

  builder 是 IRBuilder 类对象，用于传递继承属性，新生成的指令要插入的基本块，辅助进行中间代码生成。在二元加法减法运算中，首先通过 builder 得到后续生成的指令要插入的基本块 bb，然后生成子表达式的中间代码，通过 getOperand 函数得到子表达式的目的操作数，设置指令的操作码，最后生成相应的二元运算指令并插入到基本块 bb 中。

```
1
2   // 二元表达式运算以ADD、SUB、MUL、DIV、MOD为例
3
4   else if (op >= ADD && op <= MOD)
5       {
6           // 生成两个表达式的中间代码
7           expr1->genCode();
8           expr2->genCode();
9
10          // 如果第一个表达式的结果为空，报错
11          if (expr1->getOperand() == nullptr)
12          {
13              fprintf(stderr, "BinaryExpr can't be void type\n");
14              assert(expr1->getOperand() != nullptr);
15          }
16          if (expr2->getOperand() == nullptr)
17          {
18              fprintf(stderr, "BinaryExpr can't be void type\n");
19              assert(expr2->getOperand() != nullptr);
20          }
21           // 获取左右操作数的中间代码生成结果
22          Operand *src1 = expr1->getOperand();
23          Operand *src2 = expr2->getOperand();
24
25          // 根据运算符类型确定相应的中间代码操作
26          int opcode = 0;
27          switch (op)
28          {
29              case ADD:
30                  opcode = BinaryInstruction::ADD;
31                  break;
32              case SUB:
```

```
33                  opcode = BinaryInstruction::SUB;
34                  break;
35              case MUL:
36                  opcode = BinaryInstruction::MUL;
37                  break;
38              case DIV:
39                  opcode = BinaryInstruction::DIV;
40                  break;
41              case MOD:
42                  opcode = BinaryInstruction::MOD;
43                  break;
44          }
45
46          Operand *n1 = src1, *n2 = src2;
47          // 如果操作数类型为 bool, 进行隐式类型转换
48          if (src1->getType() == TypeSystem::boolType)
49          {
50              SymbolEntry *s = new TemporarySymbolEntry(TypeSystem::intType,
                    SymbolTable::getLabel());
51              n1 = new Operand(s);
52              new ZextInstruction(n1, src1, bb);
53          }
54          if (src2->getType() == TypeSystem::boolType)
55          {
56              SymbolEntry *s2 = new TemporarySymbolEntry(TypeSystem::intType,
                    SymbolTable::getLabel());
57              n2 = new Operand(s2);
58              new ZextInstruction(n2, src2, bb);
59          }
60          // 生成二元运算指令的中间代码
61          new BinaryInstruction(opcode, dst, src1, src2, bb);
62      }
63
64      // 一元运算表达式代码生成
65  void UnaryExpr::genCode()
66  {
67      // fprintf(stderr, "UnaryExpr::genCode\n");
68      // 生成子表达式的中间代码
69      expr->genCode();
70
71      // 检查子表达式的结果是否为空
72      if (expr->getOperand() == nullptr)
73      {
74          fprintf(stderr, "UnaryExpr can't be void type\n");
75          assert(expr->getOperand() != nullptr);
76      }
77
78      //获取当前插入基本块子表达式结果操作数
79      BasicBlock *bb = builder->getInsertBB();
80      Operand *src = expr->getOperand();
```

```
81      int opcode;
82
83      if (op == ADD || op == SUB)
84      {
85          // 处理一元加法和减法
86          switch (op)
87          {
88          case ADD:
89              opcode = UnaryInstruction::ADD;
90              break;
91          case SUB:
92              opcode = UnaryInstruction::SUB;
93              break;
94          }
95
96          dst = new Operand(symbolEntry);// 目的操作数
97          Operand *n = src;
98
99          // bool -> int 隐式类型转换
100         if (src->getType() == TypeSystem::boolType)
101         {
102             SymbolEntry *s = new TemporarySymbolEntry(TypeSystem::intType,
                    SymbolTable::getLabel());
103             n = new Operand(s);
104             new ZextInstruction(n, src, bb);
105         }
106
107         // 生成一元运算指令的中间代码
108         new UnaryInstruction(opcode, dst, n, bb);
109     }
110     else if (op == NOT)
111     {
112         // 处理逻辑非运算
113         // 子表达式类型int
114         if (expr->getOperand()->getType() == TypeSystem::intType)
115         {
116             opcode = CmpInstruction::E;
117             Operand *src = expr->getOperand();
118             Operand *src0 = src0_const0;
119             SymbolEntry *s = new TemporarySymbolEntry(TypeSystem::intType,
                    SymbolTable::getLabel());
120             Operand *n = new Operand(s);
121
122             // 隐式类型转换
123             // 如果目的操作数的类型与子表达式的类型不同
124             if (dst->getType() != src->getType())
125                 new ZextInstruction(n, dst, bb);
126             // 生成比较指令的中间代码
127             new CmpInstruction(opcode, this->dst, src, src0, bb);
128         }
```

```
129             // 处理逻辑非运算（等价于与 true 异或运算）
130             // 如果子表达式的类型是bool
131             else if (expr->getOperand()->getType() == TypeSystem::boolType)
132             {
133                 // 生成逻辑非指令的中间代码
134                 new NotInstruction(this->dst, src, bb);
135             }
136             fprintf(stderr, "NOT\n");
137         }
138 }
```

- 变量声明与初始化

  对于不带初始值的变量，变量的定义本身是不需要翻译成 IR 的，因为普通变量只需要在运行时分配内存即可，无需单独声明。对于带初始值的变量，只需要先生成计算变量初始值的中间代码，然后新建一个变量，并生成一条将变量的初始值赋给该变量的中间代码。对于常量，由于输入已经通过了类型检查，所以可以按照和变量一样的方法处理。

  对于全局变量，创建一个符号表项用于存储地址，将地址类型设置为指针类型，创建一个操作数表示地址，将地址操作数设置到符号表项中。如果有表达式，生成表达式的中间代码，创建一个全局变量指令将 ID 与表达式的操作数关联，并将其添加到符号表项中。如果没有表达式，创建一个全局变量指令，将 ID 与 nullptr 关联，并将其添加到符号表项中。

  对于局部变量或参数，类型设置为指针类型，创建一个临时符号表项储存地址，创建一个操作数表示地址，在函数栈中为局部变量分配空间，分配指令应该插入到入口块的最前面，设置符号表项的地址操作数，以便在后续代码生成中使用。如果是参数，将参数的值存储到刚分配的空间中，创建 store 指令将参数值存储到函数的地址中，将函数的操作数添加到参数操作数列表中。如果有表达式，生成表达式的代码，获取表达式的操作数，创建 store 指令将表达式的值存储到局部变量的地址中，串接后续函数参数/变量声明。

```
1  void DeclStmt::genCode()
2  {
3      //fprintf(stderr, "DeclStmt::genCode\n");
4      IdentifierSymbolEntry *se = dynamic_cast<IdentifierSymbolEntry
           *>(id->getSymPtr());
5
6      // 全局变量
7      if (se->isGlobal())
8      {
9          Operand *addr;
10         SymbolEntry *addr_se;
11
12         // 创建一个新的符号表项用于存储地址
13         addr_se = new IdentifierSymbolEntry(*se);
14         // 将地址的类型设置为指针类型
15         addr_se->setType(new PointerType(se->getType()));
16         // 创建一个操作数表示地址
17         addr = new Operand(addr_se);
18         // 将地址操作数设置到符号表项中
19         se->setAddr(addr);
```

```
20
21          Instruction *g;
22          if (expr != nullptr)
23          {
24              // 生成表达式的代码
25              expr->genCode();
26              // 创建一个全局变量指令，将ID与表达式的操作数关联，并将其添加到符号表项中
27              g = new GlobalInstruction(new Operand(id->getSymPtr()),
                    expr->getOperand(), se);
28          }
29          else
30          {
31              //
                    如果没有表达式，创建一个全局变量指令，将ID与nullptr关联，并将其添加到符号表项中
32              g = new GlobalInstruction(new Operand(id->getSymPtr()), nullptr, se);
33          }
34
35          // 将全局变量指令添加到全局指令列表中
36          global.push_back(g);
37      }
38      // 局部变量或参数
39      else if (se->isLocal() || se->isParam())
40      {
41          Function *func = builder->getInsertBB()->getParent();
42          BasicBlock *entry = func->getEntry();
43          Instruction *alloca;
44          Operand *addr;
45          SymbolEntry *addr_se;
46          Type *type;
47          // 将类型设置为指针类型
48          type = new PointerType(se->getType());
49          // 创建一个临时符号表项用于存储地址
50          addr_se = new TemporarySymbolEntry(type, SymbolTable::getLabel());
51          // 创建一个操作数表示地址
52          addr = new Operand(addr_se);
53          // 在函数栈中为局部变量分配空间
54          alloca = new AllocaInstruction(addr, se);
55          // 分配指令应该插入到入口块的最前面
56          entry->insertFront(alloca);
57          // 设置符号表项中的地址操作数，以便在后续代码生成中使用
58          se->setAddr(addr);
59
60          if (se->isParam())//是参数
61          {
62              // 将参数的值存储到刚刚分配的空间中
63              type = se->getType();
64              SymbolEntry *s = new TemporarySymbolEntry(type, SymbolTable::getLabel());
65              Operand *src = new Operand(s);
66              BasicBlock *bb = builder->getInsertBB();
67              // 创建store指令，将参数值存储到参数的地址中
```

```
68              new StoreInstruction(addr, src, bb);
69              // 将参数的操作数添加到函数的参数操作数列表中
70              func->pushParamsOperand(src);
71          }
72
73          if (expr != nullptr)
74          {
75              // 如果有表达式，生成表达式的代码
76              expr->genCode();
77              BasicBlock *bb = builder->getInsertBB();
78              // 获取表达式的操作数
79              Operand *src = expr->getOperand();
80              // 创建store指令，将表达式的值存储到局部变量的地址中
81              new StoreInstruction(addr, src, bb);
82          }
83      }
84
85      // 串接后续函数参数/变量声明
86      if (this->getNext())
87      {
88          this->getNext()->genCode();
89      }
90  }
91
92  // 变量声明语句
93  void DeclStmt::typeCheck()
94  {
95      // 进行标识符类型检查
96      id->typeCheck();
97
98      // 如果存在初始化表达式，则进行类型检查
99      if (expr)
100         expr->typeCheck();
101 }
```

5.4.2 控制流语句的翻译

控制流的翻译是本次实验的难点，需要通过回填技术来完成控制流的翻译。为每个结点设置两个综合属性 true_list 和 false_list, 它们是跳转目标未确定的基本块的列表，true_list 中的基本块为无条件跳转指令跳转到的目标基本块与条件跳转指令条件为真时跳转到的目标基本块，false_list 中的基本块为条件跳转指令条件为假时跳转到的目标基本块，这些目标基本块在翻译当前结点时尚不能确定，等到翻译其祖先结点能确定这些目标基本块时进行回填.

- 布尔表达式

  在本次实验中，要求支持逻辑短路，以与运算为例。与的逻辑与具有短路的特性，当第一个子表达式的值为假时，整个布尔表达式的值为假，第二个子表达式不会执行；当第一个子表达式的值为真时，根据第二个子表达式的值得到整个布尔表达式的值。对于与运算而言，只有两个运算数都为真的时候，运算结果才为真，在实验框架中，对于表达式节点提供了真分支和假分支，其中

保存了跳转语句，对于与运算而言，就是需要将两个运算数的假分支合并到一起，然后将真正确定的真分支对应跳转的目标块回填到真分支的跳转语句中。

或的逻辑也具有短路的特性，当第一个子表达式的值为真时，整个布尔表达式的值为真，第二个子表达式不会执行；当第一个子表达式的值为假时，根据第二个子表达式的值得到整个布尔表达式的值。或运算与 AND 逻辑区别在于，要将 false_List 中的跳转指令回填到 ntrueBB，true_list 为 expr1 的 true_List 与 expr2 的 true_List 合并后的结果，false_list 为 expr2 的 false_List。

```
void BinaryExpr::genCode()
{
    // fprintf(stderr, "BinaryExpr::genCode\n");
    // 获取当前插入基本块
    BasicBlock *bb = builder->getInsertBB();

    // 获取当前所处的函数
    Function *func = bb->getParent();

    if (op == AND)
    {
        // 生成第一个表达式的中间代码
        expr1->genCode();

        // 如果第一个表达式的结果为空，报错
        if (expr1->getOperand() == nullptr)
        {
            fprintf(stderr, "BinaryExpr can't be void type\n");
            assert(expr1->getOperand() != nullptr);
        }

        // 获取当前基本块的最后一条指令
        Instruction *test = bb->rbegin();

        // 考虑 if(2 && 3)，第一个表达式为算术表达式
        // 最后一条指令不是条件或无条件跳转指令
        if (!test->isCond() && !test->isUncond())
        {
            int opcode = CmpInstruction::NE;
            Operand *src1 = expr1->getOperand();
            Operand *src2 = src0_const0;

            // 创建一个临时布尔类型的符号表项、目的寄存器
            // 一个新的操作数表示第一个表达式的值。
            SymbolEntry *tse = new TemporarySymbolEntry(TypeSystem::boolType,
                SymbolTable::getLabel());
            Operand *dst = new Operand(tse);
            Operand *n1 = src1;

            // bool 转 int 的类型转换
            if (src1->getType() == TypeSystem::boolType)
            {
```

```
42                  // 创建一个临时整数类型的符号表项
43                  SymbolEntry *s = new TemporarySymbolEntry(TypeSystem::intType,
                        SymbolTable::getLabel());
44                  // 创建新的操作数表示整数类型
45                  n1 = new Operand(s);
46                  new ZextInstruction(n1, src1, bb); //
                        生成零扩展指令，将布尔类型转换为整数类型
47              }
48
49              // 跟 0 比较，判断表达式真假
50              new CmpInstruction(opcode, dst, n1, src2, bb);
51              // 创建条件跳转指令
52              Instruction *temp = new CondBrInstruction(nullptr, nullptr, dst, bb);
53
54              // trueList 和 falseList 均为目标地址未确定的跳转指令的向量
55              expr1->trueList().push_back(temp);
56              expr1->falseList().push_back(temp);
57          }
58
59           // 新建一个基本块 trueBB，用于存储逻辑与操作的真分支
60           // 如果第一个表达式的结果为真，那么跳到这个基本块
61          BasicBlock *trueBB = new BasicBlock(func);
62
63          // 将 trueList 中的跳转指令回填到 trueBB
64          backPatch(expr1->trueList(), trueBB, true);
65
66          // 设置当前的新的指令（第二个表达式对应的指令）要插入到的基本块为 trueBB
67          builder->setInsertBB(trueBB);
68
69          // trueBB 与当前基本块连接
70          trueBB->addPred(bb);
71          bb->addSucc(trueBB);
72
73          // 生成第二个表达式的中间代码
74          expr2->genCode();
75
76          // 如果第二个表达式的结果为空，报错
77          if (expr2->getOperand() == nullptr)
78          {
79              fprintf(stderr, "BinaryExpr can't be void type\n");
80              assert(expr2->getOperand() != nullptr);
81          }
82
83          // 更新当前基本块
84          bb = builder->getInsertBB();
85          test = bb->rbegin();
86
87          if (!test->isCond() && !test->isUncond())
88          {
89              int opcode = CmpInstruction::NE;
```

```
90              Operand *src1 = expr2->getOperand();
91              Operand *src2 = src0_const0;
92
93              // 目的寄存器
94              SymbolEntry *tse = new TemporarySymbolEntry(TypeSystem::boolType,
                    SymbolTable::getLabel());
95              Operand *dst = new Operand(tse);
96              Operand *n1 = src1;
97
98              if (src1->getType() == TypeSystem::boolType)
99              {
100                 SymbolEntry *s = new TemporarySymbolEntry(TypeSystem::intType,
                        SymbolTable::getLabel());
101                 n1 = new Operand(s);
102                 new ZextInstruction(n1, src1, bb);
103             }
104
105             // 跟 0 比较, 判断表达式真假
106             new CmpInstruction(opcode, dst, n1, src2, bb);
107             Instruction *temp = new CondBrInstruction(nullptr, nullptr, dst, bb);
108
109             expr2->trueList().push_back(temp);
110             expr2->falseList().push_back(temp);
111         }
112
113         // true_list 为 expr2 的 trueList, false_list 为 expr1 的 falseList 与 expr2
                的 falseList 合并后的结果
114         true_list = expr2->trueList();
115         false_list = merge(expr1->falseList(), expr2->falseList());
116     }
117     ……
118 }
```

- if-else

  对于关系运算，由于其运算结果会出现真假两个分支，考虑到逻辑短路，需要存储其真假分支的目标跳转块。但是由于在对该运算生成中间代码的时候还不知道最终的目的块，因此暂时用三个块代替。在真分支中增加一个条件跳转指令，条件为真跳转到真分支目标块，条件为假跳转到假分支目标块，在假分支中使用无条件跳转指令直接跳转到合并块。

  if-else 要将 false_List 回填到 else_bb,true_List 回填到 then_bb，在 else 分支中生成无条件跳转指令的中间代码跳转到合并块。

```
1  void IfElseStmt::genCode()
2  {
3      // fprintf(stderr, "IfElseStmt::genCode\n");
4      Function *func;
5      BasicBlock *then_bb, *else_bb, *end_bb;
6
7      // 获取当前函数和新建基本块
```

```
8       func = builder->getInsertBB()->getParent();
9       then_bb = new BasicBlock(func);
10      else_bb = new BasicBlock(func);
11      end_bb = new BasicBlock(func);
12
13      // 生成条件表达式的中间代码
14      cond->genCode();
15
16      // 获取当前基本块
17      BasicBlock *bb = builder->getInsertBB();
18      Instruction *test = bb->rbegin();
19
20      // 如果当前基本块的最后一条指令不是条件或无条件跳转指令
21      if (!test->isCond() && !test->isUncond())
22      {
23          Operand *src1 = cond->getOperand();
24          Operand *n = src1;
25
26          // int 类型需要与 0 比较并保存为 bool 类型
27          if (src1->getType() == TypeSystem::intType)
28          {
29              int opcode = CmpInstruction::NE;
30              Operand *src2 = src0_const0;
31              SymbolEntry *tse = new TemporarySymbolEntry(TypeSystem::boolType,
                    SymbolTable::getLabel());
32              Operand *dst = new Operand(tse);
33              new CmpInstruction(opcode, dst, src1, src2, bb);
34              n = dst;
35          }
36
37          // 生成条件跳转指令的中间代码
38          Instruction *temp = new CondBrInstruction(nullptr, nullptr, n, bb);
39          cond->trueList().push_back(temp);
40          cond->falseList().push_back(temp);
41      }
42
43      // 回填
44      backPatch(cond->trueList(), then_bb, true);
45      backPatch(cond->falseList(), else_bb, false);
46
47      // 设置新基本块插入点并生成 then 语句的中间代码
48      builder->setInsertBB(then_bb);
49      if (thenStmt != nullptr)
50          thenStmt->genCode();
51
52      // 获取生成的 then 语句的基本块
53      then_bb = builder->getInsertBB();
54      // 生成无条件跳转指令的中间代码
55      new UncondBrInstruction(end_bb, then_bb);
56
```

```
57      // 设置新基本块插入点并生成 else 语句的中间代码
58      builder->setInsertBB(else_bb);
59      if (elseStmt != nullptr)
60          elseStmt->genCode();
61
62      // 获取生成的 else 语句的基本块
63      else_bb = builder->getInsertBB();
64      // 生成无条件跳转指令的中间代码
65      new UncondBrInstruction(end_bb, else_bb);
66
67      // 设置插入点为 end_bb
68      builder->setInsertBB(end_bb);
69  }
```

- while

  while 要将 false_List 回填到 end_bb,true_List 回填到 loop_bb，在 end 分支中生成无条件跳转指令的中间代码跳转到合并块。

```
1   // while(cond)
2   //      loop_bb
3   // end_bb
4
5   void WhileStmt::genCode()
6   {
7       // fprintf(stderr, "WhileStmt::genCode\n");
8       Function *func;
9       BasicBlock *cond_bb, *loop_bb, *end_bb, *bb;
10
11      // 获取当前基本块所属的函数
12      bb = builder->getInsertBB();
13      func = bb->getParent();
14
15      // 新建基本块
16      cond_bb = new BasicBlock(func);
17      loop_bb = new BasicBlock(func);
18      end_bb = new BasicBlock(func);
19
20      // 将当前基本块添加为 cond_bb 的前驱, cond_bb 添加为当前基本块的后继
21      bb->addSucc(cond_bb);
22      cond_bb->addPred(bb);
23
24      // 设置基本块插入点并生成条件表达式的中间代码
25      builder->setInsertBB(cond_bb);
26      cond->genCode();
27      // 生成无条件跳转指令的中间代码
28      new UncondBrInstruction(cond_bb, bb);
29
30      // 获取当前基本块
31      bb = builder->getInsertBB();
```

```
    Instruction *test = bb->rbegin();

    // 如果当前基本块的最后一条指令不是条件或无条件跳转指令
    if (!test->isCond() && !test->isUncond())
    {
        Operand *src1 = cond->getOperand();
        Operand *n = src1;

        // int 类型需要与 0 比较并保存为 bool 类型
        if (src1->getType() == TypeSystem::intType)
        {
            int opcode = CmpInstruction::NE;
            Operand *src2 = src0_const0;
            SymbolEntry *tse = new TemporarySymbolEntry(TypeSystem::boolType,
                SymbolTable::getLabel());
            Operand *dst = new Operand(tse);
            new CmpInstruction(opcode, dst, src1, src2, bb);
            n = dst;
        }

        // 生成条件跳转指令的中间代码
        Instruction *temp = new CondBrInstruction(nullptr, nullptr, n, bb);
        cond->trueList().push_back(temp);
        cond->falseList().push_back(temp);
    }

    // 回填
    backPatch(cond->trueList(), loop_bb, true);
    backPatch(cond->falseList(), end_bb, false);
    bb = builder->getInsertBB();

    // 设置新基本块插入点并生成循环体的中间代码
    builder->setInsertBB(loop_bb);
    if (stmt != nullptr)
        stmt->genCode();

    // 获取生成的循环体的基本块
    loop_bb = builder->getInsertBB();
    // 生成无条件跳转指令的中间代码
    new UncondBrInstruction(cond_bb, loop_bb);

    // 设置插入点为 end_bb
    builder->setInsertBB(end_bb);
}
```

- return

```
void ReturnStmt::genCode()
{
    // fprintf(stderr, "ReturnStmt::genCode\n");
```

```
    Operand *src = nullptr;
    // 如果有返回值表达式，生成其代码并获取操作数
    if (retValue != nullptr)
    {
        retValue->genCode();
        src = retValue->getOperand();
    }

    // 创建返回指令，将返回值操作数传递给返回指令，并设置插入的基本块
    new RetInstruction(src, builder->getInsertBB());
}
```

5.4.3 控制流分析

生成 IR 的时候，并没有确定基本块之间的先后关系。LLVM IR 要求在每个基本块的最后都是一条跳转指令或者返回指令，可以利用这一特性，对基本块之间进行控制流分析。

选取基本块的最后一条语句，确定其跳转的目标基本块，并在这两个基本块之间建立起前后关系，在本学期实验中，采用了控制流图的方式进行实现。

获取这个块的第一条和最后一条指令，遍历移除块中的跳转指令。如果最后一条指令是有条件跳转，获取真分支和假分支，并生成对应的返回指令，更新前驱和后继关系；如果是无条件跳转指令，获取要跳转的基本块，更新前驱和后继关系。如果最后一条指令不是返回以及跳转，如果函数的返回类型是 void 类型，则在当前块生成返回指令。

```
// 函数定义中间代码生成函数
// 生成函数的入口块、参数中间代码和函数体中间代码、进行流图构造
void FunctionDef::genCode()
{
    // fprintf(stderr, "FunctionDef::genCode\n");
    // 获取当前函数的单元（IRBuilder）
    Unit *unit = builder->getUnit();
    // 创建一个新的函数对象，使用函数的符号表项（SymbolEntry）和当前单元
    Function *func = new Function(unit, se);
    // 获取函数的入口基本块
    BasicBlock *entry = func->getEntry();

    // 设置当前的插入基本块为函数的入口基本块
    builder->setInsertBB(entry);

    // 函数参数中间代码生成
    if (FuncDefParams)
        FuncDefParams->genCode();

    // 函数体中间代码生成
    if (stmt)
        stmt->genCode();

    // 根据基本块的前驱、后继关系进行流图的构造
    for (auto block = func->begin(); block != func->end(); block++)
```

```
26      {
27          // 获取该块的第一条和最后一条指令
28          Instruction *i = (*block)->begin();
29          Instruction *last = (*block)->rbegin();
30
31          // 遍历块中的指令，移除块中的跳转指令
32          while (i != last)
33          {
34              if (i->isCond() || i->isUncond())// 如果指令是条件跳转或无条件跳转
35              {
36                  // 移除块中跳转指令
37                  (*block)->remove(i);
38              }
39              i = i->getNext();
40          }
41
42          // 处理有条件跳转
43          if (last->isCond())
44          {
45              BasicBlock *truebranch, *falsebranch;
46              // 获取有条件跳转的真分支和假分支
47              truebranch = dynamic_cast<CondBrInstruction *>(last)->getTrueBranch();
48              falsebranch = dynamic_cast<CondBrInstruction *>(last)->getFalseBranch();
49
50              // 处理空块
51              if (truebranch->empty())
52              {
53                  // 在真分支上生成返回指令
54                  new RetInstruction(nullptr, truebranch);
55              }
56              else if (falsebranch->empty())
57              {
58                  // 在假分支上生成返回指令
59                  new RetInstruction(nullptr, falsebranch);
60              }
61              // 更新前驱和后继关系
62              (*block)->addSucc(truebranch);
63              (*block)->addSucc(falsebranch);
64              truebranch->addPred(*block);
65              falsebranch->addPred(*block);
66          }
67          // 处理无条件跳转
68          else if (last->isUncond())
69          {
70              // 获取要跳转的基本块
71              BasicBlock *dst = dynamic_cast<UncondBrInstruction *>(last)->getBranch();
72
73              // 更新前驱和后继关系
74              (*block)->addSucc(dst);
75              dst->addPred(*block);
```

```
76
77              // 处理空块
78              if (dst->empty())
79              {
80                  // 根据函数的返回类型生成相应的返回指令
81                  if (((FunctionType *)(se->getType()))->getRetType() ==
                        TypeSystem::intType)
82                      new RetInstruction(new Operand(new
                            ConstantSymbolEntry(TypeSystem::intType, 0)), dst);
83                  else if (((FunctionType *)(se->getType()))->getRetType() ==
                        TypeSystem::voidType)
84                      new RetInstruction(nullptr, dst);
85              }
86          }
87          // 处理最后一条语句不是返回以及跳转
88          else if (!last->isRet())
89          {
90              // 如果函数的返回类型为 void, 则在当前块上生成返回指令
91              if (((FunctionType *)(se->getType()))->getRetType() ==
                    TypeSystem::voidType)
92              {
93                  new RetInstruction(nullptr, *block);
94              }
95          }
96      }
97  }
98
99  // 函数调用中间代码生成函数
100 // 生成函数调用指令
101 void FuncCallExpr::genCode()
102 {
103     // fprintf(stderr, "FuncCallExpr::genCode\n");
104     // 用于存储函数调用的参数
105     std::vector<Operand *> operands;
106
107     // 遍历参数表达式列表, 生成参数的中间代码
108     ExprNode *temp = param;
109     while (temp)
110     {
111         // 参数中间代码生成
112         temp->genCode();
113         operands.push_back(temp->getOperand());
114         temp = ((ExprNode *)temp->getNext());
115     }
116     // 获取当前插入基本块
117     BasicBlock *bb = builder->getInsertBB();
118
119     // 生成函数调用的中间代码
120     new CallInstruction(dst, symbolEntry, operands, bb);
121 }
```

6 目标代码生成

6.1 实验内容

目标代码生成阶段主要完成了基本的 IR 指令到汇编指令的翻译。实现基本的完善 genMachineCode() 函数，实现了数据访存指令的翻译 StoreInstruction；二元运算指令的翻译 BinaryInstruction；比较指令的翻译；控制流指令的翻译，UncondBrInstruction、CondBrInstruction、RetInstruction 等；函数定义及函数调用的翻译。此外还实现了进阶要求中数组的翻译、浮点类型的翻译、break 等语句的翻译、非叶函数的翻译等。完善汇编指令中的 output 函数，实现寄存器分配的的相关工作，完成物理寄存器的分配，并对于需要进行溢出的寄存器进行相关处理。

6.2 实验效果

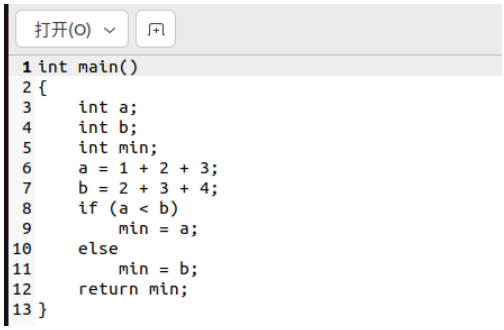


图 6.7: example.sy 截图

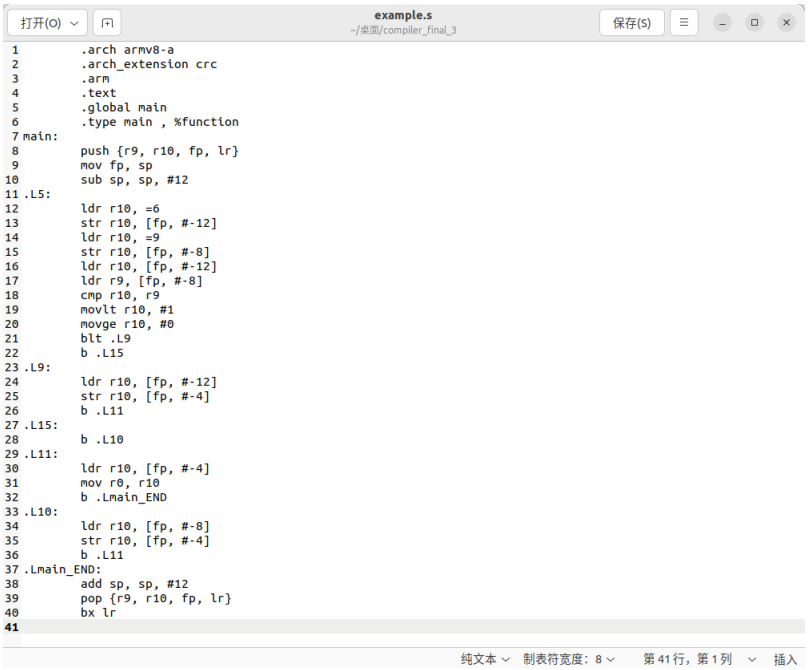


图 6.8: example.sy 目标代码生成后截图

6.3 代码设计

6.3.1 IR 指令到汇编指令的翻译

- 数据访存指令的翻译 StoreInstruction

  检查操作数 operands[1] 是否是变量(Variable)。如果是变量,进一步判断是否是参数(Parameter)。如果是参数，并且参数 ID 大于等于 4，就计算参数的偏移量（offset），并将该偏移量设置为目标操作数（operands[0]）的偏移量，然后直接返回，表示生成机器码的工作完成。

  如果 operands[1] 的类型是浮点型（Float），则生成浮点操作数（src），并检查是否是立即数。如果是立即数，生成临时寄存器 tmp_dst，然后加载指令将立即数加载到临时寄存器中，接着移动指令将数据从临时寄存器移动到新的浮点操作数 src 中。

  如果 operands[1] 的类型不是浮点型是整形，同样生成操作数 src，并检查是否是立即数。如果是立即数，生成加载指令将立即数加载到临时寄存器中，然后将数据从临时寄存器移动到新的操作数 src 中。对于整数，不需要额外的移动指令。

  如果目标操作数 operands[0] 是全局变量，生成加载地址指令，然后生成存储指令，将数据存储到全局变量地址；如果目标操作数 operands[0] 是临时变量，且该临时变量是通过分配（Allocation）而来的，生成相应的机器码指令，并计算偏移量和生成存储指令，将数据存储到相对于基地址的偏移位置；如果目标操作数 operands[0] 不是全局变量也不是临时变量，可能是从临时变量加载数据并存储到其他位置。生成相应的机器码指令，包括加载和存储指令。

```
void StoreInstruction::genMachineCode(AsmBuilder* builder)
{
    auto cur_block = builder->getBlock();
    MachineInstruction* cur_inst = nullptr;
    // 如果当前是存参数 则直接修改dst的offset
    // 这个地方 整型、浮点和数组逻辑都一致
    if(operands[1]->getEntry()->isVariable()) {
        auto id_se = dynamic_cast<IdentifierSymbolEntry*>(operands[1]->getEntry());
        if(id_se->isParam()) {
            int param_id = this->getParent()->getParent()->getParamId(operands[1]);
            if(param_id >= 4) {
                int offset = 4 * (param_id - 4);
                dynamic_cast<TemporarySymbolEntry*>(operands[0]->getEntry())->setOffset(offset
                return;
            }
        }
    }

    MachineOperand* src = nullptr;

    if(operands[1]->getType()->isFloat()) {
        src = genMachineOperand(operands[1], true);
        //如果src为常数, 需要先load进来
        if(src->isImm()){
            auto tmp_dst = genMachineVReg(true);
            auto internal_reg = genMachineVReg();
            cur_inst = new LoadMInstruction(cur_block, internal_reg, src);
            cur_block->InsertInst(cur_inst);
```

```
            internal_reg = new MachineOperand(*internal_reg);
            cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV, tmp_dst,
                internal_reg);
            cur_block->InsertInst(cur_inst);
            src = new MachineOperand(*tmp_dst);
        }
    }
    else {
        src = genMachineOperand(operands[1]);
        //如果src为常数, 需要先load进来
        if(src->isImm()){
            auto internal_reg = genMachineVReg();
            cur_inst = new LoadMInstruction(cur_block, internal_reg, src);
            cur_block->InsertInst(cur_inst);
            src = new MachineOperand(*internal_reg);
        }
    }
    // store global operand
    if(operands[0]->getEntry()->isVariable()
    && dynamic_cast<IdentifierSymbolEntry*>(operands[0]->getEntry())->isGlobal())
    {
        auto internal_reg1 = genMachineVReg();
        auto internal_reg2 = new MachineOperand(*internal_reg1);
        auto dst = genMachineOperand(operands[0]);
        if(operands[1]->getType()->isFloat()) {
            // example: load r0, addr_a
            cur_inst = new LoadMInstruction(cur_block, internal_reg1, dst);
            cur_block->InsertInst(cur_inst);
            // example: store r1, [r0]
            cur_inst = new StoreMInstruction(cur_block, src, internal_reg2, nullptr,
                StoreMInstruction::VSTR);
            cur_block->InsertInst(cur_inst);
        }
        else {
            // example: load r0, addr_a
            cur_inst = new LoadMInstruction(cur_block, internal_reg1, dst);
            cur_block->InsertInst(cur_inst);
            // example: store r1, [r0]
            cur_inst = new StoreMInstruction(cur_block, src, internal_reg2);
            cur_block->InsertInst(cur_inst);
        }
    }
    // store local operand
    else if(operands[0]->getEntry()->isTemporary()
    && operands[0]->getDef()
    && operands[0]->getDef()->isAlloc())
    {
        // example: store r1, [r0, #4]
        auto dst1 = genMachineReg(11);
        int offset =
```

```
            dynamic_cast<TemporarySymbolEntry*>(operands[0]->getEntry())->getOffset();
77          auto dst2 = genMachineImm(offset);
78          if(offset > 255 || offset < -255) {
79              auto internal_reg = genMachineVReg();
80              cur_inst = new LoadMInstruction(cur_block, internal_reg, dst2);
81              cur_block->InsertInst(cur_inst);
82              dst2 = new MachineOperand(*internal_reg);
83          }
84          if(operands[1]->getType()->isFloat()) {
85              // 对于浮点数 超过范围的 offset 需要再做处理
86              if(offset > 255 || offset < -255) {
87                  auto reg = genMachineVReg();
88                  cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::ADD,
                        reg, dst1, dst2);
89                  cur_block->InsertInst(cur_inst);
90                  dst2 = reg;
91              }
92              cur_inst = new StoreMInstruction(cur_block, src, dst1, dst2,
                    StoreMInstruction::VSTR);
93              cur_block->InsertInst(cur_inst);
94          }
95          else {
96              cur_inst = new StoreMInstruction(cur_block, src, dst1, dst2);
97              cur_block->InsertInst(cur_inst);
98          }
99      }
100     // store operand from temporary variable
101     else
102     {
103         // example: store r1, [r0]
104         if(operands[0]->getEntry()->getType()->isArray()){
105             //如果是全局数组访问, 不需要将offset与fp相加
106             //如果是函数参数传进来的, 同理
107             std::vector<int> dimensions;
108             if(operands[0]->getEntry()->getType()->isIntArray()){
109                 dimensions =
                        dynamic_cast<IntArrayType*>(operands[0]->getEntry()->getType())->getDimens
110             }
111             else if(operands[0]->getEntry()->getType()->isConstIntArray()){
112                 dimensions =
                        dynamic_cast<ConstIntArrayType*>(operands[0]->getEntry()->getType())->getD
113             }
114             else if(operands[0]->getEntry()->getType()->isFloatArray()){
115                 dimensions =
                        dynamic_cast<FloatArrayType*>(operands[0]->getEntry()->getType())->getDime
116             }
117             else{
118                 dimensions =
                        dynamic_cast<ConstFloatArrayType*>(operands[0]->getEntry()->getType())->ge
119             }
```

```
120             if(dynamic_cast<TemporarySymbolEntry*>(operands[0]->getEntry())->getGlobalArray()
                    ||
121                 dimensions[0]==-1){
122                 auto dst_addr = genMachineOperand(operands[0]);
123                 if(operands[1]->getType()->isFloat()) {
124                     cur_inst = new StoreMInstruction(cur_block, src, dst_addr,
                            nullptr, StoreMInstruction::VSTR);
125                     cur_block->InsertInst(cur_inst);
126                 }
127                 else {
128                     cur_inst = new StoreMInstruction(cur_block, src, dst_addr);
129                     cur_block->InsertInst(cur_inst);
130                 }
131             }
132             else{
133                 auto dst_addr = genMachineVReg();
134                 auto fp = genMachineReg(11);
135                 auto offset = genMachineOperand(operands[0]);
136                 if(offset->isImm()) {
137                     if(((ConstantSymbolEntry*)(operands[0]->getEntry()))->getValue()
                            > 255 ||
138                        ((ConstantSymbolEntry*)(operands[0]->getEntry()))->getValue()
                            < -255) {
139                         auto internal_reg = genMachineVReg();
140                         cur_inst = new LoadMInstruction(cur_block, internal_reg,
                                offset);
141                         cur_block->InsertInst(cur_inst);
142                         offset = new MachineOperand(*internal_reg);
143                     }
144                 }
145                 cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::ADD,
                        dst_addr, fp, offset);
146                 cur_block->InsertInst(cur_inst);
147 
148                 if(operands[1]->getType()->isFloat()) {
149                     cur_inst = new StoreMInstruction(cur_block, src, new
                            MachineOperand(*dst_addr), nullptr, StoreMInstruction::VSTR);
150                     cur_block->InsertInst(cur_inst);
151                 }
152                 else {
153                     cur_inst = new StoreMInstruction(cur_block, src, new
                            MachineOperand(*dst_addr));
154                     cur_block->InsertInst(cur_inst);
155                 }
156             }
157         }
158         else{
159             auto dst = genMachineOperand(operands[0]);
160             if(operands[1]->getType()->isFloat()) {
161                 cur_inst = new StoreMInstruction(cur_block, src, dst, nullptr,
```

```
                    StoreMInstruction::VSTR);
                cur_block->InsertInst(cur_inst);
            }
            else {
                cur_inst = new StoreMInstruction(cur_block, src, dst);
                cur_block->InsertInst(cur_inst);
            }
        }
    }
}
```

- 二元运算指令的翻译 BinaryInstruction

  调用 genMachineOperand 函数，生成目标操作数（dst）和两个源操作数（src1 和 src2）的机器码表示。对于 ADD 指令，源操作数可以是立即数，但在汇编代码中不允许。如果 src1 是立即数，就生成加载指令将立即数加载到临时寄存器中，并将 src1 替换为该寄存器；对于 MUL、DIV 和 MOD 指令，处理第二个源操作数 src2，如果它是立即数，则也生成加载指令。如果 src2 是立即数且其值大于 255 或小于 -255，同样生成加载指令，将立即数加载到临时寄存器中，并将 src2 替换为该寄存器。根据 opcode 进行 switch 分支判断，生成相应的二进制操作指令。需要注意的是对于 MOD 指令，要按照 a % b = a - a / b * b 的关系生成相应的机器码指令序列。

```
void BinaryInstruction::genMachineCode(AsmBuilder* builder)
{
    // complete other instructions
    auto cur_block = builder->getBlock();
    auto dst = genMachineOperand(operands[0]);
    auto src1 = genMachineOperand(operands[1]);
    auto src2 = genMachineOperand(operands[2]);
    /* HINT:
    * The source operands of ADD instruction in ir code both can be immediate num.
    * However, it's not allowed in assembly code.
    * So you need to insert LOAD/MOV instrucrion to load immediate num into register.
    * As to other instructions, such as MUL, CMP, you need to deal with this
        situation, too.*/
    MachineInstruction* cur_inst = nullptr;
    if(src1->isImm())
    {
        auto internal_reg = genMachineVReg();
        cur_inst = new LoadMInstruction(cur_block, internal_reg, src1);
        cur_block->InsertInst(cur_inst);
        src1 = new MachineOperand(*internal_reg);
    }
    if(opcode == MUL || opcode == DIV || opcode == MOD)
    {
        if(src2->isImm())
        {
            auto internal_reg = genMachineVReg();
            cur_inst = new LoadMInstruction(cur_block, internal_reg, src2);
            cur_block->InsertInst(cur_inst);
```

```
            src2 = new MachineOperand(*internal_reg);
        }
    }
    if(src2->isImm() && (src2->getVal() > 255 || src2->getVal() < -255)) {
        auto internal_reg = genMachineVReg();
        cur_inst = new LoadMInstruction(cur_block, internal_reg, src2);
        cur_block->InsertInst(cur_inst);
        src2 = new MachineOperand(*internal_reg);
    }
    switch (opcode)
    {
    case ADD:
        cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::ADD, dst,
            src1, src2);
        break;
    case SUB:
        cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::SUB, dst,
            src1, src2);
        break;
    case MUL:
        cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::MUL, dst,
            src1, src2);
        break;
    case DIV:
        cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::DIV, dst,
            src1, src2);
        break;
    case MOD: {
        // a % b = a - a / b * b
        cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::DIV, dst,
            src1, src2);
        MachineOperand *dst1 = new MachineOperand(*dst);
        src1 = new MachineOperand(*src1);
        src2 = new MachineOperand(*src2);
        cur_block->InsertInst(cur_inst);
        cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::MUL, dst1,
            dst, src2);
        cur_block->InsertInst(cur_inst);
        dst = new MachineOperand(*dst1);
        cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::SUB, dst,
            src1, dst1);
        break;
    }
    case AND:
        cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::AND, dst,
            src1, src2);
        break;
    case OR:
        cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::OR, dst,
            src1, src2);
```

```
69          break;
70      default:
71          break;
72      }
73      cur_block->InsertInst(cur_inst);
74  }
```

对于浮点数的二元运算指令翻译，与整数的二元运算指令翻译大致相同。对于 src1 和 src2，需要临时寄存器 tmp_reg，然后加载指令 MovMInstruction::VMOV 加载到 internal_reg 中，并将 src1 替换为 tmp_reg。

- 比较指令的翻译

  调用整数的 genMachineOperand 函数，生成两个源操作数（src1 和 src2）的机器码表示。如果 src1 是立即数，生成加载指令将立即数加载到临时寄存器中，并将 src1 替换为该寄存器。同样处理第二个源操作数 src2。生成比较指令，并插入到当前基本块，然后设置当前基本块的条件分支条件（setCurrentBranchCond），用于后续的条件分支跳转。生成目标操作数 dst、真值操作数 trueOperand 和假值操作数 falseOperand 的机器码表示。这里采用条件存储的方式将 1 或 0 存储到目标操作数 dst 中。根据比较指令的操作码，生成相应的条件移动指令插入当前基本块中。

  插入相应的条件移动指令到当前基本块中，实现条件存储。

```
1   void CmpInstruction::genMachineCode(AsmBuilder* builder)
2   {
3       MachineBlock* cur_block = builder->getBlock();
4       MachineOperand* src1 = genMachineOperand(operands[1]);
5       MachineOperand* src2 = genMachineOperand(operands[2]);
6       MachineInstruction* cur_inst = nullptr;
7       if (src1->isImm()) {
8           MachineOperand* internal_reg = genMachineVReg();
9           cur_inst = new LoadMInstruction(cur_block, internal_reg, src1);
10          cur_block->InsertInst(cur_inst);
11          src1 = new MachineOperand(*internal_reg);
12      }
13      if (src2->isImm()) {
14          MachineOperand* internal_reg = genMachineVReg();
15          cur_inst = new LoadMInstruction(cur_block, internal_reg, src2);
16          cur_block->InsertInst(cur_inst);
17          src2 = new MachineOperand(*internal_reg);
18      }
19      cur_inst = new CmpMInstruction(cur_block, src1, src2, opcode);
20      cur_block->InsertInst(cur_inst);
21      cur_block->setCurrentBranchCond(opcode);
22      // 采用条件存储的方式将1/0存储到dst中
23      MachineOperand* dst = genMachineOperand(operands[0]);
24      MachineOperand* trueOperand = genMachineImm(1);
25      MachineOperand* falseOperand = genMachineImm(0);
26      cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst, trueOperand,
            opcode);
27      cur_block->InsertInst(cur_inst);
```

```
    if(opcode == CmpInstruction::E || opcode == CmpInstruction::NE){
        cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
            falseOperand, 1-opcode);
    }
    else {
        cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
            falseOperand, 7-opcode);
    }
    cur_block->InsertInst(cur_inst);
}
```

对于浮点数的比较指令翻译,FCmpInstruction::genMachineCode 与 CmpInstruction::genMachineCode 大致相同, 但还是略有区别。对于 src1 和 src2, 需要临时寄存器 tmp_reg, 然后加载指令 MovMInstruction::VMOV 加载到 internal_reg 中。通过 VmrsMInstruction 指令获取浮点数比较的结果, 然后通过条件存储方式将 1 或 0 存储到目标寄存器中。

```
void FCmpInstruction::genMachineCode(AsmBuilder* builder)
{
    MachineBlock* cur_block = builder->getBlock();
    MachineOperand* src1 = genMachineOperand(operands[1], true);
    MachineOperand* src2 = genMachineOperand(operands[2], true);
    MachineInstruction* cur_inst = nullptr;
    if (src1->isImm()) {
        MachineOperand* tmp_reg = genMachineVReg(true);
        MachineOperand* internal_reg = genMachineVReg();
        cur_inst = new LoadMInstruction(cur_block, internal_reg, src1);
        cur_block->InsertInst(cur_inst);
        internal_reg = new MachineOperand(*internal_reg);
        cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV,tmp_reg,
            internal_reg);
        cur_block->InsertInst(cur_inst);
        src1 = new MachineOperand(*tmp_reg);
    }
    if (src2->isImm()) {
        MachineOperand* tmp_reg = genMachineVReg(true);
        MachineOperand* internal_reg = genMachineVReg();
        cur_inst = new LoadMInstruction(cur_block, internal_reg, src2);
        cur_block->InsertInst(cur_inst);
        internal_reg = new MachineOperand(*internal_reg);
        cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV,tmp_reg,
            internal_reg);
        cur_block->InsertInst(cur_inst);
        src2 = new MachineOperand(*tmp_reg);
    }
    cur_inst = new CmpMInstruction(cur_block, src1, src2, opcode,
        CmpMInstruction::VCMP);
    cur_block->InsertInst(cur_inst);
    cur_inst = new VmrsMInstruction(cur_block);
    cur_block->InsertInst(cur_inst);
```

```
    cur_block->setCurrentBranchCond(opcode);
    // 采用条件存储的方式将1/0存储到dst中
    MachineOperand* dst = genMachineOperand(operands[0]);
    MachineOperand* trueOperand = genMachineImm(1);
    MachineOperand* falseOperand = genMachineImm(0);
    cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst, trueOperand,
        opcode);
    cur_block->InsertInst(cur_inst);
    if(opcode == CmpInstruction::E || opcode == CmpInstruction::NE){
        cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
            falseOperand, 1-opcode);
    }
    else {
        cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
            falseOperand, 7-opcode);
    }
    cur_block->InsertInst(cur_inst);
}
```

- 控制流指令的翻译 UncondBrInstruction、CondBrInstruction、RetInstruction

  无条件跳转指令的翻译要构建目标标签的字符串,并加上当前基本块的编号。生成目标操作数 dst,该操作数表示跳转目标的标签。创建无条件跳转指令，指令类型为无条件跳转（B），然后还需将该指令插入到当前基本块中。

  条件跳转指令的翻译要分别构建真分支和假分支的标签字符串，并加上真分支和假分支的编号。生成真分支和假分支的目标操作数 true_dst 和 false_dst，并创建有条件跳转指令，指令类型为条件跳转（B），操作数为 true_dst，条件码为当前基本块的条件分支条件。符合当前块跳转条件的跳转到真分支，不符合当前块跳转条件的跳转到假分支。

  返回指令的翻译创建机器指令的指针 cur_inst。如果返回值非空，生成相应的机器指令，根据返回值类型将返回值保存到寄存器 r0（整数）或浮点寄存器 s16（浮点数）中。构建跳转到函数结尾的标签字符串，并生成目标操作数 dst。然后生成一条跳转到结尾函数栈帧处理的无条件跳转语句，插入当前基本块中。

```
void UncondBrInstruction::genMachineCode(AsmBuilder* builder)
{
    MachineBlock* cur_block = builder->getBlock();
    std::stringstream label;
    label << ".L" << branch->getNo();
    MachineOperand* dst = new MachineOperand(label.str());
    MachineInstruction* cur_inst = new BranchMInstruction(cur_block,
        BranchMInstruction::B, dst);
    cur_block->InsertInst(cur_inst);
}

void CondBrInstruction::genMachineCode(AsmBuilder* builder)
{
    MachineBlock* cur_block = builder->getBlock();
    std::stringstream true_label, false_label;
```

```
15      true_label << ".L" << true_branch->getNo();
16      false_label << ".L" << false_branch->getNo();
17      MachineOperand* true_dst = new MachineOperand(true_label.str());
18      MachineOperand* false_dst = new MachineOperand(false_label.str());
19      // 符合当前块跳转条件有条件跳转到真分支
20      MachineInstruction* cur_inst = new BranchMInstruction(cur_block,
            BranchMInstruction::B, true_dst, cur_block->getCurrentBranchCond());
21      cur_block->InsertInst(cur_inst);
22      // 不符合当前块跳转条件无条件跳转到假分支
23      cur_inst = new BranchMInstruction(cur_block, BranchMInstruction::B, false_dst);
24      cur_block->InsertInst(cur_inst);
25  }
26
27  void RetInstruction::genMachineCode(AsmBuilder* builder)
28  {
29      auto cur_block = builder->getBlock();
30      MachineInstruction* cur_inst = nullptr;
31      //1. Generate mov instruction to save return value in r0
32      if(!operands.empty()){
33          if (operands[0]->getType()->isFloat()) {
34              auto src = genMachineOperand(operands[0], true);
35              if(src->isImm()) {
36                  auto internal_reg = genMachineVReg();
37                  cur_inst = new LoadMInstruction(cur_block, internal_reg, src);
38                  cur_block->InsertInst(cur_inst);
39                  src = internal_reg;
40              }
41              auto dst = new MachineOperand(MachineOperand::REG, 16, true);
42              cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV, dst,
                    src);
43              cur_block->InsertInst(cur_inst);
44          }
45          else {
46              auto src = genMachineOperand(operands[0]);
47              //立即数->寄存器
48              if(src->isImm())
49              {
50                  auto internal_reg = genMachineVReg();
51                  cur_inst = new LoadMInstruction(cur_block, internal_reg, src);
52                  cur_block->InsertInst(cur_inst);
53                  src = new MachineOperand(*internal_reg);
54              }
55              auto dst = new MachineOperand(MachineOperand::REG, 0);//r0
56              cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst, src);
57              cur_block->InsertInst(cur_inst);
58          }
59      }
60      // 生成一条跳转到结尾函数栈帧处理的无条件跳转语句
61      auto dst = new MachineOperand(".L" +
            this->getParent()->getParent()->getSymPtr()->toStr().erase(0,1) + "_END");
```

```
62      cur_inst = new BranchMInstruction(cur_block, BranchMInstruction::B, dst);
63      cur_block->InsertInst(cur_inst);
64      //接下来的工作放到MachineCode.cpp: void MachineFunction::output()完成
65  }
```

- 函数定义及函数调用的翻译

  遍历函数调用指令的操作数，判断参数的类型，分别统计整数参数个数 iparam_cnt 和浮点数参数个数 fparam_cnt。对于每个参数，根据参数类型生成相应的机器指令：如果是整数类型，将参数存储到 r0-r3 寄存器中，前四个参数通过寄存器传递，超过四个则通过栈传递；如果是浮点数类型，将参数存储到 s0-s3 浮点寄存器中，前四个参数通过浮点寄存器传递，超过四个通过栈传递。如果传参时既有整数又有浮点数的情况，整数采用整数寄存器，浮点数采用浮点数寄存器。

  对于某一类参数数量超过 4 个的，需要采用压栈的方式传参，压栈的顺序应该按照传参的逆序进行压栈，无论是整数函数浮点数传参，都是压入的同一个栈中，只不过使用的指令不同。在生成中间代码的时候，对于函数参数，都先开辟了栈帧，然后将参数 store 到栈中对应的位置。对于小于 4 个的传参，只需要将 store 的源操作数和参数寄存器对应起来即可，而对于超过 4 个的参数，则需要将 store 的源操作数和栈帧中的数据对应起来。

  在函数调用的时候，统计一下需要按照整型传参和按照浮点型传参的参数数量，然后倒序遍历传参，并依次递减对应的计数器，计数器大于 3 时采用压栈处理，小于等于 3 时使用相对应的参数寄存器。

  接下来确定通过压栈传参的参数在栈中相对于 fp 的偏移量。同样，在函数最开始的部分，通过设置 operand 的 offset 将参数保存到栈中的某个位置。对于压栈传参的参数，只需要修改其 operand 的 offset 为参数在栈中的位置即可。由于压栈的顺序和传参的顺序相反，因此函数参数的位置顺序减去 3 再乘 4 就是相对于 fp 的偏移。将压栈传参对应的 operand 保存下来，在最后进行 output 的时候，再根据已经确定的需要压栈保存的寄存器的数量，更新 offset。如果采用了压栈传参的方式，还需要在完成函数调用之后，恢复栈帧，即需要记录通过压栈传参的参数的数量，在完成函数调用之后，将 sp 加上压栈传参数量乘 4 的量。

  生成 BL 指令调用目标函数。函数名从 funcSE->getName() 中获取。如果被调用的函数有返回值，生成相应的机器指令将返回值移动到目标操作数。如果存在保存的寄存器，生成指令恢复栈帧，调整栈指针 sp。

```
1   void CallInstruction::genMachineCode(AsmBuilder* builder)
2   {
3       int saved_reg_cnt = 0;
4       auto cur_block = builder->getBlock();
5       MachineInstruction* cur_inst = nullptr;
6       std::vector<MachineOperand*> additional_args;
7       // for(unsigned int i = 1;i < operands.size();i++){
8       int iparam_cnt = 0;
9       int fparam_cnt = 0;
10      for(int i = 1; i < int(operands.size()); i++) {
11          if(operands[i]->getType()->isInt()) {
12              iparam_cnt++;
13          }
14          else if(operands[i]->getType()->isFloat()) {
```

```
15              fparam_cnt++;
16          }
17          else if(operands[i]->getType()->isArray()) {
18              bool isPointer = false;
19              bool is_float = false;
20              if(operands[i]->getEntry()->getType()->isIntArray()){
21                  isPointer =
                        dynamic_cast<IntArrayType*>(operands[i]->getEntry()->getType())->getPointe
22                  is_float = false;
23              }
24              else if(operands[i]->getEntry()->getType()->isFloatArray()) {
25                  isPointer = dynamic_cast<FloatArrayType
                        *>(operands[i]->getEntry()->getType())->getPointer();
26                  is_float = true;
27              }
28              else if(operands[i]->getEntry()->getType()->isConstIntArray()) {
29                  isPointer = dynamic_cast<ConstIntArrayType
                        *>(operands[i]->getEntry()->getType())->getPointer();
30                  is_float = false;
31              }
32              else if(operands[i]->getEntry()->getType()->isConstFloatArray()) {
33                  isPointer = dynamic_cast<ConstFloatArrayType
                        *>(operands[i]->getEntry()->getType())->getPointer();
34                  is_float = true;
35              }
36              if(isPointer) {
37                  iparam_cnt++;
38              }
39              else {
40                  if(is_float){
41                      fparam_cnt++;
42                  }
43                  else {
44                      iparam_cnt++;
45                  }
46              }
47          }
48      }
49      for(unsigned int i = operands.size() - 1; i > 0; i--){
50          //需要保证不是值而是数组指针
51          bool isPointer = false;
52          //如果类型是数组，需要考虑局部数组指针的情况
53          if(operands[i]->getEntry()->getType()->isArray()) {
54              if (operands[i]->getEntry()->getType()->isIntArray()) {
55                  isPointer = dynamic_cast<IntArrayType
                        *>(operands[i]->getEntry()->getType())->getPointer();
56                  //
                        dynamic_cast<IntArrayType*>(operands[i]->getEntry()->getType())->setPointe
57                  //如果第一维为-1，表明其为指针，传参时需要注意不加fp
58                  if (dynamic_cast<IntArrayType
```

```
                    *>(operands[i]->getEntry()->getType())->getDimensions()[0] == -1)
                    {
59                      isPointer = false;
60                  }
61              } else if (operands[i]->getEntry()->getType()->isFloatArray()) {
62                  isPointer = dynamic_cast<FloatArrayType
                        *>(operands[i]->getEntry()->getType())->getPointer();
63                  if (dynamic_cast<FloatArrayType
                        *>(operands[i]->getEntry()->getType())->getDimensions()[0] == -1)
                        {
64                      isPointer = false;
65                  }
66              } else if (operands[i]->getEntry()->getType()->isConstIntArray()) {
67                  isPointer = dynamic_cast<ConstIntArrayType
                        *>(operands[i]->getEntry()->getType())->getPointer();
68                  if (dynamic_cast<ConstIntArrayType
                        *>(operands[i]->getEntry()->getType())->getDimensions()[0] == -1)
                        {
69                      isPointer = false;
70                  }
71              } else if (operands[i]->getEntry()->getType()->isConstFloatArray()) {
72                  isPointer = dynamic_cast<ConstFloatArrayType
                        *>(operands[i]->getEntry()->getType())->getPointer();
73                  if (dynamic_cast<ConstFloatArrayType
                        *>(operands[i]->getEntry()->getType())->getDimensions()[0] == -1)
                        {
74                      isPointer = false;
75                  }
76              }
77          }
78          // 表示传入的是一个数组并且是指针
79          if(isPointer){
80              --iparam_cnt;
81              MachineOperand* dst_addr = nullptr;
82              // 情况1 必须保证是局部数组, 而且不是传进来的参数 此时需要加fp
83              if(!dynamic_cast<TemporarySymbolEntry*>(operands[i]->getEntry())->getGlobalArray()
                    {
84                  auto fp = genMachineReg(11);
85                  auto offset = genMachineOperand(operands[i]);
86                  if(offset->isImm()) {
87                      if(((ConstantSymbolEntry*)(operands[i]->getEntry()))->getValue()
                            > 255 ||
88                         ((ConstantSymbolEntry*)(operands[i]->getEntry()))->getValue()
                              < -255) {
89                          auto internal_reg = genMachineVReg();
90                          cur_inst = new LoadMInstruction(cur_block, internal_reg,
                                offset);
91                          cur_block->InsertInst(cur_inst);
92                          offset = new MachineOperand(*internal_reg);
93                      }
```

```
94                  }
95                  dst_addr = genMachineVReg();
96                  cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::ADD,
                        dst_addr, fp, offset);
97                  cur_block->InsertInst(cur_inst);
98              }
99              else {
100                 dst_addr = genMachineOperand(operands[i]);
101             }
102             // 全局数组或传入的数组指针参数，此时一律按int处理，但不需要加fp
103             // 而对于局部数组需要添加fp的已经在上面处理完
104             // 左起前4个参数通过r0-r3传递
105             if(iparam_cnt < 4){
106                 auto dst = new MachineOperand(MachineOperand::REG, iparam_cnt);//r0-r3
107                 cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
                        dst_addr);
108                 cur_block->InsertInst(cur_inst);
109             }
110             else{
111                 additional_args.clear();
112                 additional_args.push_back(dst_addr);
113                 cur_inst = new StackMInstruction(cur_block, StackMInstruction::PUSH,
                        additional_args);
114                 cur_block->InsertInst(cur_inst);
115                 saved_reg_cnt++;
116             }
117         }
118         else{
119             if(operands[i]->getType()->isFloat()) {
120                 --fparam_cnt;
121                 //左起前4个参数通过s0-s3传递
122                 if(fparam_cnt < 4){
123                     auto dst = new MachineOperand(MachineOperand::REG, fparam_cnt +
                            16, true);
124                     auto src = genMachineOperand(operands[i], true);
125                     if(src->isImm()) {
126                         auto internal_reg = genMachineVReg();
127                         cur_inst = new LoadMInstruction(cur_block, internal_reg, src);
128                         cur_block->InsertInst(cur_inst);
129                         internal_reg = new MachineOperand(*internal_reg);
130                         cur_inst = new MovMInstruction(cur_block,
                                MovMInstruction::VMOV, dst, internal_reg);
131                         cur_block->InsertInst(cur_inst);
132                     }
133                     else {
134                         cur_inst = new MovMInstruction(cur_block,
                                MovMInstruction::VMOV, dst, src);
135                         cur_block->InsertInst(cur_inst);
136                     }
137                 }
```

```
138                 else{
139                     additional_args.clear();
140                     MachineOperand* operand = genMachineOperand(operands[i], true);
141                     if(operand->isImm()) {
142                         MachineOperand* internal_reg = genMachineVReg();
143                         cur_inst = new LoadMInstruction(cur_block, internal_reg,
                                operand);
144                         cur_block->InsertInst(cur_inst);
145                         operand = genMachineVReg(true);
146                         cur_inst = new MovMInstruction(cur_block,
                                MovMInstruction::VMOV, operand, internal_reg);
147                         cur_block->InsertInst(cur_inst);
148                     }
149                     additional_args.push_back(operand);
150                     cur_inst = new StackMInstruction(cur_block,
                            StackMInstruction::VPUSH, additional_args);
151                     cur_block->InsertInst(cur_inst);
152                     saved_reg_cnt++;
153                 }
154             }
155             else {
156                 --iparam_cnt;
157                 //左起前4个参数通过r0-r3传递
158                 if(iparam_cnt < 4){
159                     auto dst = new MachineOperand(MachineOperand::REG,
                            iparam_cnt);//r0-r3
160                     cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV,
                            dst, genMachineOperand(operands[i]));
161                     cur_block->InsertInst(cur_inst);
162                 }
163                 else{
164                     additional_args.clear();
165                     MachineOperand* operand = genMachineOperand(operands[i]);
166                     if(operand->isImm()) {
167                         MachineOperand* internal_reg = genMachineVReg();
168                         cur_inst = new LoadMInstruction(cur_block, internal_reg,
                                operand);
169                         cur_block->InsertInst(cur_inst);
170                         operand = new MachineOperand(*internal_reg);
171                     }
172                     additional_args.push_back(operand);
173                     cur_inst = new StackMInstruction(cur_block,
                            StackMInstruction::PUSH, additional_args);
174                     cur_block->InsertInst(cur_inst);
175                     saved_reg_cnt++;
176                 }
177             }
178         }
179     }
180     cur_inst = new BranchMInstruction(cur_block, BranchMInstruction::BL, new
```

```
        MachineOperand(funcSE->getName(), true));
    cur_block->InsertInst(cur_inst);
    // 对于有返回值的函数调用 需要提供一条从mov r0, dst的指令
    if(dynamic_cast<FunctionType*>(this->funcSE->getType())->getRetType() ==
        TypeSystem::intType) {
        auto dst = genMachineOperand(operands[0]);
        auto src = new MachineOperand(MachineOperand::REG, 0);//r0
        cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst, src);
        cur_block->InsertInst(cur_inst);
    }
    else if(dynamic_cast<FunctionType*>(this->funcSE->getType())->getRetType() ==
        TypeSystem::floatType) {
        auto dst = genMachineOperand(operands[0], true);
        auto src = new MachineOperand(MachineOperand::REG, 16, true);//s0
        cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV, dst, src);
        cur_block->InsertInst(cur_inst);
    }
    // 恢复栈帧 调整sp
    if(saved_reg_cnt){
        auto src1 = genMachineReg(13);
        auto src2 = genMachineImm(saved_reg_cnt*4);
        if(saved_reg_cnt*4 > 255 || saved_reg_cnt*4 < -255) {
            auto internal_reg = genMachineVReg();
            cur_inst = new LoadMInstruction(cur_block, internal_reg, src2);
            cur_block->InsertInst(cur_inst);
            src2 = new MachineOperand(*internal_reg);
        }
        auto dst = genMachineReg(13);
        cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::ADD, dst,
            src1, src2);
        cur_block->InsertInst(cur_inst);
    }
}
```

- 浮点数

  ZextInstruction::genMachineCode 将一个整数进行零扩展，翻译零扩展指令，并将结果存储到目标寄存器中。如果源操作数是立即数，会将其加载到寄存器。

  IntFloatCastInstructionn::genMachineCode 实现整数到浮点数和浮点数到整数的类型转换的翻译。根据操作码 opcode 区分两种情况，分别处理从浮点数到整数和从整数到浮点数的转换。如果源操作数是立即数，会将其加载到相应的寄存器中。转换过程使用浮点数指令 VcvtMInstruction 和整数指令如 MovMInstruction 完成。

```
void ZextInstruction::genMachineCode(AsmBuilder* builder)
{
    MachineBlock* cur_block = builder->getBlock();
    MachineInstruction* cur_inst = nullptr;
    MachineOperand* src = genMachineOperand(operands[1]);
    if(src->isImm())
```

```
7       {
8           auto internal_reg = genMachineVReg();
9           cur_inst = new LoadMInstruction(cur_block, internal_reg, src);
10          cur_block->InsertInst(cur_inst);
11          src = new MachineOperand(*internal_reg);
12      }
13      MachineOperand* dst = genMachineOperand(operands[0]);
14      cur_inst = new ZextMInstruction(cur_block, dst, src);
15      cur_block->InsertInst(cur_inst);
16  }
17  void IntFloatCastInstructionn::genMachineCode(AsmBuilder* builder)
18  {
19      Operand* src = operands[1];
20      Operand* dst = operands[0];
21      if(opcode == F2I) {
22
23          MachineInstruction* cur_inst;
24          auto cur_block = builder->getBlock();
25
26          auto src_operand = genMachineOperand(src, true);
27          auto dst_operand = genMachineOperand(dst);
28
29          if (src_operand->isImm()) {
30              auto tmp = genMachineVReg(true);
31              auto internal_reg = genMachineVReg();
32              cur_inst = new LoadMInstruction(cur_block, internal_reg, src_operand);
33              cur_block->InsertInst(cur_inst);
34              internal_reg = new MachineOperand(*internal_reg);
35              cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV,
                    tmp,internal_reg);
36              cur_block->InsertInst(cur_inst);
37              src_operand = tmp;
38          }
39          auto vcvtDst = genMachineVReg(true);
40          cur_inst = new VcvtMInstruction(cur_block, VcvtMInstruction::F2S, vcvtDst,
                src_operand);
41          cur_block->InsertInst(cur_inst);
42          auto movUse = new MachineOperand(*vcvtDst);
43          cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV, dst_operand,
                movUse);
44          cur_block->InsertInst(cur_inst);
45      }
46      else {
47          MachineInstruction* cur_inst;
48          auto cur_block = builder->getBlock();
49
50          auto src_operand = genMachineOperand(src);
51          auto dst_operand = genMachineOperand(dst, true);
52
53          if (src_operand->isImm()) {
```

```
54                 auto tmp = genMachineVReg();
55                 cur_inst = new LoadMInstruction(cur_block, tmp, src_operand);
56                 cur_block->InsertInst(cur_inst);
57                 src_operand = new MachineOperand(*tmp);
58             }
59             auto movDst = genMachineVReg(true);
60             cur_inst = new MovMInstruction(cur_block, MovMInstruction::VMOV, movDst,
                   src_operand);
61             cur_block->InsertInst(cur_inst);
62             auto vcvtUse = new MachineOperand(*movDst);
63             cur_inst = new VcvtMInstruction(cur_block, VcvtMInstruction::S2F,
                   dst_operand, vcvtUse);
64             cur_block->InsertInst(cur_inst);
65         }
66     }
```

6.3.2 寄存器分配

寄存器分配是编译器的一个重要优化技术，通过将程序变量尽可能地分配到寄存器，从而提高程序执行速度。在本次实验中需要完成线性扫描寄存器分配算法，遍历每个活跃区间 (Interval)，为其分配物理寄存器。

在前一步的目标代码生成过程中，已经为所有临时变量分配了一个虚拟寄存器。在这一步需要为每个虚拟寄存器计算活跃区间，活跃区间相交的虚拟寄存器不能分配相同的物理寄存器。活跃区间的计算主要依赖活跃变量分析这一数据流分析方法，活跃变量分析的结果可以判断变量 x 在程序点 p 处是否活跃，变量 x 在点 p 处活跃指的是变量 x 在点 p 处的值在点 p 或点 p 之后仍然会被用到。变量 x 编号最小和最大的两个活跃点便是其活跃区间的端点。

本次实验中的寄存器分配需要完善 linearScanRegisterAllocation() 线性扫描寄存器函数。算法主要涉及到了两个集合：intervals 表示还未分配寄存器的活跃区间，其中所有的 interval 都按照开始位置进行递增排序；active 表示当前正在占用物理寄存器的活跃区间集合，其中所有的 interval 都按照结束位置进行递增排序。

算法遍历 intervals 列表，对遍历到的每一个活跃区间 i 都进行如下的处理：

1. 遍历 active 列表，看该列表中是否存在结束时间早于区间 i 开始时间的 interval（即与活跃区间 i 不冲突），若有，则说明此时为其分配的物理寄存器可以回收，可以用于后续的分配，需要将其在 active 列表删除；

2. 判断 active 列表中 interval 的数目和可用的物理寄存器数目是否相等，

(a) 若相等，则说明当前所有物理寄存器都被占用，需要进行寄存器溢出操作。具体为在 active 列表中最后一个 interval 和活跃区间 i 中选择一个 interval 将其溢出到栈中，选择策略就是看哪个活跃区间结束时间更晚，如果是活跃区间 i 的结时间更晚，只需要置位其 spill 标志位即可，如果是 active 列表中的活跃区间结束时间更晚，需要置位其 spill 标志位，并将其占用的寄存器分配给区间 i，再将区间 i 插入到 active 列表中。

(b) 若不相等，则说明当前有可用于分配的物理寄存器，为区间 i 分配物理寄存器之后，再按照活跃区间结束位置，将其插入到 active 列表中即可。

```
1 bool LinearScan::linearScanRegisterAllocation()
```

```
{
    bool retValue = true;// 初始化返回值
    active.clear();// 清空活跃列表
    regs.clear();
    fregs.clear();

    // 初始化整数寄存器列表，添加编号为4到10的整数寄存器
    for (int i = 4; i < 11; i++)
        regs.push_back(i);

    // 初始化浮点寄存器列表，添加编号为21到47的浮点寄存器
    for (int i = 21; i < 48; i++)
        fregs.push_back(i);

    // 遍历intervals列表，对遍历到的每一个活跃区间i都进行如下的处理
    for(auto &interval : intervals){
        // expireOldIntervals处理活跃列表（active）
        // 释放那些在当前活跃区间之前结束的活跃区间所占用的寄存器
        expireOldIntervals(interval);

        // 如果当前活跃区间需要浮点寄存器
        if(interval->freg){
            // 如果可分配浮点寄存器列表为空，说明当前所有物理寄存器都被占用，表示溢出
            if(fregs.size() == 0){
                spillAtInterval(interval);// spillAtInterval进行寄存器溢出操作
                retValue = false;// 设置返回值为 false，表示寄存器分配失败
            }
            // 若不相等
            else{//当前有可用于分配的物理寄存器
                interval->rreg = fregs[fregs.size()-1];//为 unhandled interval
                    区间i分配物理寄存器
                fregs.pop_back();// 移除已分配的浮点寄存器
                active.push_back(interval);//  将当前区间添加到活跃列表
                sort(active.begin(), active.end(), insertComp);//
                    按照区间结束位置递增的顺序对活跃列表进行排序
            }
        }

        // 如果当前活跃区间需要整数寄存器
        else{
            // 如果整数寄存器列表为空，说明当前所有物理寄存器都被占用，表示溢出
            if(regs.size() == 0){
                spillAtInterval(interval);// spillAtInterval进行寄存器溢出操作
                retValue = false;// 设置返回值为 false，表示寄存器分配失败
            }
            // 若不相等
            else{//当前有可用于分配的物理寄存器
                interval->rreg = regs[regs.size()-1];//为 unhandled interval
                    分配物理寄存器
                regs.pop_back();//  移除已分配的整数寄存器
```

```
49                  active.push_back(interval);//  将当前区间添加到活跃列表
50                  sort(active.begin(), active.end(), insertComp);//
                        按照区间结束位置递增的顺序对活跃列表进行排序
51              }
52          }
53      }
54      return retValue;// 返回寄存器分配是否成功的标志
55  }
56
57  // 处理活跃列表（active），释放那些在当前活跃区间之前结束的活跃区间所占用的寄存器。
58
59  void LinearScan::expireOldIntervals(Interval *interval)
60  {
61      for(std::vector<Interval*>::iterator it = active.begin(); it != active.end(); ){
62          // victimComp 用于确定可释放的active interval
63          // 看列表中是否存在结束时间早于区间i开始时间的interval
64          if(!victimComp(*it, interval)){
65              return;
66          }
67          //rreg
68          if ((*it)->rreg < 11) {
69              regs.push_back((*it)->rreg);//释放寄存器
70              it = active.erase(find(active.begin(), active.end(), *it));
71              sort(regs.begin(), regs.end());
72          }
73          //freg
74          else{
75              fregs.push_back((*it)->rreg);//释放寄存器
76              it = active.erase(find(active.begin(), active.end(), *it));
77              sort(fregs.begin(), fregs.end());
78          }
79      }
80  }
81
82  //
      处理溢出的情况，根据活跃区间的结束时间来判断是将当前活跃区间溢出，还是将活跃列表中的区间溢出。
83  void LinearScan::spillAtInterval(Interval *interval)
84  {
85      // 如果是活跃区间i的结束时间更晚，只需要置位其spill标志位即可
86      if(active[active.size()-1]->end <= interval->end){//unhandled interval
            的结束时间更晚
87          interval->spill = true;//只需要置位其 spill 标志位
88      }
89
90      else{// active 列表中的 interval 结束时间更晚
91          active[active.size()-1]->spill = true;//置位其spill标志位
92          interval->rreg = active[active.size()-1]->rreg;//将其占用的寄存器分配给
                unhandled interval
93          active.push_back(interval);// 将区间i插入到active列表中
94          sort(active.begin(), active.end(), insertComp);//
```

```
            按照区间结束位置递增的顺序对活跃列表进行排序
95      }
96  }
```

在上一步寄存器分配结束之后，如果没有临时变量被溢出到栈内，那寄存器分配的工作就结束了，所有的临时变量都被存在了寄存器中；若有，就需要在操作该临时变量时插入对应的 LoadMInstruction 和 StoreMInstruction，其起到的实际效果就是将该临时变量的活跃区间进行切分，以便重新进行寄存器分配。这一步需要完善 LinearScan::genSpillCode() 函数。

```
1   void LinearScan::allocateRegisters()
2   {
3       for (auto &f : unit->getFuncs())
4       {
5           func = f;
6           bool success;
7           success = false;
8           while (!success)        // repeat until all vregs can be mapped
9           {
10              computeLiveIntervals();// 计算活跃区间
11              success = linearScanRegisterAllocation();// 线性扫描寄存器分配算法
12              if (success)        // all vregs can be mapped to real regs
13                  modifyCode();
14              else                // spill vregs that can't be mapped to real regs
15                  genSpillCode();
16          }
17      }
18  }
```

具体分为以下三个步骤：

1. 为其在栈内分配空间，获取当前在栈内相对 FP 的偏移；

2. 遍历其 USE 指令的列表，在 USE 指令前插入 LoadMInstruction，将其从栈内加载到目前的虚寄存器中；

3. 遍历其 DEF 指令的列表，在 DEF 指令后插入 StoreMInstruction，将其从目前的虚拟寄存器中存到栈内；

```
1   void LinearScan::genSpillCode()
2   {
3       for(auto &interval:intervals)// 遍历所有活跃区间
4       {
5           // 如果当前区间不需要溢出（spill），则继续下一轮循环，跳过不需要处理的区间
6           if(!interval->spill)//
7               continue;
8           // TODO
9           /* HINT:
10           * The vreg should be spilled to memory.
11           * 1. insert ldr inst before the use of vreg
12           * 2. insert str inst after the def of vreg
13           */
14          // The vreg should be spilled to memory.
```

```
15          // 为当前需要溢出的区间分配栈空间, disp 记录了栈帧偏移 (正数表示向下生长的栈)
16          interval->disp = func->AllocSpace(4);//正数!
17          // 1. insert ldr inst before the use of vreg
18          // 将加载 (ldr) 指令插入到虚拟寄存器使用之前
19          //
                遍历其USE指令的列表, 在USE指令前插入LoadMInstruction, 将其从栈内加载到目前的虚拟寄存器
20          for (auto use : interval->uses){
21              // 获取使用点所在基本块
22              MachineBlock* block = use->getParent()->getParent();
23              // 计算栈帧偏移的负值, 作为加载指令的偏移
24              MachineOperand* offset = new MachineOperand(MachineOperand::IMM,
                    -interval->disp);
25              // 如果使用的是整数寄存器
26                  if(!use->isFloat()){
27                      // 在使用点之前插入加载指令, 将虚拟寄存器的值加载到寄存器11中
28                      block->insertBefore(use->getParent(), new LoadMInstruction(block,
                            new MachineOperand(*use), new
                            MachineOperand(MachineOperand::REG, 11), offset,
                            LoadMInstruction::LDR));
29                  }
30                  else{
31                      block->insertBefore(use->getParent(), new LoadMInstruction(block,
                            new MachineOperand(*use), new
                            MachineOperand(MachineOperand::REG, 11), offset,
                            LoadMInstruction::VLDR));
32                  }
33              // }
34          }
35          // 2. insert str inst after the def of vreg
                将存储 (str) 指令插入到虚拟寄存器定义之后
36          //
                遍历其DEF指令的列表, 在DEF指令后插入StoreMInstruction, 将其从目前的虚拟寄存器中存到栈内
37          for (auto def : interval->defs){
38              // 获取定义点所在基本块
39              MachineBlock* block = def->getParent()->getParent();
40              // 计算栈帧偏移的负值, 作为存储指令的偏移
41              MachineOperand* offset = new MachineOperand(MachineOperand::IMM,
                    -interval->disp);
42                  // 如果使用的是整数寄存器
43                  if(!def->isFloat()){
44                      // 在定义点之后插入存储指令, 将寄存器11中的值存储到虚拟寄存器中
45                      block->insertAfter(def->getParent(), new StoreMInstruction(block,
                            new MachineOperand(*def), new
                            MachineOperand(MachineOperand::REG, 11), offset,
                            StoreMInstruction::STR));
46                  }
47                  else{
48                      block->insertAfter(def->getParent(), new StoreMInstruction(block,
                            new MachineOperand(*def), new
                            MachineOperand(MachineOperand::REG, 11), offset,
```

```
                        StoreMInstruction::VSTR));
49                  }
50              }
51          }
52      }
```

插入结束后，会迭代进行以上过程，重新计算活跃区间，进行寄存器分配，直至没有溢出情况出现。

7 总结

为期一个学期的编译原理实验结束了，在这一个学期的理论学习和实验探究中，对于编译原理的算法理论有了清晰的认识，通过实验对于编译器各模块之间的工作流程和实现细节也有了一定的理解理解。通过一个学期的学习，最终实现了基本要求和进阶要求，通过了 146 个测试样例，在测试平台上除两个编译超时样例外其余全部 AC。

在实验过程中，由于对理论的认识不够透彻，导致在一些模块的的设计上出现了各种各样的问题，这也在后续的实验模块实现中带来了很大的困扰，但最后还是能够在不断的沟通协商和尝试下和各位助教学长学姐的帮助下得到解决。这也让我们在不断的试错中认识到工程各个模块的连续性。

最后，感谢这一个学期以来王刚老师尽职尽责的理论讲解，感谢各位助教学长学姐耐心解答，感谢队友在合作过程中为我提供的支持和帮助，基本实现了简单 SysY 编译器。