南開大學
Nankai University

计算机学院

编译原理期末实验报告

简易 SysY 语言编译器的实现

小组：NKU-compile error

成员：刘宇轩 赵健坤

学号：2012677 2010535

专业：计算机科学与技术

报告作者：刘宇轩

2023 年 1 月 14 日

摘要

本课程实验跟随编译原理理论学习，整整持续一个学期，其中包含了词法分析，语法分析，类型检查，中间代码生成和目标代码生成这五大模块。在课程给定的实验框架的基础上，结合理论课程学习，补充完善了简易 SysY 语言的编译器，除了支持基本的 SysY 语法特性之外，小组还完成了 break、continue 控制流，数组和浮点数等进阶语法特性。在整个实验的过程中，和赵健坤同学分工合作，我主要负责基础语法解析，控制流构建，浮点数支持以及部分数组语法的支持，而赵健坤同学主要负责了基本语法的中间代码生成，函数调用相关问题的处理以及数组整体前后端的主要框架。本实验是笔者和同伴在本学期投入时间最多的课程实验，在实验的过程中，对于理论的认识进一步深刻，也同样因为前期理论知识不足导致个别架构设计失误，使得在后期实现过程变得十分冗杂。经过笔者和同伴一个学期地不懈努力，最终通过了 145 个测试样例，可以说是经历了一次难能宝贵的工程实验经历。

关键词：词法分析；语法分析；类型检查；中间代码生成；目标代码生成。

目录

1 分工 **3**

2 词法分析 **5**

3 语法分析 **6**
3.1 上下文无关文法的设计 6
3.2 语法树构建 11
3.2.1 算数运算节点 11
3.2.2 变量定义 12
3.2.3 函数定义 13

4 类型检查 **14**
4.1 常量计算 14
4.2 类型转换 15
4.3 数组相关初始化 16

5 中间代码生成 **18**
5.1 短路支持 18
5.2 隐式类型转换支持 20
5.3 复杂控制流指令的生成 21
5.4 控制流分析 21
5.5 浮点指令支持 22

6 目标代码生成 **22**
6.1 控制流相关目标代码生成 23
6.2 全局数据目标代码生成 23
6.3 多参数函数传参处理 25
6.4 浮点指令目标代码生成 26

7 总结 **29**

1 分工

本次实验从整体架构上来说主要包含了 5 大部分：词法分析、语法分析、类型检查、中间代码生成和目标代码生成。编译器整体架构图如图1.1所示。

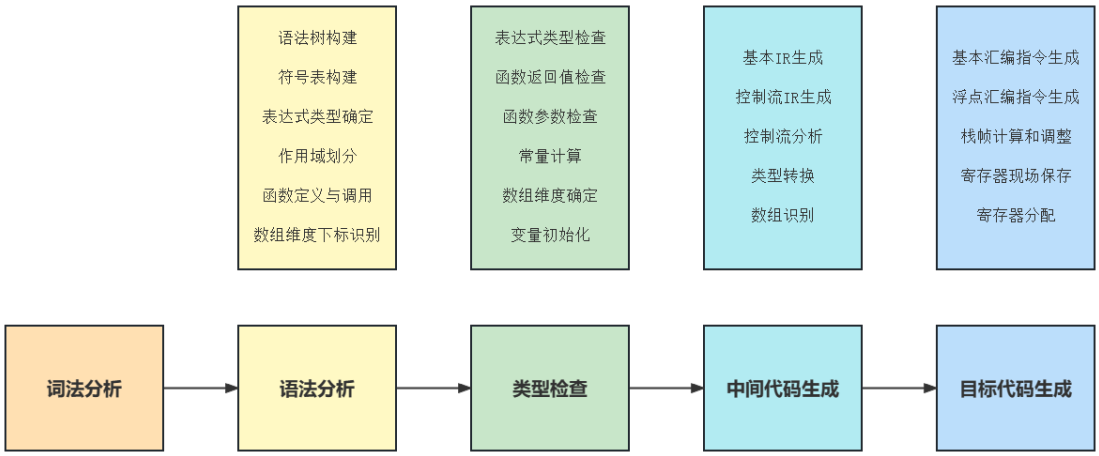

图 1.1: 编译器基本架构图

接下来简要列举各部分完成的工作：

1. **词法分析**

   在词法分析阶段，主要完成了关键字的识别，标识符的识别，符号识别和库函数的识别相关工作，对于关键字，标识符和符号标定为特定的 token，对于识别到的库函数，直接将其加入到全局符号表中。此外还采用正则表达式，对于各种进制表达的整数和浮点数进行识别。

2. **词法分析**

   在语法分析阶段，主要完成了语法树的构建，基于词法分析给出的 token 序列，构建语法树。此外，在这个阶段还根据作用域的层级关系，将识别到的标识符加入到对应层级的符号表中。在表达式运算的语法分析阶段，根据子表达式的类型初步确定父表达式的类型。对于变量的初始化赋值操作和数组的维度下表确定工作也在这个阶段完成。

3. **类型检查**

   在类型检查阶段，主要完成了常量检查，表达式类型确定，函数调用参数匹配，函数返回值匹配，数组下标维度匹配，常量计算和变量数组初始化赋值等相关操作。重点工作放在了常量计算上，即将子表达式均为常量的节点直接进行计算替换，降低语法树的复杂度。

4. **中间代码生成**

   在中间代码生成阶段，主要完成了基本 IR 的构建，在实现控制流 IR 的构建时，还额外实现了 break、continue 等控制流的构建。在完成了函数基本块的构建之后，根据 LLVM IR 的特性，对基本块进行控制流分析。除此之外，还完成了整型、浮点型和布尔类型之间的隐式类型转换工作。

5. **目标代码生成**

   在目标代码生成阶段，主要完成了基本汇编指令的生成，还引入了浮点汇编指令，并在整型和浮点运算之间加入了类型转换指令。在目标代码阶段，还需要对于局部变量开辟栈帧，完成栈帧的

分配工作。根据初步分配的临时寄存器，通过线性扫描的寄存器分配算法，完成物理寄存器的分配，并对于需要进行溢出的寄存器开辟栈帧。在函数调用的前后需要完成寄存器现场的保留和恢复工作。

接下来介绍一下本次实验过程中的小组分工情况，GitHub 不完全提交记录如图1.2所示：

1. **词法分析**

   在词法分析阶段，我主要完成了关键字、标识符和基本符号的识别工作，并且加入了多进制下整型的识别。赵健坤同学主要完成了库函数的识别并加入全局符号表，以及多进制下浮点数的识别。

2. **词法分析**

   在语法分析阶段，我主要完成了基本语法特性的上下文无关文法设计和语法树的构建工作，主要包含标识符的定义和初始化，函数的定义和调用，基本算术表达式，分支循环以及作用域。赵健坤同学主要完成了数组相关的上下文无关文法设计以及语法树的构建工作。

3. **类型检查**

   在类型检查阶段，我主要负责函数调用参数匹配，以及条件表达式中整型的处理。赵健坤同学主要负责了常量计算，函数返回值匹配，常量变量初始化工作。

4. **中间代码生成**

   在中间代码生成阶段，我主要负责控制流相关 IR 的生成，并且增加了浮点数相关的 IR，同时提供了整型、浮点型和布尔类型之间的类型转换指令，最后为整个函数提供控制流分析操作。赵健坤同学主要完成了基本 IR 指令的生成，以及函数定义和调用相关指令，并为数组提供基本指令。

5. **目标代码生成**

   在目标代码生成阶段，我主要负责完成了控制流相关的汇编代码生成，全局变量相关定义，函数传参过程中多余四个参数的处理，数组初始化值的确定，以及浮点相关指令的引入。赵健坤同学主要负责了基本汇编指令的生成，函数调用的栈帧调整，数组相关汇编代码的生成。

图 1.2: GitHub 提交记录统计

2 词法分析

词法分析应该是本次实验中最简单的模块，实验框架中给出的示例已经非常详细了，在此不做过多赘述，直接展示我们增加的正则表达式以及部分词法分析代码

词法分析示例代码

```
INTEGER ([1-9][0-9]*|0)
OCTAL (0[0-7][0-7]*)
HEXAL (0(x|X)[0-9a-fA-F][0-9a-fA-F]*)
FLOATING ((([0-9]*[.][0-9]*([eE][+-]?[0-9]+)?)|([0-9]+[eE][+-]?[0-9]+))[fL1L]?)
HEXADECIMAL_FLOAT (0[xX](([0-9A-Fa-f]*[.][0-9A-Fa-f]*([pP][+-]?[0-9]+)?)
                        |([0-9A-Fa-f]+[pP][+-]?[0-9]+))[fL1L]?)
ID [[:alpha:]_][[:alpha:][:digit:]_]*
EOL (\r\n|\n|\r)
WHITE [\t ]
COMMENT (\/\/[^\n]*)
commentbegin "/*"
commentelement .
commentline \n
commentend "*/"
%x COMMENT
%x COMBLOCK

%%
{commentbegin} {BEGIN COMMENT;}
<COMMENT>{commentelement} {}
<COMMENT>{commentline} {yylineno++;}
<COMMENT>{commentend} {BEGIN INITIAL;}

"if" {
    #ifdef ONLY_FOR_LEX
        DEBUG_FOR_LAB4("IF", "if");
        offset+=strlen(yytext);
    #else
        return IF;
    #endif
}

"getint" {
    #ifdef ONLY_FOR_LEX
        DEBUG_FOR_LAB4("ID", "getint");
        offset+=strlen(yytext);
    #else
        char *lexeme = new char[strlen(yytext) + 1];
        strcpy(lexeme, yytext);
        yylval.strtype = lexeme;
        if(identifiers->lookup(yytext)==nullptr){//符号表内未找到，插入
            Type* funcType = new FunctionType(TypeSystem::intType,
                {});//返回类型int，无参数
```

```
            SymbolTable* globalTable;     //全域符号表
            for(globalTable = identifiers;globalTable->getPrev();globalTable =
                globalTable->getPrev()); //全域符号表
            SymbolEntry* entry = new IdentifierSymbolEntry(funcType, yytext,
                0);//作用域GLOBAL(0)
            globalTable->install(yytext, entry);
        }
        return ID;
    #endif
}
```

3 语法分析

3.1 上下文无关文法的设计

要进行语法分析，首先要针对 SysY 语法特性设计对应的上下文无关文法，我们根据实验指导书给出的示例，以及参考资料中给出的 SysY 语言特性，设计了如下上下文无关文法。上下文无关文法设计的具体依据在该次实验报告中已经详细阐述了，在本次报告中就不再赘述。

上下文无关文法

```
// 程序
Program
    :   Stmts
    ;

// 语句序列
Stmts
    :   Stmts Stmt
    |   Stmt
    ;

// 语句
Stmt
    :   AssignStmt
    |   ExpStmt SEMICOLON
    |   BlockStmt
    |   IfStmt
    |   WhileStmt
    |   BreakStmt
    |   ContinueStmt
    |   ReturnStmt
    |   DeclStmt
    |   FuncDef
    |   SEMICOLON
    ;

// 左值
```

```
LVal
    :   ID
    |   ID ArrValIndices
    ;

// 赋值语句
AssignStmt
    :   LVal ASSIGN Exp SEMICOLON
    ;

// 表达式语句
ExpStmt
    :   ExpStmt COMMA Exp
    |   Exp
    ;

// 语句快
BlockStmt
    :   LBRACE Stmts RBRACE
    |   LBRACE RBRACE
    ;

// if语句
IfStmt
    :   IF LPAREN Cond RPAREN Stmt %prec THEN
    |   IF LPAREN Cond RPAREN Stmt ELSE Stmt
    ;

//while 语句
WhileStmt
    :   WHILE LPAREN Cond RPAREN Stmt
    ;

//break 语句
BreakStmt
    :   BREAK SEMICOLON
    ;

//continue 语句
ContinueStmt
    :   CONTINUE SEMICOLON
    ;


// return 语句
ReturnStmt
    :   RETURN Exp SEMICOLON
    |   RETURN SEMICOLON
    ;
```

```
77
78  // 变量表达式
79  Exp
80      :   AddExp
81      ;
82
83  // 常量表达式
84  ConstExp
85      :   AddExp
86      ;
87
88  // 加法级表达式
89  AddExp
90      :   MulExp
91      |   AddExp ADD MulExp
92      |   AddExp SUB MulExp
93      ;
94
95  // 乘法级表达式
96  MulExp
97      :   UnaryExp
98      |   MulExp MUL UnaryExp
99      |   MulExp DIV UnaryExp
100     |   MulExp MOD UnaryExp
101     ;
102
103 // 非数组表达式
104 UnaryExp
105     :   PrimaryExp
106     |   ID LPAREN FuncRParams RPAREN
107     |   ADD UnaryExp
108     |   SUB UnaryExp
109     |   NOT UnaryExp
110     ;
111
112 // 基础表达式
113 PrimaryExp
114     :   LVal
115     |   LPAREN Exp RPAREN
116     |   INTEGER
117     |   FLOATING
118     ;
119
120 // 函数参数列表
121 FuncRParams
122     :   FuncRParams COMMA Exp
123     |   Exp
124     |   %empty
125     ;
```

```
126
127 // 条件表达式
128 Cond
129     :   LOrExp
130     ;
131
132 // 或运算表达式
133 LOrExp
134     :   LAndExp
135     |   LOrExp OR LAndExp
136     ;
137
138 // 与运算表达式
139 LAndExp
140     :   EqExp
141     |   LAndExp AND EqExp
142     ;
143
144 // 相等判断表达式
145 EqExp
146     :   RelExp
147     |   EqExp EQ RelExp
148     |   EqExp NEQ RelExp
149     ;
150
151 // 关系表达式
152 RelExp
153     :   AddExp
154     |   RelExp LESS AddExp
155     |   RelExp LESSEQ AddExp
156     |   RelExp GREAT AddExp
157     |   RelExp GREATEQ AddExp
158     ;
159
160 // 类型
161 Type
162     :   TYPE_INT
163     |   TYPE_FLOAT
164     |   TYPE_VOID
165     ;
166
167 // 数组的常量下标表示
168 ArrConstIndices
169     :   ArrConstIndices LBRACKET ConstExp RBRACKET
170     |   LBRACKET ConstExp RBRACKET
171     ;
172
173 // 数组的变量下标表示
174 ArrValIndices
```

```
175     :    ArrValIndices LBRACKET Exp RBRACKET
176     |    LBRACKET Exp RBRACKET
177     ;
178
179 // 声明语句
180 DeclStmt
181     :    CONST Type ConstDefList SEMICOLON
182     |    Type VarDefList SEMICOLON
183     ;
184
185 // 常量定义列表
186 ConstDefList
187     :    ConstDefList COMMA ConstDef
188     |    ConstDef
189     ;
190
191 // 常量定义
192 ConstDef
193     :    ID ASSIGN ConstExp
194     |    ID ArrConstIndices ASSIGN ConstInitVal
195     ;
196
197 // 常量初始化值
198 ConstInitVal
199     :    ConstExp
200     |    LBRACE ConstInitValList RBRACE
201     |    LBRACE RBRACE
202     ;
203
204 // 数组常量初始化列表
205 ConstInitValList
206     :    ConstInitValList COMMA ConstInitVal
207     |    ConstInitVal
208     ;
209
210 // 变量定义列表
211 VarDefList
212     :    VarDefList COMMA VarDef
213     |    VarDef
214     ;
215
216 // 变量定义
217 VarDef
218     :    ID
219     |    ID ASSIGN Exp
220     |    ID ArrConstIndices
221     |    ID ArrConstIndices ASSIGN VarInitVal
222     ;
223
```

```
// 变量初始化值
VarInitVal
    :   Exp
    |   LBRACE VarInitValList RBRACE
    |   LBRACE RBRACE
    ;

// 数组变量初始化列表
VarInitValList
    :   VarInitValList COMMA VarInitVal
    |   VarInitVal
    ;

// 函数定义
FuncDef
    :   Type ID LPAREN FuncParams RPAREN BlockStmt
    ;

// 函数参数列表
FuncParams
    :   FuncParams COMMA FuncParam
    |   FuncParam
    |   %empty
    ;

// 函数参数
FuncParam
    :   Type ID
    |   Type ID LBRACKET RBRACKET ArrConstIndices
    |   Type ID LBRACKET RBRACKET
    ;
```

3.2 语法树构建

语法树构建的过程十分冗杂，其实难点并不多，重复工作很多，在实验报告中，只对实现相对复杂的重难点进行展示。

3.2.1 算数运算节点

对于算数运算节点，包含一元运算和二元运算，但无论是哪一种，在构建当前节点的时候，都需要先将运算数节点构建完成。对于四则运算的算术节点，在语法分析阶段，就对其运算结果节点的类型进行判断，需要选择运算节点中类型优先级较高的节点作为运算结果的节点，简单来说，如果两个运算数节点都为 int 类型，则运算结果的类型才能为 int 类型，如果任何一个运算数节点为 float，则最终的运算结果节点的类型就应该为 float。在这里提前进行了类型的判断，主要是方便在生成中间代码的时候，插入类型转换指令。这里需要注意的是，在对关系运算节点构建语法树的时候，无论其运算数节点的类型为什么，其运算结果节点的类型都应该为 bool 类型。这里简单展示两个示例：

算数运算节点语法树构建示例

```
AddExp
    :   AddExp ADD MulExp {
            SymbolEntry *se;
            if($1->getType()->isAnyInt() && $3->getType()->isAnyInt()){
                se = new TemporarySymbolEntry(TypeSystem::intType,
                    SymbolTable::getLabel());
            }
            else{
                se = new TemporarySymbolEntry(TypeSystem::floatType,
                    SymbolTable::getLabel());
            }
            $$ = new BinaryExpr(se, BinaryExpr::ADD, $1, $3);
        }
RelExp
    :   RelExp LESS AddExp {
            SymbolEntry *se = new TemporarySymbolEntry(TypeSystem::boolType,
                SymbolTable::getLabel());
            $$ = new BinaryExpr(se, BinaryExpr::LESS, $1, $3);
        }
```

3.2.2 变量定义

对于变量定义，其实会出现大量的变量定义列表，如 int a, b, c =10; 此类。因此我们设计了一个 DeclStmt 节点来表示此类包含多个变量定义的语句。而对于单独的 a 或者 c=10 则定义了 DefNode，其中包含了两个域，一个是表示标识符的 Id 节点，另一个是用来表示初始化值得 initVal 节点。在 DeclStmt 节点中维护了一个保存 DefNode 的 vector。其结构如图3.3所示。

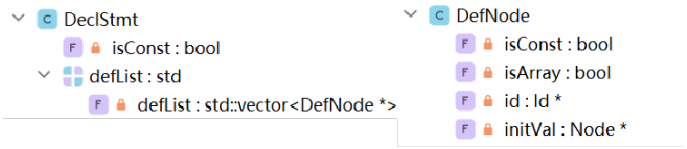

图 3.3: DeclStmt 和 DefNode 结构

下面展示一个简单的代码示例

变量定义列表示例代码

```
// 变量定义列表
VarDefList
    :   VarDefList COMMA VarDef {
            DeclStmt* node = dynamic_cast<DeclStmt*>($1);
            node->addNext(dynamic_cast<DefNode*>($3));
            $$ = node;
        }
    |   VarDef {
            DeclStmt* node = new DeclStmt(true);
            node->addNext(dynamic_cast<DefNode*>($1));
```

```
            $$ = node;
        }
    ;

// 变量定义
VarDef
    :   ID {
            Type* type = currentType->isInt() ? TypeSystem::intType :
                TypeSystem::floatType;
            SymbolEntry *se = new IdentifierSymbolEntry(type, $1,
                identifiers->getLevel());
            identifiers->install($1, se);
            $$ = new DefNode(new Id(se), nullptr, false, false);
        }
```

3.2.3 函数定义

对于函数定义节点，需要维护表示函数标识符的 Id 节点，params 来表示函数参数节点列表，以及 stmt 表示函数体，其具体结构如图3.4所示。

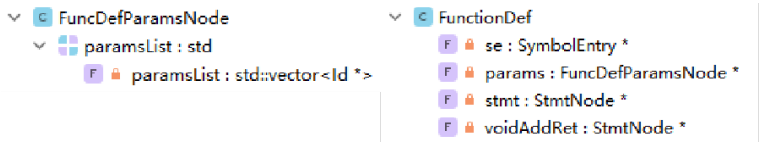

图 3.4: FunctionDef 结构

在函数定义的时候，需要注意的是在对函数参数构建语法树的时候，首先需要对符号表进行提升，因为函数参数的作用域应该在函数内部。此外对于函数的 FunctionType 中维护了函数参数的类型，所以在对函数参数构建语法树完成之后，需要将函数参数的类型写入到函数类型对应的域中。下面展示一段示例代码

函数定义示例代码

```
FuncDef
    :   Type ID {
            Type *funcType = new FunctionType($1,{});
            SymbolEntry *se = new IdentifierSymbolEntry(funcType, $2,
                identifiers->getLevel());
            identifiers->install($2, se);
            identifiers = new SymbolTable(identifiers);
        }
        LPAREN FuncParams{
            SymbolEntry *se;
            se = identifiers->lookup($2);
            if($5!=nullptr){
                (dynamic_cast<FunctionType*>(se->getType()))->setparamsType(
                    (dynamic_cast<FuncDefParamsNode*>($5))->getParamsType()
                );
```

```
            }
        }
        RPAREN BlockStmt {
            SymbolEntry *se;
            se = identifiers->lookup($2);
            $$ = new FunctionDef(se, dynamic_cast<FuncDefParamsNode*>($5), $8);
            SymbolTable *top = identifiers;
            identifiers = identifiers->getPrev();
        }
    ;
```

4 类型检查

在类型检查阶段，我们的工作重心放在了常量计算和数组相关的初始化上，此外还对部分 int 到 bool 的类型转换进行了处理。

4.1 常量计算

在语法分析阶段，我们对于常量和变量在对应的节点上通过标志位或者符号表中的标志位进行了区分，因此对于包含常量的表达式，完全可以在类型检查阶段，将其直接替换成字面值，使得后面生成的汇编代码更加简便，而对于可以进行计算的常量算数表达式，也全部进行常量计算的工作。这里我们以二元运算节点为例，首先对于两个运算数执行类型检查，这里是为了方便将可以进行常量计算的运算数进行计算。然后根据类型检查的结果，判断两个运算数是否为常量，如果均为常量，则根据运算符对该表达式节点进行计算，并且生成新的 Constant 节点保存计算结果，这里需要区分 int 和 float 两种类型的节点。下面给出一小段示例代码

常量计算示例代码

```
expr1->typeCheck((Node**)&(this->expr1));
expr2->typeCheck((Node**)&(this->expr2));
//检查是否void函数返回值参与运算
Type* realTypeLeft = expr1->getType()->isFunc() ?
    ((FunctionType*)expr1->getType())->getRetType() :
    expr1->getType();
Type* realTypeRight = expr2->getType()->isFunc() ?
    ((FunctionType*)expr2->getType())->getRetType() :
    expr2->getType();
//左右子树均为常数，计算常量值，替换节点
if(realTypeLeft->isConst() && realTypeRight->isConst()){
    SymbolEntry *se;
    if(this->getType()->isInt()){
        int val = 0;
        int leftValue = expr1->getSymPtr()->isConstant() ?
            ((ConstantSymbolEntry*)(expr1->getSymPtr()))->getValue() : //字面值常量
            ((IdentifierSymbolEntry*)(expr1->getSymPtr()))->value;  //符号常量
        int rightValue = expr2->getSymPtr()->isConstant() ?
            ((ConstantSymbolEntry*)(expr2->getSymPtr()))->getValue() :
```

```
            ((IdentifierSymbolEntry*)(expr2->getSymPtr()))->value;
        switch (op)
        {
        case ADD:
            val = leftValue + rightValue;
        break;
        case SUB:
            val = leftValue - rightValue;
        break;
        case MUL:
            val = leftValue * rightValue;
        break;
        case DIV:
            val = leftValue / rightValue;
        break;
        case MOD:
            val = leftValue % rightValue;
        break;
        }
        se = new ConstantSymbolEntry(TypeSystem::constIntType, val);
    }
    Constant* newNode = new Constant(se);
    *parentToChild = newNode;
}
```

4.2 类型转换

我们在语法分析阶段，已经对于运算节点的类型根据运算数的类型做了确认，由于我们没有打算实现一个隐式类型转换节点，因此在类型检查的阶段，并没有处理隐式类型转化的工作，而是放到了中间代码生成阶段进行处理。但是这里为了简化后续中间代码生成的工作，我们对于 int 到 bool 的类型转化做了处理。这个处理是十分必要的，一方面是对与和或运算，其两个运算数必须为 bool 类型，而此时如果传一个 int 类型的值，则需要进行类型转化。还有就是在 if 的条件判断的时候，经常会传一个 int 值，以其不为 0 作为判断标准。因此在类型检查的时候，我们对上述情况进行了处理，当发现一个需要 bool 类型的值的接受了一个 int 或者 float 类型的值的时候，我们增加了一个该 int 值和 0 不相等的比较节点，这样就实现了从 int 到 bool 的类型转换。这里展示部分 int 到 bool 的类型转换示例代码。

int 到 bool 类型转化示例代码

```
void BinaryExpr::typeCheck(Node** parentToChild)
{
    if(op == AND || op == OR) {
            if(!expr1->getSymPtr()->getType()->isBool() ||
                expr1->getSymPtr()->isConstant()) {
                Constant* zeroNode = new Constant(new
                    ConstantSymbolEntry(TypeSystem::constIntType, 0));
                TemporarySymbolEntry* tmpSe = new
                    TemporarySymbolEntry(TypeSystem::boolType,
```

```
                    SymbolTable::getLabel());
                BinaryExpr* newCond = new BinaryExpr(tmpSe, BinaryExpr::NEQ,
                    zeroNode, expr1);
                expr1 = newCond;
            }
            if(!expr2->getSymPtr()->getType()->isBool() ||
                expr2->getSymPtr()->isConstant()) {
                Constant* zeroNode = new Constant(new
                    ConstantSymbolEntry(TypeSystem::constIntType, 0));
                TemporarySymbolEntry* tmpSe = new
                    TemporarySymbolEntry(TypeSystem::boolType,
                    SymbolTable::getLabel());
                BinaryExpr* newCond = new BinaryExpr(tmpSe, BinaryExpr::NEQ,
                    zeroNode, expr2);
                expr2 = newCond;
            }
        }
}
void IfStmt::typeCheck(Node** parentToChild)
{
    if(!cond->getSymPtr()->getType()->isBool() || cond->getSymPtr()->isConstant()) {
        Constant* zeroNode = new Constant(new
            ConstantSymbolEntry(TypeSystem::constIntType, 0));
        TemporarySymbolEntry* tmpSe = new TemporarySymbolEntry(TypeSystem::boolType,
            SymbolTable::getLabel());
        BinaryExpr* newCond = new BinaryExpr(tmpSe, BinaryExpr::NEQ, zeroNode, cond);
        cond = newCond;
    }
}
```

4.3 数组相关初始化

数组相关的初始化，类型检查阶段，主要是将数组各个维度的大小进行计算，并且将计算的结果写入到 ArrayType 中对应的维度域中。这里的大致流程是，首先在对 ID 进行类型检查的时候，判断当前的符号表项对应的类型是否为函数类型，如果是函数类型，则需要对 indices 域进行类型检查，这步类型检查考虑到在数组定义的时候，维度必定是常量，因此在我们提供了常量计算之后，这个值能够在类型检查阶段确定。在对 indices 完成了类型检查之后，就需要判断当前的 ArrayType 中维度域是否被初始化过，如果没有被初始化则需要对该维度域进行初始化。下面展示一小段示例代码

数组维度初始化示例代码

```
void Id::typeCheck(Node** parentToChild)
{
    if(isArray() && indices!=nullptr){
        indices->typeCheck(nullptr);
        if((getType()->isArray() &&
            dynamic_cast<ArrayType*>(getType())->getDimensions().empty()) {
            indices->initDimInSymTable((IdentifierSymbolEntry*)getSymPtr());
```

```
        }
    }
}
void ExprStmtNode::initDimInSymTable(IdentifierSymbolEntry* se)
{
    for(auto expr :exprList){
        // 字面值常量，值存在ConstantSymbolEntry中
        if(expr->getSymPtr()->isConstant()){
            if(se->getType()->isArray()){
                dynamic_cast<ArrayType*>(se->getType())->pushBackDimension(
                (int)((ConstantSymbolEntry*)(expr->getSymPtr()))->getValue());
            }
        }
    }
}
```

在完成了数组维度信息的初始化之后，我们还需要对有初始值的数组进行初始化，这个过程是非常复杂的，因为给出的初始化结构可能和数组维度完全不匹配，因此在实现的过程中，考虑采用递归遍历的方式将所有的初始化值展平成一个维度，并且按照需要进行补零填充。

在对 DefNode 进行类型检查的时候，如果发现当前定义了一个数组并且有初始化值得时候，需要首先通过 ArrayUtil 记录下数组得维度信息，方便后续进行填充。然后对 initVal 进行类型检查，在完成了对其类型检查之后，可以将 initVal 节点替换成展平并填充 0 之后得初始化值节点。

数组初始化值

```
void DefNode::typeCheck(Node** parentToChild)
{
    id->typeCheck(nullptr);
    if(id->getType()->isArray()) {
        ArrayUtil::init();
        ArrayUtil::setArrayType(id->getType());
        initVal->typeCheck((Node**)&(initVal));
        this->initVal = new
            InitValNode(dynamic_cast<InitValNode*>(this->initVal)->isConst());
        std::vector<ExprNode*> initList = ArrayUtil::getInitVals();
        for(auto & child : initList){
            InitValNode* newNode = new
                InitValNode(dynamic_cast<InitValNode*>(this->initVal)->isConst());
            newNode->setLeafNode(child);
            dynamic_cast<InitValNode*>(this->initVal)->addNext(newNode);
        }
    }
}
```

在调用 InitVal 节点的类型检查函数的时候，会首先递增当前所咋的数组维度，便于在 ArrayUtil 内访问到当前维度所包含的元素数量，方便进行对齐。然后对当前其维护的孩子节点调用类型检查，完成后判断当前是否需要进行 padding。示例代码如下

数组初始化值填充

```
void InitValNode::typeCheck(Node** parentToChild)
{
    ArrayUtil::incCurrentDim();
    bool padding = true;
    if(this->leafNode != nullptr) {
        this->leafNode->typeCheck((Node**)&(this->leafNode));
        ArrayUtil::insertInitVal(this->leafNode);
        padding = false;

    }
    int size = 0;
    for(auto & child : innerList){
        child->typeCheck((Node**)&child);
        size++;
    }
    if(padding) {
        ArrayUtil::paddingInitVal(size);
    }
    ArrayUtil::decCurrentDim();
}
void ArrayUtil::paddingInitVal(int size) {
    int padding = getCurrentDimCapacity();
    while(size++ < arrayDims[currentArrayDim] || initVals.size() % padding != 0) {
        if(getElementType()->isInt()) {
            initVals.push_back(new Constant(new
                ConstantSymbolEntry(TypeSystem::constIntType, 0)));
        }
        else {
            initVals.push_back(new Constant(new
                ConstantSymbolEntry(TypeSystem::constFloatType, 0)));
        }
    }
};
```

5 中间代码生成

在中间代码生成阶段，我主要负责控制流相关 IR 的生成，并且增加了浮点数相关的 IR，同时提供了整型、浮点型和布尔类型之间的类型转换指令，最后为整个函数提供控制流分析操作。

5.1 短路支持

在本次实验中，要求支持逻辑短路，这里以与运算为例。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，因此对于有与运算而言，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。而对于关系运算，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。

逻辑短路代码示例

```
void BinaryExpr::genCode()
{
    BasicBlock *bb = builder->getInsertBB();
    Function *func = bb->getParent();
    Type* maxType = TypeSystem::getMaxType(expr1->getSymPtr()->getType(),
        expr2->getSymPtr()->getType());
    if (op == AND)
    {
        BasicBlock *trueBB = new BasicBlock(func);  // if the result of lhs is true,
            jump to the trueBB.
        genBr = 1;
        expr1->genCode();
        backPatch(expr1->trueList(), trueBB);
        builder->setInsertBB(trueBB);               // set the insert point to the
            trueBB so that intructions generated by expr2 will be inserted into it.
        expr2->genCode();
        true_list = expr2->trueList();
        false_list = merge(expr1->falseList(), expr2->falseList());
    }
    if(op >= LESS && op <= NEQ)
    {
        genBr--;
        expr1->genCode();
        expr2->genCode();
        genBr++;
        Operand *src1 = typeCast(maxType, expr1->getOperand());
        Operand *src2 = typeCast(maxType, expr2->getOperand());
        int opcode;
        switch (op)
        {
        case LESS:
            opcode = CmpInstruction::L;
            break;
        }
        new CmpInstruction(opcode, dst, src1, src2, bb);
        if(genBr > 0){
            // 跳转目标block
            BasicBlock* trueBlock, *falseBlock, *mergeBlock;
            trueBlock = new BasicBlock(func);
            falseBlock = new BasicBlock(func);
            mergeBlock = new BasicBlock(func);
            true_list.push_back(new CondBrInstruction(trueBlock, falseBlock, dst,
```

```
                bb));
            false_list.push_back(new UncondBrInstruction(mergeBlock, falseBlock));
        }
    }
}
```

5.2 隐式类型转换支持

由于我们支持 SysY 语言中的浮点运算，因此需要考虑到在运算过程中，浮点数和整数之间的类型转换工作。在任何一个可能发生类型转换的地方，比如表达式运算，赋值语句，函数调用传参，函数返回值等地方，都加入了隐式类型转化的处理。考虑到了这个方法会被重复利用，因此我们封装了一个 typecsat 的方法，该方法接受两个参数，一个是目标转向的类型，另一个是本身的操作数。如果是 bool 到 int 类型，则插入一条无符号扩展指令，如果是整数和浮点数之间的转换，则插入整型和浮点型之间的转换指令。

类型转换代码示例

```
Operand* Node::typeCast(Type* targetType, Operand* operand) {
    // 首先判断是否真的需要类型转化
    if(!TypeSystem::needCast(operand->getType(), targetType)) {
        return operand;
    }
    BasicBlock *bb = builder->getInsertBB();
    //Function *func = bb->getParent();
    Operand* retOperand = new Operand(new TemporarySymbolEntry(targetType,
        SymbolTable::getLabel()));
    // 先实现bool扩展为int
    if(operand->getType()->isBool() && targetType->isInt()) {
        // 插入一条符号扩展指令
        new ZextInstruction(operand, retOperand, bb);
    }
    // 实现 int 到 float 的转换
    else if(operand->getType()->isInt() && targetType->isFloat()) {
        // 插入一条类型转化指令
        new IntFloatCastInstructionn(IntFloatCastInstructionn::I2F, operand,
            retOperand, bb);
    }
    // 实现 float 到 int 的转换
    else if(operand->getType()->isFloat() && targetType->isInt()) {
        // 插入一条类型转化指令
        new IntFloatCastInstructionn(IntFloatCastInstructionn::F2I, operand,
            retOperand, bb);
    }
    return retOperand;
}
```

5.3 复杂控制流指令的生成

这里我们以 while 语句为例展示复杂控制流 IR 的生成，while 语句中可能会包含了 break 或 continue 等语句，会对原本的控制流进行打断。因此这个样例相对比较复杂，但能够较全面的反应控制流指令的生成过程。对于 while 语句，共需要生成 3 个新的基本块，一个基本快用来生成条件判断语句，一个基本块用来生成循环体，另一个基本块用来表示循环结束跳转的目标块。为了能够使得循环体内部的 break 和 continue 准确知道当前在哪一个循环内，我们还维护了一个 while 语句栈，每次对 while 语句生成中间代码的时候，都要把当前节点压栈，生成完成之后再出栈。在 while 节点内，维护了条件语句块和结束语句块，方便内部的 continue 和 break 语句找到目标跳转块。对于 continue 语句，应该生成一条无条件的跳转语句跳转到条件判断块，对于 break 语句应该生成一条无条件跳转指令跳转到结束块。

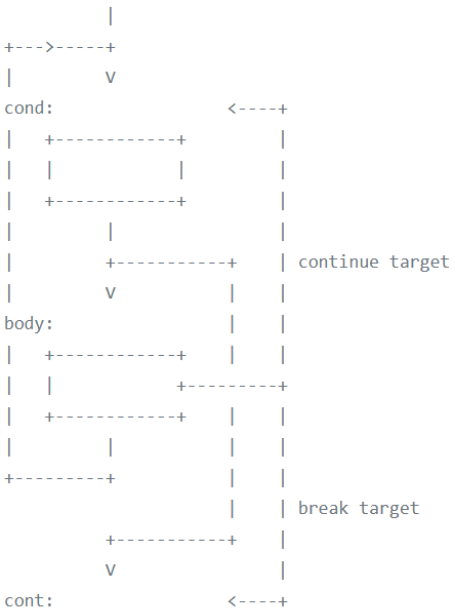

图 5.5: while 语句中控制流示例图

5.4 控制流分析

在我们生成 IR 的时候，其实只生成了基本块，并没有确定基本块之间的先后关系。而 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，因此我们可以利用这一特性，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，在实验框架中，采取了控制流图的方式进行实现，即采用有向图的方式来进行控制流分析。但这里需要注意的是，个别情况下，会在 ret 语句后还会生成中间代码，此时需要将 ret 语句后的中间代码全部移除掉。控制流图构建示例代码如下

控制流图构建示例代码

```
1 void FunctionDef::genCode()
2 {
3     ......
4     for (auto block = func->begin(); block != func->end(); block++) {
5         // 清除ret之后的全部指令
```

```
        Instruction* index = (*block)->begin();
        while(index != (*block)->end()) {
            if(index->isRet()) {
                while(index != (*block)->rbegin()) {
                    (*block)->remove(index->getNext());
                }
                break;
            }
            index = index->getNext();
        }
        Instruction* last = (*block)->rbegin();
        if (last->isCond()) {
            BasicBlock *trueBlock =
                dynamic_cast<CondBrInstruction*>(last)->getTrueBranch();
            BasicBlock *falseBlock =
                dynamic_cast<CondBrInstruction*>(last)->getFalseBranch();
            (*block)->addSucc(trueBlock);
            (*block)->addSucc(falseBlock);
            trueBlock->addPred(*block);
            falseBlock->addPred(*block);
        }
        if (last->isUncond()) {
            BasicBlock* dstBlock =
                dynamic_cast<UncondBrInstruction*>(last)->getBranch();
            (*block)->addSucc(dstBlock);
            dstBlock->addPred(*block);
        }
    }
}
```

5.5 浮点指令支持

在实验中，支持浮点运算，因此需要引入浮点指令，对于浮点指令，主要考虑两类，一类是浮点二元运算指令，另一类是浮点数的比较指令。在中间代码阶段，浮点指令的引入并无太多工作，只需要新创建两个指令类型，针对相应的运算进行不同的 output 操作。判断是否需要生成浮点指令，需要在遍历到语法树的二元运算节点的时候，判断两个运算数中是否有浮点数，如果有一个是浮点数，则需要将两个运算数全部提升为浮点类型，然后再根据运算符不同，生成浮点的二元运算指令或者浮点的比较指令。而对于浮点数更加复杂的处理在目标代码阶段再详细阐述。

6 目标代码生成

在目标代码生成阶段，我主要负责完成了控制流相关的汇编代码生成，全局变量相关定义，函数传参过程中多余四个参数的处理，数组初始化值的确定，以及浮点相关指令的引入。

6.1 控制流相关目标代码生成

对于比较指令，需要注意的是，除了需要根据需要的关系运算，生成对应的比较指令之外，在某些情况下，可能会用到比较运算的结果。但在汇编层面上，如果只是进行比较然后跳转，其实是不需要单独保存比较的结果的。跳转指令会自动从标志寄存器中读取相关的标志位，来判断是否需要进行跳转。但如果需要用比较指令的结果继续进行运算的话，就需要显式保存比较结果。在这里采用的是条件 mov 操作。例如：如果比较指令是一条大于的判断，则在完成比较指令之后，生成大于则移动 1 到结果寄存器，小于等于则移动 0 到结果寄存器这样两条条件移动指令。简单示例如下

cmp 汇编指令生成示例

```
cmp    r4, r5
movgt  r6, #1
movle  r6, #0
```

这里需要注意的是，由于浮点比较运算不会直接置位标志寄存器，因此需要手动将 FPSRC 浮点状态标志寄存器移动到标志寄存器中。其简单示例如下

vcmp.f32 汇编指令生成示例

```
vcmp.f32  s4, s5
vmrs  APSR_nzcv, FPSCR
movgt  r6, #1
movle  r6, #0
```

对于条件跳转指令，在汇编层面需要生成两条跳转指令，一条是条件成立时的有条件跳转指令，另一条是条件不成立时的无条件跳转指令。下面给出一个简单示例

条件跳转汇编指令生成示例

```
cmp   r4, r5
beq   .L1
b     .L2
```

6.2 全局数据目标代码生成

全局数据主要可以分为全局变量和全局常量，在类型检查阶段，我们已经对非数组的全局常量进行了常量替换操作，因此对于非数组的全局常量已经不需要进行处理了。而对于数组的全局常量，需要将其写入到 rodata 段中。对于全局变量需要将其写入到 data 段中。对于有初始值的变量，还需要在全局声明的时候带上初始值，这里需要注意的是，浮点数初始值在全局声明的时候，需要按照 32 位整数进行输出。

这里还需要单独说明一下数组的全局声明。对于没有初始值的全局数组，需要给其加上.comm 标签，加上这个标签之后，该数组的初始值被置为 0。对于有初始值的数组，在类型检查阶段我们已经完成了初始值按维度填充的工作。因此在这里只需要按照顺序声明初始值即可。

在访问全局数据的时候，这个操作一共包含两个步骤，第一步是将全局变量对应的地址 load 到一个寄存器中，第二步再从这个寄存器中 load 出全局变量的值。但是这里需要注意的是，由于 load 操作不能够从跨页的标签中取值，因此需要引入数据缓冲区，并将指令序列拆分，在本次实验中，每 500 条指令拆分一次，然后增加数据缓冲区，在数据缓冲区中记录全局变量的地址。每一段指令序列中的

全局变量访问，都需要从该序列对应的数据缓冲区中 load 全局变量的地址到寄存器中，然后再从寄存器中保存的地址处 load 出全局变量的数值。对于全局数组，还需要在 load 出变量所在的地址之后，加上在数组中的偏移。下面是一个简单的示例

全局数据目标代码生成示例

```
int a = 10;
int arr[2] = {1,2};
int main() {
    a = arr[0];
    return 0;
}
  .data
  .global a
  .align 4
  .size a, 4
a:
  .word 10
  .global arr
  .align 4
  .size arr, 8
arr:
  .word 1
  .word 2
  .text
  .global main
  .type main , %function
main:
  push {r8, r9, r10, fp, lr}
  mov fp, sp
.L2:
  ldr r10, =0
  ldr r9, =4
  mul r8, r10, r9
  ldr r10, addr_arr_0
  add r9, r8, r10
  ldr r10, [r9]
  ldr r9, addr_a_0
  str r10, [r9]
  ldr r10, =0
  mov r0, r10
  b .Lmain_END
.Lmain_END:
  pop {r8, r9, r10, fp, lr}
  bx lr
addr_a_0:
  .word a
addr_arr_0:
  .word arr
```

6.3 多参数函数传参处理

在 ARM 汇编中，函数传参可以使用部分寄存器来进行，对于整数传参，参数按顺序使用 r0-r3 寄存器，对于浮点数传参，参数按顺序使用 s0-s3 寄存器，这里需要注意，传参中既有整数又有浮点数的时候，整数采用整数寄存器，浮点数采用浮点寄存器，两类分别计数。

对于参数某一类参数数量超过 4 个的，需要采用压栈的方式传参，需要注意的是，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，我们都先开辟了栈帧，然后将参数 store 到栈中对应的位置。这里对于小于 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。

在传参的过程中，如何获取当前的参数应该获取整型曜浮点型的哪个寄存器是一个比较麻烦的事情。为了解决这个问题，我们在对函数定义的节点生成中间代码，能够获取到此时函数定义的参数，将参数对应的 operand 按照顺序保存下来，此时需要区分参数的类型，只有浮点数需要是浮点型，整数和数组传参都是整型，因此需要将 operand 分类保存，在查询参数 id 的时候也要分类查询。

在函数调用的时候，也需要首先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。

接下来需要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，我们会将参数保存到栈中的某个位置，这是通过设置 operand 的 offset 实现的。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。但是这里还有一个问题，由于此时并没有进行寄存器的分配，因此我们并不知道在进入函数的时候需要保存那些寄存器，因此参数在栈中真正的偏移现在还不能确定，需要等到完成了寄存器分配之后再进行更新，为此在此我们将压栈传参对应的 operand 先保存下来，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。

这里给出简单的示例

多参数函数传参

```
void CallInstruction::genMachineCode(AsmBuilder* builder){
    for(unsigned int i = operands.size() - 1; i > 0; i--){
        if(iparam_cnt < 4){
            auto dst = new MachineOperand(MachineOperand::REG, iparam_cnt);//r0-r3
            cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
                dst_addr);
            cur_block->InsertInst(cur_inst);
        }
        else{
            additional_args.clear();
            additional_args.push_back(dst_addr);
            cur_inst = new StackMInstruction(cur_block, StackMInstruction::PUSH,
                additional_args);
            cur_block->InsertInst(cur_inst);
            saved_reg_cnt++;
        }
```

```
    }
    ......
    // 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);
    }
}
void StoreInstruction::genMachineCode(AsmBuilder* builder){
    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;
            }
        }
    }
}
void MachineFunction::output(){
    // 调整 additional_args 中的偏移
    for(auto param : this->saved_params_offset) {
        param->setVal(4 * (this->saved_regs.size() + 2) + param->getVal());
    }
}
```

6.4 浮点指令目标代码生成

在这里首先要介绍一下浮点立即数的处理，在 ARM 中，将浮点数当作 32 位整数进行输出。在使用浮点立即数的时候，首先需要通过一条 load 指令，将 32 位整数表示的浮点数 load 到一个整型寄存器中，然后再通过 vmov 指令从这个整型寄存器中移动到浮点型寄存器中。比如使用浮点立即数 1.0，简单示例如下

浮点立即数目标代码示例

```
ldr r10, =1065353216
vmov s31, r10
```

浮点数的内存读写操作也需要和整数区别开，但是这里需要做的修改并不大，只需要将之前的 str 指令和 ldr 指令修改为 vstr.32 和 vldr.32 即可，并且将 store 指令的源寄存器和 load 指令的目的寄存器指定为浮点寄存器即可。例如给局部变量 a 加 1.0 的操作示例如下

浮点数读写目标代码示例

```
float a = 1.0;
a = a + 1.0;

ldr r10, =1065353216
vmov s31, r10
vstr.32 s31, [fp, #-4]
vldr.32 s31, [fp, #-4]
ldr r10, =1065353216
vmov s30, r10
vadd.f32 s29, s31, s30
vstr.32 s29, [fp, #-4]
```

浮点数还会参与算术运算和关系运算，因此还需要增加相关的浮点指令。在汇编层面也比较简单，只需要将之前的 add 等指令修改为 vadd.f32 等即可。对于比较指令，只需要将之前的 cmp 修改为 vcmp.f32，但是需要注意的是，浮点比较指令，并不会直接修改标志寄存器，需要手动将浮点标志寄存器的值移动到标志寄存器中，即需要在 vcmp.f32 指令之后，紧跟一条 vmrs APSR_nzcv, FPSCR 指令，这样之后的跳转才能够读取到相应的标志位然后进行条件判断。代码示例如下

浮点数运算目标代码示例

```
void BinaryMInstruction::output()
{
    switch (this->op)
    {
    case BinaryMInstruction::ADD:
        fprintf(yyout, "\tadd ");
        break;
    case BinaryMInstruction::SUB:
        fprintf(yyout, "\tsub ");
        break;
    case BinaryMInstruction::MUL:
        fprintf(yyout, "\tmul ");
        break;
    case BinaryMInstruction::DIV:
        fprintf(yyout, "\tsdiv ");
        break;
    case BinaryMInstruction::AND:
        fprintf(yyout, "\tand ");
        break;
```

```
20      case BinaryMInstruction::OR:
21          fprintf(yyout, "\tor ");
22          break;
23      case BinaryMInstruction::VADD:
24          fprintf(yyout, "\tvadd.f32 ");
25          break;
26      case BinaryMInstruction::VSUB:
27          fprintf(yyout, "\tvsub.f32 ");
28          break;
29      case BinaryMInstruction::VMUL:
30          fprintf(yyout, "\tvmul.f32 ");
31          break;
32      case BinaryMInstruction::VDIV:
33          fprintf(yyout, "\tvdiv.f32 ");
34          break;
35      default:
36          break;
37      }
38      this->PrintCond();
39      this->def_list[0]->output();
40      fprintf(yyout, ", ");
41      this->use_list[0]->output();
42      fprintf(yyout, ", ");
43      this->use_list[1]->output();
44      fprintf(yyout, "\n");
45  }
46  void CmpMInstruction::output()
47  {
48      switch(this->type) {
49          case CMP:
50              fprintf(yyout, "\tcmp ");
51              break;
52          case VCMP:
53              fprintf(yyout, "\tvcmp.f32 ");
54              break;
55          default:
56              break;
57      }
58      this->use_list[0]->output();
59      fprintf(yyout, ", ");
60      this->use_list[1]->output();
61      fprintf(yyout, "\n");
62  }
63  void VmrsMInstruction::output() {
64      fprintf(yyout, "\tvmrs APSR_nzcv, FPSCR\n");
65  }
```

在引入浮点数之后，就会涉及到浮点数和整数之间的隐式类型转换问题，在中间代码阶段，我们已经在 IR 中显式标准了类型转化的中间代码，因此在生成目标代码的时候，只需要根据 IR 中的转换指令选择相应的操作即可。汇编中提供了 VCVT 指令来辅助完成浮点数和整数之间的类型转换问题，相关的操作也比较简单，这里直接展示 output 函数即可。

浮点数和整数类型转换目标代码示例

```
void VcvtMInstruction::output() {
    switch (this->op) {
        case VcvtMInstruction::F2S:
            fprintf(yyout, "\tvcvt.s32.f32 ");
            break;
        case VcvtMInstruction::S2F:
            fprintf(yyout, "\tvcvt.f32.s32 ");
            break;
        default:
            break;
    }
    PrintCond();
    fprintf(yyout, " ");
    this->def_list[0]->output();
    fprintf(yyout, ", ");
    this->use_list[0]->output();
    fprintf(yyout, "\n");
}
```

7 总结

为期一个学期的编译原理实验至此终于结束了，在这一个学期的理论学习和实验探究中，我对于编译原理的算法理论有了清晰的认识，通过工程实验，对于编译器的前后端架构以及各模块之间的工作流程和实现细节也有了更加深刻的理解。在实验过程中，我对 ARM 汇编以及程序执行过程的底层原理也进行了比较全面和深刻的学习。通过一个学期的不断努力，我们最终除了实现了 SysY 语言的基础语法特性之外，我们还额外实现了浮点数和数组等高级语法特性。实验码量最终达到了 9000+ 行，通过了 144 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，拿到了 99 分这一令人满意的成绩。

在实验过程中，由于最初对于整体工程以及理论认识不够深刻，导致在前期的一些框架的设计上出现了难复用难扩展的问题，这也在后续的功能实现中给我们带来了巨大的麻烦。经过这一个学期的深刻教训，让我们更加深刻意识到，对于工程实验，前期的框架设计以及理论调研有着十分重要的意义。

最后，感谢这一个学期以来王刚老师尽职尽责的理论讲解，感谢时浩铭、林坤、尧泽斌学长和严诗慧、周辰霏学姐的耐心解答，感谢赵健坤同学在合作过程中为我提供的支持和帮助，为我们的实验克服了诸如数组、传参、寄存器分配等关键问题。