



南开大学  
Nankai University

南 开 大 学

计 算 机 学 院

编译原理实验报告

---

## 简易 SysY 语言编译器的实现

---

2113644 于洋淼 2111445 马睿遥

年级：2021 级

专业：物联网工程

指导教师：王刚

2024 年 1 月 16 日

## 摘要

本实验结合理论课学习，构建了一个简易的 SysY 语言编译器，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，本小组还完成了数组、函数等部分进阶要求。

**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，编译器**

# 目录

<b>一、 编译器整体结构</b>	<b>1</b>
(一) 编译器整体结构	1
1. 词法分析	1
2. 语法分析	1
3. 类型检查	1
4. 中间代码生成	1
5. 目标代码生成	1
(二) 小组分工	1
1. 词法分析	1
2. 语法分析	1
3. 类型检查	2
4. 中间代码生成	2
5. 目标代码生成	2
<b>二、 词法分析</b>	<b>2</b>
<b>三、 语法分析</b>	<b>3</b>
(一) 多维数组赋值	3
(二) 函数定义	4
(三) 各级表达式	5
<b>四、 类型检查</b>	<b>7</b>
(一) 隐式转换	7
(二) 常量计算	8
(三) 返回值类型检查	10
(四) 数组初始化	10
<b>五、 中间代码生成</b>	<b>11</b>
(一) 控制流语句翻译	11
(二) 数组偏移量计算	12
(三) 建立控制流图	13
<b>六、 目标代码生成</b>	<b>15</b>
(一) 浮点数相关汇编代码	15
1. vmov 指令	15
2. vstr 和 vldr 指令	15
3. 浮点数运算	15
(二) 全局变量声明及定义	15
(三) 函数调用及传参	16

## 一、 编译器整体结构

### (一) 编译器整体结构



图 1: 编译器整体结构

#### 1. 词法分析

词法分析阶段完成了将源代码的字符序列转化为一系列的 token。功能上实现了跳过空白字符及注释、识别整形和浮点数、识别关键字、识别标识符、识别 sysy 库函数并添加进符号表。

#### 2. 语法分析

语法分析主要完成了语法树的构建。这部分根据设计的 CFG 文法以及词法分析阶段识别出来的 token 来构建语法树节点、将标识符添加到对应的符号表中、确定变量及函数返回值类型、确定数组各个维度下标。

#### 3. 类型检查

类型检查阶段主要完成了表达式类型检查、函数调用参数列表检查、函数返回值类型检查，对语法树上子树均为字面值常量的节点的数值计算。

#### 4. 中间代码生成

中间代码生成主要完成了对数据流和控制流语句的 IR 代码生成。数据流语句翻译包括了表达式运算、变量声明与赋值、函数参数声明与赋值、数组偏移量计算；控制流语句翻译实现了通过回填技术翻译 break、continue、while 等语句；对 bool/int/float 的隐式转换。

#### 5. 目标代码生成

目标代码生成阶段主要完成了根据中间代码结果翻译出 arm 汇编代码。完成了寄存器分配、浮点数单独的汇编指令、开辟栈帧与恢复栈帧、函数调用前后寄存器及栈寄存器的保存与恢复。

### (二) 小组分工

#### 1. 词法分析

本人负责十六进制数正则表达式编写、库函数识别以及符号表构建；马睿遥同学负责关键字、标识符、浮点数识别。

#### 2. 语法分析

本人负责多维数组赋值表达式、函数定义与调用、函数作用域及对应符号表创建；马睿遥同学负责加法级表达式、乘法级表达式、条件表达式、分支及循环语句。

### 3. 类型检查

本人负责函数调用参数列表检查、函数返回值检查、以及实际上在中间代码生成阶段实现的隐式转换；马睿遥同学负责常量计算、变量常量初始化。

### 4. 中间代码生成

本人负责控制流语句翻译、truelist/falselist 的回填、数组偏移量计算；马睿遥同学负责基本 IR 指令生成、函数定义及调用。

### 5. 目标代码生成

本人负责寄存器分配、全局变量声明及定义、函数调用栈帧调整；马睿遥同学负责数组相关汇编代码、浮点数相关汇编代码。

## 二、词法分析

词法分析大体上可以分为设计正则式识别 token，设计对应 token 的操作 (添加 sysy 函数到符号表、识别标识符传递给语法分析阶段等)

### 词法分析示例

```

1 INTEGER ([1-9][0-9]*|0)
2 OCTAL (0[0-7][0-7]*)
3 HEXAL (0(x|X)[0-9a-fA-F][0-9a-fA-F]*)
4 FLOATING ((([0-9]*[.][0-9]*([eE][+-]?[0-9]+)?)|([0-9]+[eE][+-]?[0-9]+)) [fLlIlL]?)
5 HEXADEDECIMAL_FLOAT (0[xX](([0-9A-Fa-f]*[.][0-9A-Fa-f]*([pP][+-]?[0-9]+)?)|([0-9A-Fa-f]+[pP][+-]?[0-9]+)) [fLlIlL]?)
6 ID ([[:alpha:]]_)[[:alpha:]][[:digit:]]_*
7
8 /* 结束符 */
9 EOL (\r\n|\n|\r)
10 /* 空白符 */
11 WHITE [\t]
12 /* 单行注释, //开始后匹配任意非\n字符 */
13 COMMENT (\/\[/[^\n]*)
14 /* 多行注释 */
15 commentbegin "/*"
16 /* 任意字符都有效 */
17 commentelement .
18 commentline \n
19 commentend "*/"
20 %x COMMENT
21
22 {ID} {
23     char *lexeme;
24     //dump_tokens("ID\t%s\n", yytext);
25     lexeme = new char[strlen(yytext) + 1];
26     strcpy(lexeme, yytext);

```

```

27     yylval.strtype = lexeme;
28     return ID;
29 }
30
31 "putint" {
32     //dump_tokens("ID\t%s\n", yytext);
33     char *lexeme = new char[strlen(yytext) + 1];
34     strcpy(lexeme, yytext);
35     yylval.strtype = lexeme;
36     if(identifiers->lookup(yytext)==nullptr){//符号表内未找到, 插入
37         vector<Type*> vec; //形参类型表
38         vec.push_back(TypeSystem::intType);
39         Type* funcType = new FunctionType(TypeSystem::voidType, vec);//返回类
           型void, 参数int
40         SymbolTable* globalSymbolTable = identifiers; // 全局符号表
41         while(globalSymbolTable->getPrev() != nullptr) {globalSymbolTable =
           globalSymbolTable->getPrev();}
42         SymbolEntry* entry = new IdentifierSymbolEntry(funcType, yytext, 0);
43         globalSymbolTable->install(yytext, entry);
44     }
45     return ID;
46 }

```

### 三、 语法分析

#### (一) 多维数组赋值

多维数组初始化列表的文法书写上使用 list 和 val 相互嵌套, 可以实现多维数组赋值表达式 (比如 1,2,3,4)。

##### 多维数组赋值

```

1 // 常量初始化值
2 ConstInitVal
3     : ConstExp {// 对于一个expr也需要先创建一个Node
4         InitValNode* newNode = new InitValNode(true);
5         newNode->setLeafNode((ExprNode*)$1);
6         $$ = newNode;
7     }
8 // 常量list, 用于常量数组初始化, 这样书写文法可以支持多维数组初始化时候嵌套
   多个{}
9 | LBRACE ConstInitValList RBACE{
10     $$ = $2;
11 }
12 | LBRACE RBACE{
13     $$ = new InitValNode(true);
14 }
15 ;

```

```

16
17 // 数组常量初始化列表
18 ConstInitValList
19   : ConstInitValList COMMA ConstInitVal{ // 可以嵌套多个{}
20       InitValNode* node = (InitValNode*)$1;
21       node->addNext((InitValNode*)$3);
22       $$ = node;
23   }
24   | ConstInitVal{ // 可以嵌套多个{}
25       InitValNode* newNode = new InitValNode(true);
26       newNode->addNext((InitValNode*)$1);
27       $$ = newNode;
28   }
29   ;

```

## (二) 函数定义

函数定义需要函数名及返回类型、参数列表、函数体。在识别到参数列表时创建新的符号表并压入符号表栈，在函数体结束后弹出符号表。

### 函数定义

```

1 // 函数定义
2 FuncDef
3   : Type ID {
4       Type *funcType;
5       funcType = new FunctionType($1,{});
6       SymbolEntry *se = new IdentifierSymbolEntry(funcType, $2, identifiers
7           ->getLevel());
8       identifiers->install($2, se);
9       identifiers = new SymbolTable(identifiers);
10   }
11   LPAREN FuncParams{
12       SymbolEntry *se;
13       se = identifiers->lookup($2);
14       assert(se != nullptr);
15       if($5!=nullptr){
16           //将函数参数类型写入符号表
17           ((FunctionType*)(se->getType()))->setparamsType(((
18               FuncDefParamsNode*)$5)->getParamsType());
19       }
20   }
21   RPAREN BlockStmt { // 释放符号表
22       SymbolEntry *se;
23       se = identifiers->lookup($2);
24       assert(se != nullptr);
25       $$ = new FunctionDef(se, (FuncDefParamsNode*)$5, $8);
26       SymbolTable *top = identifiers;
27       identifiers = identifiers->getPrev();

```

```

26         delete top;
27         delete [] $2;
28     }
29     ;

```

### (三) 各级表达式

设计 cfg 文法需要注意各级表达式的优先级。重点要注意单目表达式 > 乘法级表达式 > 加法级表达式 > 关系表达式

#### 各级表达式示例

```

1 // 加法级表达式
2 AddExp
3     : MulExp {
4         $$ = $1;
5     }
6     | AddExp ADD MulExp {
7         SymbolEntry *se;
8         if ($1->getType()->isInt () && $3->getType()->isInt ()) {
9             se = new TemporarySymbolEntry (TypeSystem::intType , SymbolTable::
10                getLabel ());
11         }
12         else {
13             se = new TemporarySymbolEntry (TypeSystem::floatType , SymbolTable
14                ::getLabel ());
15         }
16         $$ = new BinaryExpr (se , BinaryExpr::ADD, $1 , $3);
17     }
18     | AddExp SUB MulExp {
19         SymbolEntry *se;
20         if ($1->getType()->isInt () && $3->getType()->isInt ()) {
21             se = new TemporarySymbolEntry (TypeSystem::intType , SymbolTable::
22                getLabel ());
23         }
24         else {
25             se = new TemporarySymbolEntry (TypeSystem::floatType , SymbolTable
26                ::getLabel ());
27         }
28         $$ = new BinaryExpr (se , BinaryExpr::SUB, $1 , $3);
29     }
30     ;
31
32 // 乘法级表达式
33 MulExp
34     : UnaryExp {
35         $$ = $1;
36     }
37     | MulExp MUL UnaryExp {

```



```

34     SymbolEntry *se;
35     if ($1->getType()->isInt () && $3->getType()->isInt ()) {
36         se = new TemporarySymbolEntry (TypeSystem::intType, SymbolTable::
            getLabel());
37     }
38     else {
39         se = new TemporarySymbolEntry (TypeSystem::floatType, SymbolTable
            ::getLabel());
40     }
41     $$ = new BinaryExpr(se, BinaryExpr::MUL, $1, $3);
42 }
43 | MulExp DIV UnaryExp {
44     SymbolEntry *se;
45     if ($1->getType()->isInt () && $3->getType()->isInt ()) {
46         se = new TemporarySymbolEntry (TypeSystem::intType, SymbolTable::
            getLabel());
47     }
48     else {
49         se = new TemporarySymbolEntry (TypeSystem::floatType, SymbolTable
            ::getLabel());
50     }
51     $$ = new BinaryExpr(se, BinaryExpr::DIV, $1, $3);
52 }
53 | MulExp MOD UnaryExp {
54     SymbolEntry *se;
55     if ($1->getType()->isInt () && $3->getType()->isInt ()) {
56         se = new TemporarySymbolEntry (TypeSystem::intType, SymbolTable::
            getLabel());
57     }
58     else {
59         se = new TemporarySymbolEntry (TypeSystem::floatType, SymbolTable
            ::getLabel());
60     }
61     $$ = new BinaryExpr(se, BinaryExpr::MOD, $1, $3);
62 }
63 ;
64
65 // 非数组表达式
66 UnaryExp
67 : PrimaryExp {
68     $$ = $1;
69 }
70 | ID LPAREN FuncCallParams RPAREN { //mark 函数调用, 和一元表达式属于一个
    优先级
71     SymbolEntry *se;
72     se = identifiers->lookup($1);
73     if (se == nullptr)
74     {

```

```

75         fprintf(stderr, "identifier \"%s\" is undefined\n", (char*)$1);
76         delete [] (char*)$1;
77         assert(se != nullptr);
78     }
79     SymbolEntry *tmp = new TemporarySymbolEntry(se->getType(),
80         SymbolTable::getLabel());
81     $$ = new FuncCallNode(tmp, new Id(se), (FuncCallParamsNode*)$3);
82 }
83 | ADD UnaryExp {
84     $$ = $2;
85 }
86 | SUB UnaryExp {
87     SymbolEntry *tmp = new TemporarySymbolEntry($2->getType(),
88         SymbolTable::getLabel());
89     $$ = new UnaryExpr(tmp, UnaryExpr::SUB, $2);
90 }
91 | NOT UnaryExp {
92     SymbolEntry *tmp = new TemporarySymbolEntry($2->getType(),
93         SymbolTable::getLabel());
94     $$ = new UnaryExpr(tmp, UnaryExpr::NOT, $2);
95 }
96 ;

```

## 四、 类型检查

### (一) 隐式转换

隐式转换的主要实现在中间代码生成阶段，对于 int 和 float 的转换、bool->int 的转换，在类型检查阶段只是检查是否需要隐式转换，如果需要隐式转换则不会报类型错误，而是添加指令留到中间代码生成阶段实现；对于 int 到 bool 的转换，则需要类型转换阶段实现，添加一个该 int 值与 0 的比较节点。

#### 隐式转换

```

1 // int->bool 示例
2 void IfStmt::typeCheck(Node** parentToChild)
3 {
4     cond->typeCheck((Node*)&(this->cond));
5
6     // int 到 bool 的隐式转换(如果虽然不是 bool 但是是个常量，则比较这个值和 0)
7     if (!cond->getSymPtr()->getType()->isBool() || cond->getSymPtr()->
8         isConstant()) {
9         Constant* zeroNode = new Constant(new ConstantSymbolEntry(TypeSystem
10             ::constIntType, 0)); // 创建一个 0 节点
11         TemporarySymbolEntry* tmpSe = new TemporarySymbolEntry(TypeSystem::
12             boolType, SymbolTable::getLabel()); // 创建一个临时的 bool 符号表项
13         BinaryExpr* newCond = new BinaryExpr(tmpSe, BinaryExpr::NOTEQUAL,
14             zeroNode, cond); // 创建一个新的表达式，比较原条件语句和 0 是否不相

```

```

    等
11     cond = newCond;
12 }
13 }
14
15 // 其余的隐式转换
16 Operand* Node::typeCast(Type* targetType, Operand* operand) {
17     // 首先判断是否真的需要类型转化
18     if (!TypeSystem::needCast(operand->getType(), targetType)) {
19         return operand;
20     }
21     BasicBlock *bb = builder->getInsertBB();
22     Function *func = bb->getParent();
23     Operand* retOperand = new Operand(new TemporarySymbolEntry(targetType,
24                                     SymbolTable::getLabel()));
25
26     // bool 到 int 的转换, 符号扩展
27     if (operand->getType()->isBool() && targetType->isInt()) {
28         new ZextInstruction(operand, retOperand, bb);
29     }
30     // int 到 float 的转换
31     else if (operand->getType()->isInt() && targetType->isFloat()) {
32         new IntFloatCastInstructionn(IntFloatCastInstructionn::I2F, operand,
33                                     retOperand, bb);
34     }
35     // float 到 int 的转换
36     else if (operand->getType()->isFloat() && targetType->isInt()) {
37         new IntFloatCastInstructionn(IntFloatCastInstructionn::F2I, operand,
38                                     retOperand, bb);
39     }
40     return retOperand;
41 }

```

## (二) 常量计算

在语法分析结束构建好了语法树后, 可以将子树为常量的节点的值计算出来, 替换为字面值常量, 这样之后无论是生成中间代码还是目标代码都可以直接用字面值。

首先对于两个运算数执行类型检查, 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, 并且生成新的字面值常量节点保存计算结果, 注意这里要区分 int/float。

### 常量计算

```

1 // 检查左右孩子
2 expr1->typeCheck((Node*)&(this->expr1));
3 expr2->typeCheck((Node*)&(this->expr2));
4 // 检查左右孩子是否为void函数返回值
5 Type* realTypeLeft = expr1->getType()->isFunc() ? ((FunctionType*)expr1->
    getType())->getRetType() : expr1->getType();

```

```

6  if (!realTypeLeft->calculatable()) {
7      exit(EXIT_FAILURE);
8  }
9  Type* realTypeRight = expr2->getType()->isFunction() ? ((FunctionType*)expr2->
    getType()->getRetType() : expr2->getType());
10 if (!realTypeRight->calculatable()) {
11     exit(EXIT_FAILURE);
12 }
13 //左右子树均为常数，计算常量值，替换节点
14 if (realTypeLeft->isConst() && realTypeRight->isConst()) {
15     SymbolEntry *se;
16     // 如果该节点结果的目标类型为bool
17     if (this->getType()->isBool()) {
18         bool val = 0;
19         float leftValue = expr1->getSymPtr()->isConstant() ? ((
                ConstantSymbolEntry*)(expr1->getSymPtr()))->getValue() : ((
                IdentifierSymbolEntry*)(expr1->getSymPtr()))->value;
20         float rightValue = expr2->getSymPtr()->isConstant() ? ((
                ConstantSymbolEntry*)(expr2->getSymPtr()))->getValue() : ((
                IdentifierSymbolEntry*)(expr2->getSymPtr()))->value;
21         switch(op)
22         {
23             case AND:
24                 val = leftValue && rightValue;
25             break;
26             case OR:
27                 val = leftValue || rightValue;
28             break;
29             case LESS:
30                 val = leftValue < rightValue;
31             break;
32             case LESSEQUAL:
33                 val = leftValue <= rightValue;
34             break;
35             case GREATER:
36                 val = leftValue > rightValue;
37             break;
38             case GREATEREQUAL:
39                 val = leftValue >= rightValue;
40             break;
41             case EQUAL:
42                 val = leftValue == rightValue;
43             break;
44             case NOTEQUAL:
45                 val = leftValue != rightValue;
46             break;
47         }
48         se = new ConstantSymbolEntry(TypeSystem::constBoolType, val);

```

```

49     }
50     Constant* newNode = new Constant(se);
51     *parentToChild = newNode;
52 }

```

### (三) 返回值类型检查

检查函数返回值依次检查是否出现返回语句不在函数中、void 有返回值、非 void 无返回值、返回值类型与函数返回类型不匹配的情况。

#### 返回值类型检查

```

1 // 返回语句
2 void ReturnStmt::typeCheck(Node** parentToChild)
3 {
4     // 不在函数中, 错误
5     if(returnType == nullptr){
6         fprintf(stderr, "return statement outside functions\n");
7         exit(EXIT_FAILURE);
8     }
9     // 返回void类型, 但是有返回值, 错误
10    else if(returnType->isVoid() && retValue!=nullptr){
11        fprintf(stderr, "value returned in a void() function\n");
12        exit(EXIT_FAILURE);
13    }
14    // 返回非void类型, 但是没有返回值, 错误
15    else if(!returnType->isVoid() && retValue==nullptr){
16        fprintf(stderr, "expected a %s type to return, but returned nothing\n",
17                returnType->toStr().c_str());
18        exit(EXIT_FAILURE);
19    }
20    //检查非void语句的返回值类型
21    if(!returnType->isVoid()){
22        retValue->typeCheck((Node**)&(retValue));
23    }
24    this->retType = returnType;
25    funcReturned = true;//都正确才正确
26 }

```

### (四) 数组初始化

类型检查阶段要计算数组各个维度大小, 并将数组各个维度大小保存下来。

对标识符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数组类型, 如果是数组类型, 则对 indices 进行类型检查。对 indices 完成了类型检查之后, 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。

#### 数组初始化

```

1 void Id::typeCheck(Node** parentToChild)

```

```

2 {
3     // 如果当前标识符是数组且具有索引
4     if (isArray() && indices != nullptr)
5     {
6         // 进行数组索引的类型检查
7         indices->typeCheck(nullptr);
8
9         // 如果数组的维度为空, 需要根据索引初始化数组的维度
10        if ((getType()->isArray() && dynamic_cast<IntArrayType*>(getType())
11            ->getDimensions().empty()) ||
12            (getType()->isConstIntArray() && dynamic_cast<ConstIntArrayType
13            *>(getType())->getDimensions().empty()) ||
14            (getType()->isFloatArray() && dynamic_cast<FloatArrayType*>(
15            getType())->getDimensions().empty()) ||
16            (getType()->isConstFloatArray() && dynamic_cast<
17            ConstFloatArrayType*>(getType())->getDimensions().empty()))
18        {
19            // 初始化符号表中的数组维度信息
20            indices->initDimInSymTable((IdentifierSymbolEntry*)getSymPtr());
21        }
22    }
23 }

```

## 五、 中间代码生成

### (一) 控制流语句翻译

控制流语句要求实现短路求值。我们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跳转指令列表, true-list 中为无条件跳转指令跳转与条件跳转指令条件为真时的跳转指令, false-list 为条件跳转指令条件为假时的跳转指令, 这些指令的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。

以下面这段代码为例, 如果 op 为 AND, 那么只有表达式 1 为真才有必要去计算表达式 2。在表达式 1 进行 genCode 的时, 表达式 1 不知道自己结果为真和结果为假应该跳转到哪。当表达式 1 代码生成结束后, 我们才知道表达式 2 的基本块, 这时候把表达式 2 的基本块回填到表达式 1 的 true-list 中, 而表达式 1 为假的跳转位置应该是整个 and 表达式为假的跳转位置, 这部分交给父节点回填。

对于关系运算, 运算结果会出现真假两个分支, 新建两个基本块, 对于 true-list, 添加一个条件为真时跳转到为真基本块, 条件为假跳转到中转基本块; 对于 false-list, 添加一个从中转基本块跳转到为假基本块的无条件跳转指令。这样做的目的是为了在 false-list 进行回填后, true-list 中的指令依旧能跳转到正确的为假位置。

#### 控制流语句

```

1 // 二元表达式
2 void BinaryExpr::genCode()
3 {
4     BasicBlock *bb = builder->getInsertBB();

```

```

5   Function *func = bb->getParent();
6   Type* maxType = TypeSystem::getMaxType(expr1->getSymPtr()->getType(),
    expr2->getSymPtr()->getType());
7   if (op == AND)
8   {
9       BasicBlock *trueBB = new BasicBlock(func); // if the result of lhs
    is true, jump to the trueBB.
10      IsgenBranch = 1;
11      expr1->genCode(); // expr1在genCode时候不知道自己的truelist和falselist
12      backPatch(expr1->trueList(), trueBB); // 现在知道了是expr2的bb, 回填
13      builder->setInsertBB(trueBB); // 设置表达式2的插入点
14      expr2->genCode();
15      true_list = expr2->trueList();
16      false_list = merge(expr1->falseList(), expr2->falseList()); // 无法得知
    falselit, 等着父节点回填
17  }
18  else if (op == OR)
19  {
20      BasicBlock *falseBB = new BasicBlock(func);
21      IsgenBranch = 1;
22      expr1->genCode(); // 和and相反, and是truelist, or是falselist
23      backPatch(expr1->falseList(), falseBB);
24      builder->setInsertBB(falseBB);
25      expr2->genCode();
26      true_list = merge(expr1->trueList(), expr2->trueList());
27      false_list = expr2->falseList();
28  }
29  else if (op >= LESS && op <= NOTEQUAL)
30  {
31      if (IsgenBranch > 0) {
32          // 跳转目标block
33          BasicBlock* bb1, *bb2, *bb3;
34          bb1 = new BasicBlock(func);
35          bb2 = new BasicBlock(func);
36          bb3 = new BasicBlock(func);
37          true_list.push_back(new CondBrInstruction(bb1, bb3, dst, bb));
38          false_list.push_back(new UncondBrInstruction(bb3, bb2));
39      }
40  }
41  }

```

## (二) 数组偏移量计算

在中间代码生成阶段计算出数组元素的偏移量，通过遍历数组各个维度，计算偏移量并不断更新。

### 数组偏移量

```

1 // 计算数组元素的偏移

```

```

2 for(unsigned int i = 1; indices!=nullptr && i < indices->exprList.size(); i
  ++){
3   Operand* dim_i = new Operand(new ConstantSymbolEntry(TypeSystem::
    constIntType, dimensions[i]));
4   TemporarySymbolEntry* se1 = new TemporarySymbolEntry(TypeSystem::intType,
    SymbolTable::getLabel());
5   Operand* offset1 = new Operand(se1);
6   new BinaryInstruction(BinaryInstruction::MUL, offset1, offset, dim_i, bb)
    ; // offset1 = offset * dimensions[i]
7
8   TemporarySymbolEntry* se2 = new TemporarySymbolEntry(TypeSystem::intType,
    SymbolTable::getLabel());
9   offset = new Operand(se2);
10  new BinaryInstruction(BinaryInstruction::ADD, offset, offset1, indices->
    exprList[i]->getOperand(), bb); // offset = offset1 + indices[i]
11 }
12
13 // 索引维度 < 总维度, 需要再乘一次最后的维度
14 if(indices!=nullptr && indices->exprList.size() < dimensions.size()){
15   Operand* dim_i = new Operand(new ConstantSymbolEntry(TypeSystem::
    constIntType, dimensions[indices->exprList.size()]));
16   TemporarySymbolEntry* se1 = new TemporarySymbolEntry(TypeSystem::intType,
    SymbolTable::getLabel());
17   Operand* offset1 = new Operand(se1);
18   new BinaryInstruction(BinaryInstruction::MUL, offset1, offset, dim_i, bb)
    ; // offset1 = offset * dimensions[i]
19   offset = offset1;
20 }
21
22 Operand* offset1 = nullptr;
23 if(indices!=nullptr){
24   TemporarySymbolEntry* se1 = new TemporarySymbolEntry(TypeSystem::intType,
    SymbolTable::getLabel());
25   offset1 = new Operand(se1);
26   Operand* align = new Operand(new ConstantSymbolEntry(TypeSystem::
    constIntType, 4));
27   new BinaryInstruction(BinaryInstruction::MUL, offset1, offset, align, bb)
    ; // offset1 = offset * 4
28 }
29 else{
30   offset1 = new Operand(new ConstantSymbolEntry(TypeSystem::constIntType,
    0));
31 }

```

### (三) 建立控制流图

遍历所有基本块, 对于每个基本块, 首先清除 ret 之后的全部指令, 然后获取该块的最后一条指令, 对于有条件的跳转指令, 需要对其 true 分支和 false 分支都设置控制流关系; 对于无条



件的跳转指令，只需要对其目标基本块设置控制流关系即可。

### 控制流图

```

1 void FunctionDef::genCode()
2 {
3     // 遍历Function中所有的BasicBlock，在各个BasicBlock之间建立控制流关系
4     for (auto block = func->begin(); block != func->end(); block++) {
5         // 清除ret之后的全部指令
6         Instruction* index = (*block)->begin();
7         while(index != (*block)->end()) {
8             if(index->isRet()) {
9                 while(index != (*block)->rbegin()) {
10                     (*block)->remove(index->getNext());
11                 }
12                 break;
13             }
14             index = index->getNext();
15         }
16         // 获取该块的最后一条指令
17         Instruction* last = (*block)->rbegin();
18         // (*block)->output();
19         // 对于有条件的跳转指令，需要对其true分支和false分支都设置控制流关系
20         if (last->isCond()) {
21             BasicBlock *trueBlock = dynamic_cast<CondBrInstruction*>(last)->
22                 getTrueBranch();
23             BasicBlock *falseBlock = dynamic_cast<CondBrInstruction*>(last)->
24                 getFalseBranch();
25             (*block)->addSucc(trueBlock);
26             (*block)->addSucc(falseBlock);
27             trueBlock->addPred(*block);
28             falseBlock->addPred(*block);
29         }
30         // 对于无条件的跳转指令，只需要对其目标基本块设置控制流关系即可
31         if (last->isUncond()) {
32             BasicBlock* dstBlock = dynamic_cast<UncondBrInstruction*>(last)->
33                 getBranch();
34             (*block)->addSucc(dstBlock);
35             dstBlock->addPred(*block);
36         }
37     }
38 }

```

## 六、 目标代码生成

### (一) 浮点数相关汇编代码

#### 1. vmov 指令

使用浮点立即数，需要先将浮点数当做 32 位整数输出，通过 ld 指令先将其保存在整形寄存器中，进而再保存到浮点寄存器中。

vmov 示例

```
1 ldr r10, =0
2  vmov s30, r10
```

#### 2. vstr 和 vldr 指令

浮点数读写相比于整形，使用 vstr.32 代替 str，使用 vldr.32 代替 ldr。

#### 3. 浮点数运算

浮点数运算使用 vadd、vsub、vmul、vdiv 等指令，注意这些指令的操作数都是浮点寄存器，而不是整形寄存器。

### (二) 全局变量声明及定义

全局常量在类型检查阶段已经替换为字面值常量，因此我们只需要处理全局变量。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。

访问全局数据的时候，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。

全局变量汇编代码示例

```
1  .arch armv8-a
2  .arch_extension crc
3  .arm
4  .data
5  .global a
6  .align 4
7  .size a, 4
8  a:
9  .word 3
10 .global b
11 .align 4
12 .size b, 4
13 b:
14 .word 5
15 .global main
16 .type main , %function
17 main:
```

```

18     push {r8, r9, r10, fp, lr}
19     mov fp, sp
20     sub sp, sp, #4
21 .L1:
22     ldr r10, =5
23     str r10, [fp, #-4]
24     ldr r10, [fp, #-4]
25     ldr r9, addr_b_0
26     ldr r8, [r9]
27     add r9, r10, r8
28     mov r0, r9
29     b .Lmain_END
30 .Lmain_END:
31     add sp, sp, #4
32     pop {r8, r9, r10, fp, lr}
33     bx lr
34
35 addr_a_0:
36     .word a
37 addr_b_0:
38     .word b

```

### (三) 函数调用及传参

函数传参可以使用部分寄存器，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，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，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，需要先开辟栈帧，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。

在函数调用的时候，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。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数量，在完成函数调用之后，将 sp 加上压栈传参数量乘 4。

## 函数调用及传参

```

1 void MachineFunction::output()
2 {
3
4     fprintf(yyout, "\t.global %s\n", this->sym_ptr->toStr().c_str() + 1);
5     fprintf(yyout, "\t.type %s , %%function\n", this->sym_ptr->toStr().c_str
6         () + 1);
7     fprintf(yyout, "%s:\n", this->sym_ptr->toStr().c_str() + 1);
8     // 保存寄存器
9     //3. Save callee saved register
10    fprintf(yyout, "\tpush {");
11    for(auto reg : getSavedRRegs()){
12        reg->output();
13        fprintf(yyout, ", ");
14    }
15    //1. Save fp
16    fprintf(yyout, "fp, lr}\n");
17
18    // 保存浮点寄存器
19    std::vector<MachineOperand*> fregs = getSavedFRegs();
20    if(!fregs.empty()) {
21        fprintf(yyout, "\tvpush {");
22        fregs[0]->output();
23        for (int i = 1; i < int(fregs.size()); i++) {
24            fprintf(yyout, ", ");
25            fregs[i]->output();
26        }
27        fprintf(yyout, "}\n");
28    }
29    // 调整 additional_args 中的偏移
30    for(auto param : this->saved_params_offset) {
31        param->setVal(4 * (this->saved_regs.size() + 2) + param->getVal());
32    }
33    //2. 设置fp = sp
34    fprintf(yyout, "\tmov fp, sp\n");
35    //4. 给局部变量分配栈空间
36    if(stack_size!=0){
37        if(stack_size > 255) {
38            fprintf(yyout, "\tldr r4,=%d\n", stack_size);
39            fprintf(yyout, "\tsub sp, sp, r4\n");
40        }
41        else {
42            fprintf(yyout, "\tsub sp, sp, #%d\n", stack_size);
43        }
44    }
45    // 遍历代码块生成代码(广度优先搜索)
46    std::queue<MachineBlock*> q;
47    std::set<MachineBlock*> v;

```

```

47     q.push(block_list[0]);
48     v.insert(block_list[0]);
49     int cnt = 0;
50     while(!q.empty()) {
51         MachineBlock* cur = q.front();
52         q.pop();
53         cur->output();
54         cnt += int(cur->getInsts().size());
55         if(cnt > 160) {
56             fprintf(yyout, "\tb .F%d\n", parent->getN());
57             parent->PrintGlobal();
58             fprintf(yyout, ".F%d:\n", parent->getN()-1);
59             cnt = 0;
60         }
61         for(auto iter : cur->getSuccs()) {
62             if(v.find(iter) == v.end()) {
63                 q.push(iter);
64                 v.insert(iter);
65             }
66         }
67     }
68     // output label .LEND
69     fprintf(yyout, ".L%s_END:\n", this->sym_ptr->toStr().erase(0,1).c_str());
70     // 恢复寄存器和sp fp
71     //2. Restore callee saved registers and sp, fp
72     if(stack_size!=0){
73         if(stack_size > 255) {
74             fprintf(yyout, "\tldr r4,=%d\n", stack_size);
75             fprintf(yyout, "\tadd sp, sp, r4\n");
76         }
77         else {
78             fprintf(yyout, "\tadd sp, sp, #%d\n", stack_size);
79         }
80     }
81     //恢复浮点寄存器
82     if(!fregs.empty()) {
83         fprintf(yyout, "\tvpop {");
84         fregs[0]->output();
85         for (int i = 1; i < int(fregs.size()); i++) {
86             fprintf(yyout, ", ");
87             fregs[i]->output();
88         }
89         fprintf(yyout, "}\n");
90     }
91     fprintf(yyout, "\tpop {");
92     for(auto reg : getSavedRRegs()){
93         reg->output();
94         fprintf(yyout, ", ");

```

```
95     }  
96     fprintf(yyout, "fp, lr}\n");  
97     // 3. bx指令返回  
98     fprintf(yyout, "\tbx lr\n\n");  
99 }
```