南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

编译原理实验报告

上机大作业报告

徐文斌

年级：2020 级

专业：计算机科学与技术

指导教师：王刚

2023 年 1 月 15 日

摘要

通过一个学期的编译原理课程的学习和实践，我和我的搭档孙昊鹏最终成功实现了一个简易的 SysY 编译器，并实现了对数组和浮点数的支持。最终通过了绝大多数测试样例。本文将具体说明我们的 SysY 编译器的总体设计和各模块的设计，并对我个人在整个实验过程中所负责的工作进行一个具体的介绍。

关键字：SysY 编译器、总体设计、负责的工作

目录

一、 SysY 编译器总体设计

首先，从一个较为整体的角度对我们的编译器的执行流程以及各个模块的划分进行一个大致的介绍。编译器执行总体流程如图1所示，大体分为以下几个步骤：

1. 词法分析：首先编译器通过词法分析将输入的源程序字符流识别为单词流并输出到语法分析模块中。
2. 语法/语义分析：语法分析模块根据输入的单词流生成语法树。这里由于我们实现的编译器较为简单，语义分析中并不会包含太多工作。所以我们并没有像一般的编译器那样单独划分出一个语义分析阶段，而是将语义分析与语法分析合并到了一起，即语法分析模块在根据输入的单词流生成语法树的过程中，捎带着进行语义分析。
3. 中间代码生成：中间代码生成阶段，编译器根据上一阶段生成的语法树来生成相应的 LLVM IR 中间代码。
4. 目标代码生成：目标代码生成阶段，编译器将生成的中间代码翻译为对应的 ARM 汇编代码。

在编译器的工作过程中，符号表几乎贯穿着编译器编译的每一个阶段。符号表中存储了关于程序中的变量、函数等一些比较重要的信息，在整个翻译过程中都起到了较为重要的作用。

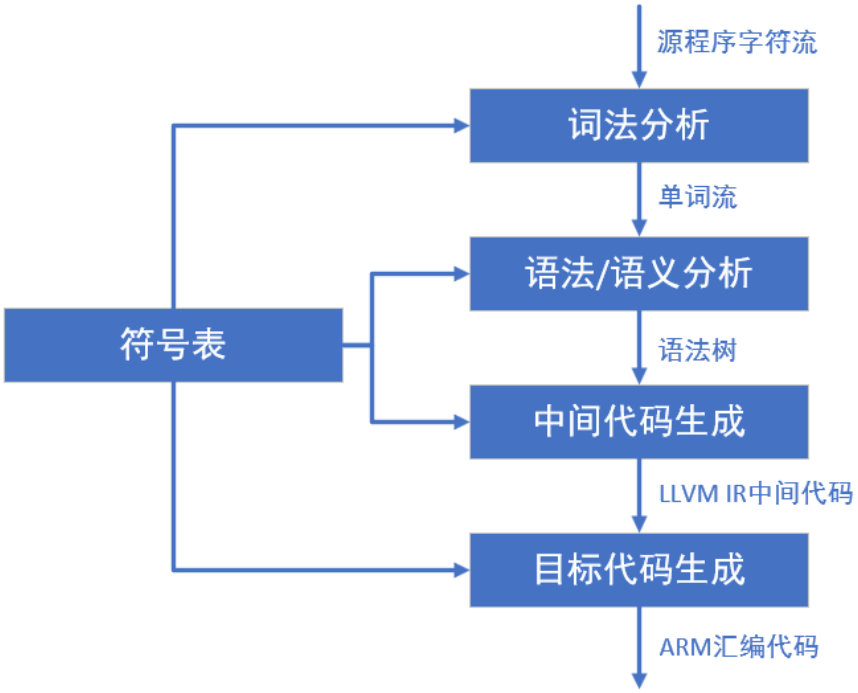

图 1: 编译器执行总体流程

二、 个人分工及各模块设计

首先，说明一下我个人在各个模块中所负责的工作。

1. 词法分析：词法分析方面，本部分的工作较为简单，我们两人均是各自独立完成整个词法分析 lexer.l 文件，然后将二人所作的工作进行了一个筛选和整合。

2. 语法分析：语法分析中，我们首先共同商讨确认了所实现的 Sysy 语言的产生式，然后分工对于产生式进行处理，生成语法树。其中，我主要负责对于变量、常量的声明和初始化，数组的声明和初始化，浮点数的声明和初始化，并实现了对常量和变量作用域的区分。

3. 类型检查：类型检查方面，我主要负责对于变量未声明，及变量在同一作用域下重复声明；条件判断表达式 float、int 至 bool 类型隐式转换，算数表达式中 int 和 float 之间相互的隐式类型转换；数值运算表达式运算数类型是否正确这三种情况进行类型检查工作。

4. 中间代码生成：中间代码生成方面，我主要负责对于变量、常量的声明和初始化，一般语句如块语句、赋值语句、函数调用语句等的翻译。

5. ARM 目标代码生成：目标代码生成方面，我主要负责对于比较指令，控制流指令如 UncondBrInstr、CondBrInstr、RetInstr 指令，getelementptr 指令的翻译。以及对于寄存器分配线性扫描算法中 linearScanRegisterAllocation 等几个函数的实现。

下面，将根据我的具体分工来对各个模块的实现进行一个详细的介绍。

（一） 词法分析

我们依靠 flex 工具来进行词法分析的实现。具体为编写一个 Flex 程序 lexer.l，程序分为定义部分、规则部分、用户子例程三个部分，每个部分之间用两个% 分隔。我们的工作主要是规则部分的编写，规则部分包含模式行与 C++ 代码。当识别到对应的单词之后，便会执行对应的 C++ 代码。

在单词识别部分，对于运算符、关键字以及很多常用符号都直接返回相应的预定义的单词类即可。对于标识符以及整型和浮点型字面值常量来说，我们需要写出它们的正则表达式，并在识别到相应的单词之后通过 yylval 来将字符串或者数值传递给语法分析程序，然后再返回对应的单词类别。这里我们以标识符的识别为例，对于单词的识别进行一个大致的描述。代码如下所示，首先我们在定义段写出标识符的正则表达式，即以字母或下划线开后，后跟若干字母数字或下划线的符号串。然后我们在规则段对其进行识别，在识别到一个 ID 后，首先将具体的字符串通过 yylval 传递给语法分析节点，然后再返回单词类别 ID。

```
// 定义段
ID [[:alpha:]_][[:alpha:][:digit:]_]*

// 规则段
{ID} {
    if(dump_tokens)
        DEBUG_FOR_LAB4(yytext);
    char *lexeme;
    lexeme = new char[strlen(yytext) + 1];
    strcpy(lexeme, yytext);
    lexeme[strlen(yytext)] = 0;
    yylval.strtype = lexeme;
    return ID;
}
```

其余的单词的识别也是类似的，只是对于运算符、关键字等的单词不需要在定义段写出正则表达式。这里有一个特殊的地方，就是 sysy 运行时库的识别，由于 sysy 运行时库的定义并没有出现在源代码中，在何处将这些运行时库的定义添加到符号表中便成了一个需要解决的问题。这

里我们选择在词法分析时处理，即在词法分析时识别到 sysy 运行时库函数后，便将其定义存入符号表中。这里以 getfloat 函数为例，代码如下所示，在识别到符号串"getfloat" 后，我们除了将其通过 yylval 传递给语法分析外，还要新建符号表项，表项中存储对于 getfloat 函数的定义信息，然后将该表项存入到符号表中。这样，当语法分析识别到对于 getfloat 函数的调用后，便可以从符号表项中得到对应的定义，然后生成语法树节点。

```
"getfloat" {
    if(dump_tokens)
        DEBUG_FOR_LAB4(yytext);
    char *lexeme;
    lexeme = new char[strlen(yytext) + 1];
    strcpy(lexeme, yytext);
    yylval.strtype = lexeme;
    Type* funcType = new FunctionType(TypeSystem::floatType, {});
    SymbolEntry* se = new IdentifierSymbolEntry(funcType, yytext, globals->
        getLevel(), true);
    globals->install(yytext, se);
    return ID;
}
```

（二） 语法分析

语法分析阶段，使用 bison 来协助我们进行语法树的生成。我们首先要做的便是 Sysy 语言上下文无关文法的设计，此部分工作由我二人共同商议完成。之后便是设计语法树具体节点的数据结构，设计翻译模式，利用 bison 生成语法分析程序。

下面首先展示语法分析中我主要负责翻译的产生式。我主要负责变量常量声明和初始化相关的一系列产生式的翻译。

$Type \longrightarrow INT \mid FLOAT \mid VOID$

$DeclStmt \longrightarrow VarDeclStmt \mid ConstDeclStmt$

$VarDeclStmt \longrightarrow Type\ VarDefList\ ";"$

$VarDefList \longrightarrow VarDefList\ "," \ VarDef \mid VarDef$

$VarDef \longrightarrow ID \mid ID\ "=" \ InitVal \mid ID\ ArrayIndices \mid ID\ ArrayIndices\ "=" \ InitVal$

$ConstDeclStmt \longrightarrow "const"\ Type\ ConstDefList\ ";"$

$ConstDefList \longrightarrow ConstDefList\ "," \ ConstDef \mid ConstDef$

$ConstDef \longrightarrow ID\ "=" \ ConstInitVal \mid ID\ ArrayIndices\ "=" \ ConstInitVal$

$ArrayIndices \longrightarrow "["\ ConstExp\ "]" \mid ArrayIndices\ "["\ ConstExp\ "]"$

$InitVal \longrightarrow Exp \mid "{"\ "}" \mid "{"\ InitValList\ "}"$

$InitValList \longrightarrow InitVal \mid InitValList\ "," \ InitVal$

$ConstInitVal \longrightarrow ConstExp \mid "{"\ "}" \mid "{"\ ConstInitValList\ "}"$

$ConstInitValList \longrightarrow ConstInitVal \mid ConstInitValList\ "," \ ConstInitVal$

明确了我们需要翻译的产生式之后，我们便需要开始设计所需的数据结构。首先是变量如何表示和存储的问题。显然应该把变量相关的信息存储到符号表中。因此我们设计 IdentifierSym-

bolEntry 符号表项，用于存储关于变量的一些信息，其中要存储变量名、变量的类型、变量的作用域、变量是否初始化 (这是由于 const 变量初始化之后不能再被赋值，这里需要记录一下) 以及变量的值。此外，由于我们实现了数据，因此符号表项中还需要记录数组的初值，这里我使用一个一位数组来记录数组的初值。

根据上述分析，我们便可以得到所需的符号表项的数据结构。如下所示，这里隐去了符号表项类的成员函数。

```
class IdentifierSymbolEntry : public SymbolEntry //变量的类型存储在基类的成员中
{
private:
    enum
    {
        GLOBAL,
        PARAM,
        LOCAL
    };
    std::string name;    // 存储变量名
    int scope;           // 变量的作用域
    double value;        // 变量的值
    bool initial;        // 变量是否初始化
    double *arrayValue; // 数组变量的初值
}
```

设计完符号表项后，我们还需要设计数据结构以表示变量的类型，主要设计整型、浮点型、数组类型的类型表示。对于整型和浮点型的表示比较简单，我们只需要实现相应的继承 Type 的类，并添加一个成员变量用以表示是否为 const int 或 const float 即可。对于数组类型的设计，我的设计方式和课程所讲有一些区别，这里我直接使用一个 vector 来记录数组的各个维度大小，使用一个成员变量来表示数组元素的类型。数组类型类的定义如下，同样隐去成员函数。

```
class ArrayType : public Type
{
private:
    std::vector<int> indexs;  // 记录数组各个维度的值
    Type *baseType;           // 数组元素的类型
}
```

接下来，便需要设计所需的表示语法树节点的数据结构。这里我设计了 Id 类和 DeclStmt 类分别表示变量节点和变量声明语句节点。其中 Id 类的设计较为简单，只需要记录一下该变量对应的符号表项和变量作为数组时的索引值即可。对于 DeclStmt 变量声明节点，其需要一个 Id* 类型的成员变量表示声明的具体变量，还需要一个 ExprNode* 类型的节点用于存储变量声明时的初值，同时对于数组类型变量的声明，我们还需要一个 ExprNode** 类型的变量用于存储数组的初始化表达式，该变量是一个指向元素类型为 ExprNode* 的数组的指针。DeclStmt 的设计具体如下。

```
class DeclStmt : public StmtNode
{
private:
    Id *id;                 // 记录声明的变量
```

```
    ExprNode *expr;          // 声明变量的初值
    ExprNode **exprArray; // 当Id为数组的时候，用来存储数组各个元素的初值
}
public:
    DeclStmt(Id *id, ExprNode *expr = nullptr);
    void setInitArray(ExprNode **exprArray);
```

设计好所需的数据结构之后，我们便需要开始设计翻译模式，依据上述的产生式设计翻译模式。这里对于形如"int a = 10; float b = 1.0;" 这样的变量声明语句的翻译模式较为简单。在识别到相应的产生式之后，我们只需要创建符号表项，存储变量的信息，然后把符号表项加入到符号表中。最后，再根据符号表项以及产生式中已经生成的节点来生成我们需要的 Id 和 DeclStmt 节点即可。在实现过程中遇到的一个难点便是如何把变量的类型这一继承属性给传递到符号表项中。这里我的处理方法是使用一个全局变量保存变量类型这一继承属性的值，当识别到 $VarDef \longrightarrow ID$ 之类的产生式后便可以通过该全局变量把变量的类型信息传递过来。

此外，对于形如"int a, b, c;" 这样的变量声明语句的处理也是一个难点。对于上述语句，我们应该生成三个 DeclStmt 节点，关键是如何把这三个节点插入到语法树中。这里我们把这三个节点给组织成链表的形式，只需把链表头部的节点插入到语法树中，当对语法树进行遍历时，遇到这样的链表也顺序遍历即可。具体翻译模式如下所示。

```
VarDefList
    : VarDefList COMMA VarDef {
        $$ = $1;
        $1->setNext($3);
    }
    | VarDef { $$ = $1; }
    ;
```

对于数组变量声明时数组维度的识别我们也使用上述的链表的形式，即把数组的各个维度给链接成一个链表，只需要遍历链表便可以得到数组具体的维度信息。变量声明部分最后一个比较困难的问题便是数组初始值的识别和存储，上述我们已经说明使用一个 ExprNode* 来存储数组的初始化表达式。但是如何识别数组初始化表达式却是一个问题。这里我们提出了一个基于栈的数组初始化值的识别算法。下面具体介绍一下算法的设计。

首先是数据结构的设计，使用一个维度栈 dimesionStack，其中存储 vector<int> 类型的变量。开辟一个 ExprNode* 类型的一维数组 initArray，用于存储数组的初值表达式，这里即使是多维数组，我们也可以将其展平为一维。定义一个 int 类型的变量 idx，作为 initArray 的索引，将新识别到的表达式赋值给 initArray[idx]。

下面，具体介绍算法的执行流程。

1. 初始化：假设数组定义为 int a[x1][x2][x3]...[xn]，则将 vector {x1 , x2, x3, ..., xn}push 到 dimesionStack 中。将 initArray 中的元素全部初始化为 nullptr。将 idx 初始化为 0。

2. 识别到一个表达式 expr：initArray[idx] = expr;idx++;

3. 识别到一个"{"：取出 dimesionStack 中栈顶的元素 dimesion，向 dimesionStack 中压入 {-1, idx}。将 dimesion[0] 从 dimesion 中删除，如果此时 dimesion 为空，则将 1 放入 dimesion 中。之后将 dimesion 压入 dimesionStack 中。

4. 识别到一个"}"：一直弹栈，直到栈顶元素 dimesion 满足 dimesion[0] == -1，弹出栈顶元素 dimesion，将 idx 赋值为 dimesion[1]。再次弹出栈顶元素 dimesion，将 dimesion 中各

个元素相乘得到 step，id += step。

上述算法经过我的模拟测试，可以正确的将多维数组的初始值表达式给存储到一维数组 initArray 的对应位置处。

最后，简单说明我对于变量的作用域的处理，这里我们使用龙书上提出的一种符号表组织方式，即每次进入一个花括号就新建一个符号表，并在新的符号表中创建一个指向老的符号表的指针，出花括号时恢复老的符号表。这样就形成了一个符号表的树形层次结构，符号表的深度便可以作为变量的作用域的等级，深度越低，对应的变量的作用域越广。这样，当定义一个变量时，我们只需要在当前的符号表中寻找，如果重复则报错。当引用一个变量时，我们顺着符号表指针一直向上寻找即可。这样就实现了变量的作用域。

（三） 类型检查

类型检查的工作较为简单，上文已经提出，由于语义分析较为简单，我们在语法分析生成语法树的时候捎带着进行类型检查，也就是说，类型检查的部分实现在了 parser.y 和 Ast.cpp 语法树节点的构造函数中。下面具体介绍我负责部分的实现。

1. 变量未声明以及变量在同一作用域下重复声明

在我们的实现中，赋值语句以及表达式中的变量都是作为左值 LVal 出现的。对应的产生式如下。

$$LVal \longrightarrow ID \mid ID\ ArrayIndices$$

对于上述产生式，我们只需要从符号表中查询 ID 对应的表项，如果发现表项不存在，则产生了变量未声明的问题。至于变量在同一作用域下重复声明，我们在上一节中已经进行了一个大致的说明，即在上一节变量声明对应的产生式中，首先从当前作用域的符号表中查询 ID 对应的表项，如果发现表项已经存在，则明显出现了变量在同一作用域下重复声明的问题。

如下是具体代码 (只展示了部分)。

```
LVal
    : ID {
        SymbolEntry* se;
        se = identifiers->lookup($1);
        if(se == nullptr) // 产生了变量未声明
        {
            fprintf(stderr, "identifier \"%s\" is undefined\n", (char*)$1);
            delete []$1;
            exit(-1);
        }
        $$ = new Id(se);
        delete []$1;
    }

VarDef
    : ID {
        SymbolEntry* se;
        se = identifiers->lookup($1, true);
        if (se != nullptr) { //变量重复定义了
```

```
            fprintf(stderr, "variable \"%s\" is repeated declared\n", (char*)
                $1);
            delete []$1;
            assert(se == nullptr);
        }
        se = new IdentifierSymbolEntry(recentVarType, $1, identifiers->
            getLevel());
        identifiers->install($1, se);
        $$ = new DeclStmt(new Id(se));
        delete []$1;
    }
```

2. float、int、bool 之间的隐式类型转换

接下来是条件判断表达式 float、int 至 bool 类型隐式转换，算数表达式中 int 和 float 之间相互的隐式类型转换。

隐式类型转化方面，我首先设计了一个语法树隐式类型转化节点 ImplictCastExpr，对于需要类型转化的表达式，我们将原表达式替换为相应的 ImplictCastExpr 表达式。ImplictCastExpr 具体设计如下。

```
class ImplictCastExpr : public ExprNode
{
private:
    ExprNode *expr;
    int op; // 节点类型

public:
    enum
    {
        BTI, // bool转int
        ITB, // int转bool
        FTI, // float转int
        ITF, // int转float
        BTF, // bool转float
        FTB  // float转bool
    };
    ImplictCastExpr(ExprNode *expr, int op);
    double getValue();
    void output(int level);
    void typeCheck(){};
    void genCode();
};
```

首先讲述条件判断表达式 float、int 至 bool 类型隐式转换，该隐式类型转化主要出现在两处，一处是二元运算表达式中操作符为 && 和 || 时，如果操作数为 int 或 float 类型，需要将其隐式类型转换为 bool 类型；第二处是在 if、while 语句中如果 cond 为 int 或 float 类型，需要将其隐式类型转换为 bool 类型。具体代码如下 (仅展示部分)。

```
// 对于AND和OR逻辑运算，如果操作数表达式不是bool型，需要进行隐式转换，转为bool型。
if (op == BinaryExpr::AND || op == BinaryExpr::OR)
{
    if (type1->isInt())
        this->expr1 = new ImplictCastExpr(expr1, ImplictCastExpr::ITB);
    else if (type1->isFloat())
        this->expr1 = new ImplictCastExpr(expr1, ImplictCastExpr::FTB);
    if (type2->isInt())
        this->expr2 = new ImplictCastExpr(expr2, ImplictCastExpr::ITB);
    else if (type2->isFloat())
        this->expr2 = new ImplictCastExpr(expr2, ImplictCastExpr::FTB);
}

// If语句中cond为float或int类型
IfStmt::IfStmt(ExprNode *cond, StmtNode *thenStmt) : cond(cond), thenStmt(thenStmt)
{
    if (cond->getType()->isInt())
        this->cond = new ImplictCastExpr(cond, ImplictCastExpr::ITB);
    else if (cond->getType()->isFloat())
        this->cond = new ImplictCastExpr(cond, ImplictCastExpr::FTB);
}
```

接下来是算数表达式中 int、float 和 bool 之间相互的隐式类型转化。在二元表达式中，如果操作符为加减乘除且操作数中有 float 类型操作数，则需要将所有不是 float 类型的操作数隐式转化为 float 类型。此外，如果运算符是比较运算符，也应该做如此处理。这一部分的代码和上方代码是非常相似的，这里不做过多的展示。除了算数表达式中进行隐式类型转换之外，我们还需要在赋值语句、函数返回语句、函数调用语句等多处进行隐式类型转换，代码和上方代码都十分相似，这里均不做过多展示。

3. 数值运算表达式运算数类型是否正确

最后，对检查数值运算表达式运算数类型是否正确进行大致描述。在 Sysy 中，数值运算表达式运算数类型不正确大致出现在运算表达式中出现了 void 类型的函数调用。因此，这里我们只需要在表达式节点的构造函数中进行判断即可，如果出现类型为 void 的操作数，则类型检查出错。如下二元表达式中的处理为例。

```
if (expr1->getType()->isVoid() || expr2->getType()->isVoid())
{
    fprintf(stderr, "invalid operand of type \'void\' to binary \'opeartor%s\'\n", op_type.c_str());
}
```

（四） 中间代码生成

中间代码生成部分，我们需要将语法树的节点翻译为 LLVM IR 中间代码。翻译的框架实验已经给出。使用 Unit 类表示翻译单元、Function 类表示单个函数、BasicBlock 类表示基本块、

Instruction 类表示 IR 中间代码指令、Operand 类表示指令中的操作数，如立即数、寄存器、标签等。其中，一个 Unit 中可以有多个 Function，一个 Function 中可以有多个 BasicBlock，一个 BasicBlock 中可以有多个 Instruction，一条 Instruction 中可以有多个 Opernad。使用一个 IRBuilder 类来协助翻译，记录翻译过程中当前的基本块。实验原有代码已经给的比较充分，我们只需要添加若干指令，对原有框架进行一些小的改动即可，这部分由我二人商议完成，在此不做过多说明。

我们要做的工作主要便是编写语法树节点的 genCode 代码，将整个语法树翻译为中间代码。这里我主要负责对变量常量声明语句 DeclStmt、块语句 CompoundStmt、顺序语句 SeqNode、赋值语句 AssignStmt、隐式类型转换节点 ImplictCastExpr、函数定义 FunctionDef、函数调用 CallExpr 节点的翻译。下面将对它们进行一个详细的说明。

1. 变量声明语句 DeclStmt

首先是变量和常量声明语句 DeclStmt 节点的翻译。在对节点进行翻译之前，我们需要对符号表项的设计进行完善，对于 IdentifierSymbolEntry 符号表项，我们需要在其中添加存储变量存储地址的 addr 变量。同时对于函数的形参来说，在函数参数传递时我们需要一个额外的临时寄存器，因此这里对于这种情况的处理，我们也在 IdentifierSymbolEntry 符号表项中添加一个用于函数传参的 Operand 操作数。

除此之外，为了实现数组功能，我们还需要添加 GepInstruction 类，表示 getelementptr 指令。这里我们先介绍一下 getelemetnptr 指令的作用。对于一个形如”int a[4][5];” 的数组，如果我们想要访问 a[2][3]，我们需要根据 a 的基地址和索引生成对应元素的指针。这个过程可以通过一条 gep 指令来实现，如下所示，假设数组 a 的基地址存储在寄存器%t0 中，我们只需要如下一条 gep 指令便可以计算出 a[2][3] 元素的地址，%t1 中存储的便是 a[2][3] 元素地址。

```
%t1 = getelementptr inbounds [4 x [5 x i32]], [4 x [5 x i32]]* %t0, i32 0,
    i32 2, i32 3
```

对于该类的实现,我们只需要一个 Operand* 来存储计算的数组基地址以及一个 vector<Operand*> 来存储相对于基地址的偏移即可。GepInstruction 类构造函数如下，其中，base 为基地址，offs 存储了数组索引。

```
GepInstruction::GepInstruction(Operand *dst, Operand *base, std::vector<
    Operand *> offs, BasicBlock *insert_bb, bool type2) : Instruction(GEP,
    insert_bb), type2(type2)
{
    operands.push_back(dst);
    operands.push_back(base);
    dst->setDef(this);
    base->addUse(this);
    for (auto off : offs)
    {
        operands.push_back(off);
        off->addUse(this);
    }
}
```

接下来讲解具体的翻译过程，由于涉及的情况较多，DeclStmt 节点的中间代码翻译较为复杂。首先，我们得到定义的变量的符号表项，然后判断其是否为全局变量，由于全局变量不属于

任何一个函数，我们无法将对于全局变量的初始化放到任何一个函数中。因此，这里我们在 Unit 类中添加 global _vars 变量，用于记录全局变量，在 Unit 的 output 函数中将全局变量的声明打印出来。对于全局变量，我们首先赋值其符号表项中的 addr 变量，然后将其符号表项添加到 Unit 的 global_vars 变量中。此外，我们还需要考虑全局数组，对于全局数组我们则根据语法分析中得到的数组初始值表达式数组，来赋值该变量的符号表项中的数组初始值部分。

接下来是局部变量和函数参数的声明的翻译。对于局部变量和函数参数来说，我们需要为他们在栈中开辟空间，因此我们首先需要生成 Alloca 指令用于生成空间。并将变量的存储地址赋值到符号表的 addr 中。之后我们需要判断当前变量是否是函数形参，如果是，则要为其生成一条 Store 指令。对于局部变量来说，我们则要判断该变量是否有初始值，如果有初始值，对于非数组的局部变量，我们直接调用其初始值表达式的 genCode 函数，然后生成一条 Store 指令将初始值存入变量中即可。对于数组类型的变量，我们则需要对每一个数组元素进行初始值的赋值，这里对于数组清零操作，我们没有选择调用函数，这里我是遍历数组的每一个元素，然后根据 DeclStmt 中存储的数组的初始值表达式数组来对数组进行初始化。具体为首先通过 getelementptr 指令计算处数组的基地址，然后遍历数组的初始值表达式数组，如果对应的表达式不为空，则调用其 genCode 函数，并生成 Store 指令初始化数组元素；如果表达式为空，则生成 Store 指令将 0 赋值给当前数组元素，每遍历一次，就通过 getelementptr 指令将数组的指针加一。

最后，我们之前提到，我们将 DeclStmt 组织成了链表的形式，因此在 DeclStmt 的 genCode 函数的最后，我们需要调用链表中下一个节点的 genCode 函数。

2. 块语句 CompoundStmt 和 SeqNode

接下来说明块语句 CompoundStmt 和顺序语句 SeqNode 的翻译。对于块语句来说，其只有一个子节点，子节点往往是 SeqNode 语句或者单条语句。因此，块语句的 genCode 中，我们只需调用其子节点的 genCode 函数即可。SeqNode 也是类似的，SeqNode 节点有两个子节点，这两个子节点是顺序的语句。SeqNode 第一个子节点是 SeqNode 节点或者单条语句节点，其第二个节点是单条语句节点。因此，SeqNode 节点的 genCode 中，我们只需要依次调用其两个子节点的 genCode 函数即可。

3. 赋值语句 AssignStmt 节点

下面具体说明 AssignStmt 赋值节点的翻译。AssignStmt 赋值节点由一个 ExprNode 类型的左值和一个 ExprNode 类型的赋值表达式组成，其中，左值便是 Id 节点。该节点具体翻译过程如下，首先调用赋值表达式的 genCode 函数生成其 IR 代码，然后得到左值变量的符号表项，根据符号表项中存储的变量的类型信息判断变量是否是数组。如果不是数组变量，则从符号表项中得到变量的存储位置，并生成一条 Store 指令把表达式的结果存入变量中即可。如果变量是数组类型的变量，则首先我们要生成对应数组元素的地址，具体为根据 Id 节点中存储的对于该数组变量的索引来生成一条 getelementptr 指令，得到数组元素的地址，然后生成 Store 指令将表达式的结果存入对应地址处即可。

4. 隐式类型转换 ImplictCastExpr 节点

接下来说明隐式类型转换节点 ImplictCastExpr 的翻译。首先，我们调用被进行类型转换的表达式的 genCode 函数，生成其中间代码。然后我们根据具体的转换类型来进行相应的代码生成。对于 bool 转 int，我们只需要生成一条 zext 指令即可；对于 int 转 bool，我们生成一条 cmp 指令，将 int 值与 0 做不等的比较；对于 float 转 int，我们需要生成一条 fptosi 指令；对于 int 转 float，我们需要生成一条 sitofp 指令；对于 float 转 bool，我们同样只需生成一条 cmp 指令，

将 float 值与 0 做不等的比较即可；对于 bool 转 float，这里我们首先生成一条 zext 指令，将 bool 转为 int，然后生成一条 sitofp 指令，将 int 转为 float。

5. 函数定义节点 FunctionDef

接下来说明函数定义节点 FunctionDef 的翻译，一个函数定义节点 FunctionDef 由函数的形参 DeclStmt、函数体 StmtNode 组成。我们在翻译 FunctionDef 节点时，首先判断函数是否有参数，如果有参数，则在函数的入口基本块中生成参数 DeclStmt 节点的 IR 代码，然后我们调用函数体的 genCode 代码即可。但是，此时我们的工作并没有完成，因为我们在翻译中间代码的时候对于每一个基本块来说，我们并没有确定它们的前驱和后继关系，而是将它们作为一个个独立的基本块进行操作。所以在进行完函数体的翻译之后，我们需要建立起基本块的流图，以便于对基本块进行更进一步的操作。该过程也比较简单，我们知道，一个基本块只有在基本块的开头会有跳转标签，只有在基本块的末尾才会有分支跳转指令，我们便可以利用基本块末尾的分支跳转指令来构建基本块的控制流图。具体做法为，遍历函数的基本块列表，对于遍历到的基本块，我们取出它的最后一条指令，如果最后一条指令是分支指令，我们可以很轻易地根据分支指令的目的基本块来设置前驱和后继关系；如果最后一条指令不是分支指令，则其有可能是函数返回指令，这种情况不需要处理；如果最后一条指令既不是分支指令也不是函数返回指令，那么不难推断出该基本块是函数的出口基本块，我们需要在基本块的末尾根据函数的返回值类型添加一条函数返回指令。完成上述操作后，我们便建立起了基本块之间的前驱和后继关系。

6. 函数调用 CallExpr 节点

我所负责的最后一部分是函数调用 CallExpr 节点的翻译，该节点由一个存储了函数实参表达式的 vector 组成，在 CallExpr 的 genCode 函数中，我们只需要遍历其每个实参，调用实参的 genCode 函数，然后生成一条 LLVM IR call 指令即可。

中间代码生成部分的代码比较多，这里不做过多展示，请参看文末链接以了解具体的代码实现。

（五） 目标代码生成

目标代码生成的框架和中间代码生成的框架比较类似。MachineUnit 类表示编译单元，MachineFunction 表示函数，MachineBlock 表示基本块，MachineInstruction 表示汇编指令，MachineOperand 表示汇编操作数，如寄存器、立即数、标签等，它们的组织方式和中间代码较为相似。此外，使用一个 AsmBuilder 来协助 arm 汇编代码的翻译，用于记录当前翻译的函数、基本块等信息。我们主要需要做的工作便是实现中间代码各个 Instruction 类的 genMachineCode 函数，同时补全线性扫描 LinearScan.cpp 中所需的若干函数。还有一些琐碎的工作这里就简单略过了。在本部分中我主要负责对于 LLVM IR 中间代码中 cmp、uncondbr、condbr、ret、getelementptr 指令向 arm 汇编代码的翻译，以及补全线性扫描算法中空缺的 linearScanRegisterAllocation、expireOldIntervals、spillAtInterval、genSpillCode 函数。下面对于我所作的工作进行一个具体的描述。

1. CmpInstruction 翻译

CmpInstruction 翻译方面，比较关键的一个问题便是 LLVM IR 中的 cmp 指令有返回值寄存器，condbr 指令根据 cmp 指令的结果寄存器的值来判断是否跳转。而在 arm 汇编中，汇编指令 cmp 并没有显式的结果寄存器，该指令的结果直接影响着 arm 架构中 flag 寄存器的某些标

志位。汇编指令中的分支指令则是根据 flag 寄存器中的标志位来判断是否进行跳转。如下所示，下方代码中上侧的 LLVM IR 中间代码将会被翻译为下侧的 arm 汇编代码的形式。

```
%t2 = icmp slt i32 %t0, %t1
br i1 %t2, label %B0, label %B1 ;两条指令间通过%t2传递比较结果

cmp r0, r1
blt B0 ;两条指令间通过flag寄存器传递比较结果
b B1
```

因此，我们需要考虑如何把 IR 代码中 cmp 指令的比较操作码传递给与它相匹配的 condbr 指令。这里我们可以通过 AsmBuilder 进行传递，在翻译的过程中，AsmBuilder 对象是保持不变的，我们可以在 AsmBuilder 类中添加一个成员变量用于 cmp 指令比较操作码的传递。

接下来，具体描述 cmp 指令的翻译，首先调用 genMachineOperand 函数生成 cmp 指令的两个汇编操作数，然后判断操作数是否为立即数，因为 arm 中 cmp 的操作数不能为立即数，这里我们需要把立即数先放到寄存器中，对于整型立即数，我们要把它移动到通用寄存器中；对于浮点型立即数，我们要把它移动到浮点寄存器中。之后，我们便可以根据指令的类型生成对应的 arm cmp 指令。这里对于整型的比较指令，我们只需生成一条 cmp 指令即可，但是对于浮点类型的比较指令，我们首先需要生成一条 vcmp 指令，我们还需要生成一条 vmrs 指令将 fpscr 中的标志位加载到 apsr 中，才能进行比较指令的跳转。最后，我们要将当前 cmp 指令的比较码记录到 AsmBuilder 中，从而向之后的条件跳转指令传递。

2. UncondBrInstruction 和 CondBrInstruction 翻译

对于无条件跳转和条件跳转指令的翻译较为简单。对于无条件跳转来说，我们只需要生成一条对应的 b 指令即可。对于条件跳转来说，我们首先需要从 AsmBuilder 中获得上一个 cmp 指令的比较码，然后生成一条条件码为得到的比较码的指向真分支的条件跳转指令和一条指向假分支的无条件跳转指令。条件跳转指令的翻译如下。

```
void CondBrInstruction::genMachineCode(AsmBuilder *builder)
{
    auto cur_block = builder->getBlock();
    cur_block->InsertInst(new BranchMInstruction(cur_block,
        BranchMInstruction::B, genMachineLabel(true_branch->getNo()), builder
        ->getCmpOpcode()));
    cur_block->InsertInst(new BranchMInstruction(cur_block,
        BranchMInstruction::B, genMachineLabel(false_branch->getNo())));
}
```

3. RetInstruction 翻译

对于函数 return 语句的翻译，我们首先判断 ret 语句是否有返回值，如果有返回值，则需要根据函数的返回类型来将返回值给移动到 r0 寄存器 (整型) 或 s0 寄存器 (浮点型) 中。然后，我们需要释放函数的栈空间，只需要简单的生成一条 mov 指令即可，将 fp 的值移动到 sp 中。此外，由于在函数入口处会对函数中用到的寄存器进行入栈保存，这里我们在函数返回时也需要把保存的寄存器给弹出栈，这里需要生成两条指令，分别是 VPOP 指令和 POP 指令，前者用于恢复保存的 s 寄存器，后者用于恢复保存的 r 寄存器。但是，在指令翻译的时候，由于没有进行线性扫描分配物理寄存器，这里我们并不知道 VPOP 指令和 POP 指令的具体要恢复的寄存器。

我们的处理方式是在 MachineBlock 中添加一个 vector<MachineInstruction*> 类型的变量，此变量专门用于记录上述提到的这些在翻译时参数无法确定的指令，当进行完线性扫描后，打印指令时，我们可以使用类似用中间代码控制流翻译时的回填技术，将指令所需的参数给填入指令合适的位置。这样便可以打印出正确的指令。最后，我们要生成一条 bx 指令，用于返回函数被调用处的地址。

4. GepInstruction 翻译

接下来是 getelementptr 指令的翻译，该指令是实现数组过程中比较重要的一条指令。我们在中间代码生成部分已经对于 gep 指令的作用和形式进行了一个大致的描述，这里直接说明其翻译方式。首先我们得到数组的基地址，对于全局变量，我们需要使用 ldr 指令来将其地址 load 到寄存器中；对于局部变量，我们则需要使用一条 add 指令将 fp 加上局部变量的栈内偏移，得到数组基地址。得到数组的基地址之后，我们便需要根据 gep 指令中存储的数组的索引来计算数组元素地址相对于数组基地址的偏移，具体为首先得到数组的维度信息 indexs，然后遍历 gep 指令中数组的索引值，使用 mul 指令计算出当前索引值造成的偏移值，并生成 add 指令将偏移值添加到数组基地址中，这样遍历所有的数组索引即可。对于上述过程，我们还可以进一步的优化，这里对于立即数索引我们并不需要为其生成 mul 指令，而是编译的时候便可以得到它造成的偏移值，这样我们可以先不处理立即数索引，将它们全部存储起来，最后再计算所有立即数索引造成的偏移值，使用 add 指令将其加到基地址中即可。

5. linearScanRegisterAllocation 函数实现

关于线性扫描的实现，由于我们实现了浮点数功能，我们需要对 Interval 结构体进行修改，在结构体中添加 fpu 变量，表示该区间对应的虚拟寄存器是一个浮点寄存器。此外，我们在 LinearScan 类中添加 vector<int> 类型变量 fpuRegs 用于记录可用的浮点寄存器。此外，由于 active 需要按照区间的结束时间升序排序，我们这里定义函数 compareEnd 来作为 sort 函数的参数。compareEnd 函数如下所示。

```
bool LinearScan::compareEnd(Interval *a, Interval *b)
{
    return a->end < b->end;
}
```

接下来具体说明 linearScanRegisterAllocation 函数的实现。该函数的作用就是扫描一趟所有的 interval，并未每个 interval 分配物理寄存器。首先我们将 active 列表清空，初始化 regs 和 fpuRegs 向量，向 regs 中填入 4 到 10，向 fpuRegs 中填入 16 到 31。接下来开始遍历所有的 intervals，对于每一个 interval，我们首先调用 expireOldIntervals 函数将 active 中所有与当前 interval 不相交的区间占用的寄存器给释放掉，腾出物理寄存器分配给当前 interval。然后我们判断 regs 和 fpuRegs 中是否有 interval 所需的寄存器，如果没有，则要调用 spillAtInterval 函数进行溢出操作，并将函数返回结果置为 false；如果有所需的寄存器，我们只需要将寄存器从对应的列表中删除，并分配给当前的 interval，然后将 interval 添加到 active 中。函数具体代码如下所示。

```
bool LinearScan::linearScanRegisterAllocation()
{
    active.clear(); // 清空active
    regs.clear();
    fpuRegs.clear();
```

```
    for (int i = 4; i < 11; i++) // 初始化regs
        regs.push_back(i);
    for (int i = 16; i <= 31; i++) // 初始化fpuRegs
        fpuRegs.push_back(i);
    bool result = true;
    for (auto interval : intervals)
    {
        expireOldIntervals(interval); // 将与当前interval不冲突的区间释放掉
        if ((interval->fpu && fpuRegs.empty()) || (!interval->fpu && regs.
            empty()))
        {
            spillAtInterval(interval); // 没有可用物理寄存器，溢出到栈中
            result = false;
        }
        else
        {
            if (interval->fpu) // 有可用物理寄存器，分配即可
            {
                interval->rreg = fpuRegs[0];
                fpuRegs.erase(fpuRegs.begin());
            }
            else
            {
                interval->rreg = regs[0];
                regs.erase(regs.begin());
            }
            active.push_back(interval);
            sort(active.begin(), active.end(), compareEnd);
        }
    }
    return result;
}
```

6. expireOldIntervals 函数实现

expireOldIntervals 函数的作用为将所有与 interval 不冲突的区间所占用的寄存器资源释放掉。我们只需要遍历整个 active 列表，对于遍历到的 active 中的 activeInterval ，如果其结束位置早于 interval 的开始位置，则可以将该 activeInterval 占用的资源释放掉，将其使用的物理寄存器放入 regs 或 fpuRegs 中，并从 active 中删除 activeInterval。否则就不能释放当前的 activeInterval 区间。具体代码如下。

```
void LinearScan::expireOldIntervals(Interval *interval)
{
    std::vector<Interval *>::iterator it = active.begin();
    while (it != active.end())
    {
        Interval *actInterval = *it;
        if (actInterval->end >= interval->start) // 由于active列表根据end升序
```

```
            排序，这里一出现冲突的情况，我们便可以结束遍历
        {
            break;
        }
        if (actInterval->fpu)
            fpuRegs.push_back(actInterval->rreg);
        else
            regs.push_back(actInterval->rreg);
        it = active.erase(it);
    }
}
```

7. spillAtInterval 函数实现

此函数的实现也较为简单，这里我们首先逆向扫描整个 active，找到寄存器属性和 interval 所需寄存器属性相同的一个区间。然后判断这两个区间的结束位置，我们将结束比较晚的区间溢出到寄存器中。具体代码如下所示。

```
void LinearScan::spillAtInterval(Interval *interval)
{
    auto it = active.rbegin();
    for (; it != active.rend(); it++)
    {
        if ((*it)->fpu == interval->fpu) // 由于实现了浮点数，这里需要找到一个与interval寄存器属性相同的区间
            break;
    }
    if ((*it)->end > interval->end) // 哪个结束的晚就把哪个溢出到栈中
    {
        interval->rreg = (*it)->rreg;
        (*it)->spill = true;
        spillIntervals.push_back(*it); // spillIntervals记录所有需要溢出的区间
        active.erase((++it).base());
        active.push_back(interval);
        sort(active.begin(), active.end(), compareEnd);
    }
    else
    {
        interval->spill = true;
        spillIntervals.push_back(interval);
    }
}
```

8. genSpillCode 函数实现

最后是 genSpillCode 函数的实现，此函数实现较为简单。这里其实每一个区间对应着一个虚拟寄存器，对于 spillAtInterval 函数中所产生的需要溢出到栈中的区间，我们首先为其在栈中分

配四字节的空间，然后遍历使用该寄存器的指令，并在对应的指令之前添加 Load 指令将存储在栈中的值加载到对应的寄存器中；然后遍历定义该寄存器的指令，在对应的指令之后添加 Store 指令将寄存器中的值存储到栈中即可。

三、 实验结果与总结

图2为我们的编译器最终的测试结果。经过一整个学期的学习和实验，我和孙昊鹏二人成功地完成了一个 Sysy 编译器，并实现了数组、浮点数等进阶特性。通过本学期的实验，我们对于编译器的架构和原理有了一个清晰的认识，学到了许多编译器相关的知识，掌握了如 flex、bison、gdb、linux 等工具的应用方法。此外，对于编译器的构建也使我们对于 ARM 体系结构有了一个大致的了解，对于编程语言的一些特性有了更进一步的理解。

但是，我们的编译器仍有许多不足，比如对于数组的初始化会产生很多冗余代码；有时会产生一些没有必要的指令；以及对于寄存器的分配还有待进一步的优化等等。我们会继续学习，以解决这些问题。

		通过样例	总数	满分	得分		优化算法		满分	得分
	level1-1	39	60	4	4.00		寄存器分配		1	1
	level1-2	21					mem2reg		1	0
	level2-1	4	14	1	1.00		优化算法1		1	0
	level2-2	5					其余优化算法			0
	level2-3	5								
	level2-4	40	40	1.5	1.50					
	level2-5	25	26	0.5	0.48					
	level2-6	11	11	1	1.00		满分	11		
		150	151		7.98		总分	8.98		

图 2: 编译器执行总体流程

最后，在此由衷地感谢王刚老师以及各位助教学长在本学期为同学们的付出，如果不是老师和各位学长的帮助和指导，我们本学期的学习和实验一定不会进行得如此开心顺利。

四、 项目链接

https://gitlab.eduxiji.net/your-words-all-right/compilation-principle/-/tree/master/%E6%9C%BA%E5%99%A8%E4%BB%A3%E7%A0%81%E7%94%9F%E6%88%90-%E7%A1%AC%E6%B5%AE%E7%82%B9