



南開大學  
Nankai University

计算机学院  
编译系统原理 Lab2

定义你的编译器 & 汇编编程

姓名：李威远 曹珉浩  
学号：2112338 2113619  
专业：计算机科学与技术

2023 年 10 月 10 日

## 摘要

本次实验主要分为两个部分：确立 SysY 语言特性并给出其形式化定义、实现 SysY 程序的 ARM 汇编编程

在第一部分的实验工作中，我们考察 SysY 官方给出的 Sysy 定义文档，确立下来了包含**数据类型、常变量、运算语句、分支循环语句、赋值语句、注释、数组、函数**等等需要实现的语言特性，并深入研究这些语言特性，对其进行了详细的分析。随后，我们参考教材第 2 章、SysY 官方文档等相关资料，针对先前分析所得的语言特性，依次采用上下文无关文法，成功地自上而下地构建了这些语言特性组成的 SysY 语言子集的形式化定义设计。

在第二部分的实验工作中，我们根据第一部分分析的语言特性作为 SysY 程序选择的切入点，将这些不同语言特性归纳为**赋值与算术运算、数组声明与其运算、函数定义及其调用、分支语句与关系运算、循环语句与逻辑运算**五个方向，根据其对应的 SysY 程序来设计不同的五个 ARMv7 程序，以考察其语言特性，最终都成功通过编译，正常运行得到编译结果，顺利验证了这些语言特性在 ARMv7 程序上的实现。

此外，我们还对 SysY 转 ARMv7 汇编程序设计展开了讨论与思考，确立下来采取类似编译的不同阶段分析的设计思路，从词法分析、语法分析等方面依次展开设计，并在不同阶段采取不同的数据结构与算法，从而实现这个编译程序。

本实验中所用到的代码和中间文件，已经上传至[GitHub](#)。

**关键字：**语言特性、CFG 设计、ARMv7、SysY、语法分析

# 目录

|          |                           |           |
|----------|---------------------------|-----------|
| <b>1</b> | <b>SysY 语言特性及形式化定义</b>    | <b>3</b>  |
| 1.1      | 编译器支持的 SysY 语言特性          | 3         |
| 1.2      | CFG 描述 SysY 语言子集          | 4         |
| 1.2.1    | 终结符集合 $V_T$ 、非终结符集合 $V_N$ | 4         |
| 1.2.2    | 产生式集合 $\mathcal{P}$       | 5         |
| 1.2.3    | 开始符号 $S$                  | 9         |
| <b>2</b> | <b>ARM 汇编编程</b>           | <b>9</b>  |
| 2.1      | armv7-a 汇编学习              | 9         |
| 2.2      | 赋值与算术运算                   | 11        |
| 2.3      | 数组声明与运算                   | 14        |
| 2.4      | 函数定义及其调用                  | 15        |
| 2.5      | 分支语句与关系运算                 | 17        |
| 2.6      | 循环语句与逻辑运算                 | 19        |
| <b>3</b> | <b>思考题</b>                | <b>20</b> |
| <b>4</b> | <b>分工情况</b>               | <b>21</b> |

# 1 SysY 语言特性及形式化定义

## 1.1 编译器支持的 SysY 语言特性

- **数据类型:** 对于我们要实现的 SysY 语言编译器, 其本质上是 c 语言的子集, 考察其官方文档 [3], 可以认识到, 在数据类型上, SysY 实现了 C 语言中 int(32 位有符号数) 和 float(32 位单精度浮点数) 的类型, 以及元素为 int/float 类型且按行优先存储的多维数组类型。此外, SysY 还实现了 int 和 float 类型之间的隐式类型转换和 const 修饰常量的功能。

若想定义某个数据类型的变量, SysY 语法一般是: `type id1,id2,...`; 其形式化定义为:

$decl \rightarrow type\ idlist$

$type \rightarrow \text{int} \mid \text{float}$

$idlist \rightarrow idlist\ id$

- **常量, 变量:** SysY 允许在一个语句中声明一个或多个变量或者常量, 声明常量需要用 const 修饰。声明时可以带初始化表达式。所有变量/常量要求先定义再使用。在函数外声明的为全局变量/常量, 在函数内声明的为局部变量/常量。

变量一般为标识符 (identifier), 形式化定义为 *id*, 常量声明的形式化定义可抽象为如下产生式:

$ConstDecl \rightarrow \text{const}\ type\ ConstDef$

$ConstDef \rightarrow id = ConstInitVal$

- **基本表达式语句:** SysY 保留了 if 语句和 while 语句, 实现了分支和循环的功能, 同时, 实现了 continue 和 break 关键字, 以实现对循环的控制。此外, 基本表达式还包括赋值语句, 基本算术运算 (+, -, \*, /) 和基本逻辑运算 (>, <, ==, >=, <=) 等。

if 语句可以形式化定义为: `if(expr) stmt else stmt`

while 语句和其相似, 形式化定义为: `while(expr) stmt`

运算语句的形式很灵活, 最常见的一种是: `id1 = id2 operator id3`, 其形式化定义为:

$id1 \rightarrow id2\ id3\ operator$

$operator \rightarrow + \mid - \mid * \mid / \mid == \mid != \mid > \mid < \mid >= \mid <= \mid \&\& \mid \mid$

- **注释:** 实现的 SysY 编译器可以在预处理阶段删除注释
- **数组:** 进阶要求的 SysY 编译器可以实现数组, 以整形数组为例, 可形式化定义为:

$array \rightarrow [ \ elements ]$

$elements \rightarrow element\ ",\ " \ elements \mid element$

$element \rightarrow number \mid array$

$number \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

- **函数:** SysY 基本实现了 C 语言中函数的功能, 返回类型支持 int/float 类型和不返回, 而参数则可以接收 int/float 类型及它们相应的数组类型, 其中, 前者是按值传递, 后者则是传递数组的起始地址, 函数的形式化定义可以为:

$function \rightarrow function\_name\ "(" \ parameters \ ")" \ return\_type \ "" \ statements \ ""$

$function\_name \rightarrow identifier$   
 $parameters \rightarrow parameter \mid parameter \text{ }, parameters$   
 $parameter \rightarrow identifier \text{ } : \text{ } type$   
 $return\_type \rightarrow type$   
 $statements \rightarrow statement \mid statement statements$   
 $statement \rightarrow assignment \mid if\_statement \mid return\_statement \mid \dots$   
 $type \rightarrow \text{int} \mid \text{float}$   
 $identifier \rightarrow [a-zA-Z\_][a-zA-Z0-9\_]*$

## 1.2 CFG 描述 SysY 语言子集

上下文无关文法 (CFG) 是一种描述程序语言语法的表示方式, 可视为一个四元组  $(V_T, V_N, P, S)$ :

- 终结符号集合  $V_T$ : 单词, 程序语言的基本符号的集合
- 非终结符号集合  $V_N$ : 语法变量, 每个非终结符号表示一个终结符号串的集合
- 产生式集合  $P$ : 定义语法范畴, 产生式的形式为  $A \rightarrow \alpha$ , 其中左部为一个非终结符, 右部为由终结符和非终结符组成的串
- 一个开始符号  $S$ : 一个特定的非终结符

在描述文法时, 将数位, 符号和黑体字符串看作终结符号, 将斜体字符串看作非终结符号。[\[1\]](#) 下面, 我们将分别从这四个方面入手, 构造 CFG 来描述我们的 SysY 语言子集:

### 1.2.1 终结符集合 $V_T$ 、非终结符集合 $V_N$

如上所述, 终结符是程序语言的基本符号的集合, 大致可以分为几类: 关键字, 标识符, 数值常量, 运算符以及包括括号等的标点符号。而每个非终结符是一个终结符号串的集合, 定义好终结符之后, 按照一定的语法规则就可以组合成合法的非终结符。

- 关键字 (keyword):

$keyword \rightarrow \text{int} \mid \text{float} \mid \text{void}$   
 $\rightarrow \text{if} \mid \text{else} \mid \text{for} \mid \text{while} \mid \text{switch} \mid \text{case} \mid \text{default}$   
 $\rightarrow \text{return} \mid \text{const} \mid \text{break} \mid \text{continue} \mid \text{sizeof}$

- 标识符 (identifier): 采用递归定义

$identifier \rightarrow nodigit$   
 $\rightarrow identifier nodigit$   
 $\rightarrow identifier digit$   
 $nodigit \rightarrow \_ \mid a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m$   
 $\rightarrow n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z$   
 $\rightarrow A \mid B \mid C \mid D \mid E \mid F \mid G \mid H \mid I \mid J \mid K \mid L \mid M$   
 $\rightarrow N \mid O \mid P \mid Q \mid R \mid S \mid T \mid U \mid V \mid W \mid X \mid Y \mid Z$   
 $digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

- **常量 (constant):** 数值常量可以按照整型常量和浮点常量进行分类, 在这两类当中又可按照二进制, 八进制, 十进制, 十六进制分类:

```

constant → int_constant
          → float_constant
int_constant → binary_constant
              → octal_constant
              → dec_constant
              → hex_constant
binary_constant → 0b1 | 0b1 binary_constant
octal_constant → 0 oct_nonzero | 0 oct_nonzero octal_constant
dec_constant → nonzero | nonzero dec_constant
hex_constant → 0x hex_nonzero | 0x hex_nonzero hex_constant
oct_nonzero → 1 | 2 | 3 | 4 | 5 | 6 | 7
nonzero → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
hex_nonzero → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F
float_constant → fractional_constant [exponent_part] [floating_suffix]
               → int_constant exponent_part [floating_suffix]
fractional_constant → [int_constant].[int_constant]
                   → int_constant
exponent_part → e [sign] digit_sequence
sign → + | -
floating_suffix → F | L | f | l

```

- **运算符 (operator):**

```

operator → + | - | * | / | % | > | < | = | == | >= | <= | ! | != | && | ||
        → += | -= | ++ | --

```

- **标点符号 (包括括号以及注释):**  $punctuation \rightarrow , | ; | ( | ) | [ | ] | \{ | \} | // | /* | */$

此外, 非终结符集合还包括很多其他内容, 比如我们下面产生式集合里提到的 **Decl** (声明), 若干常量定义 **ConstDefs**, 以及开始符号编译单元 **CompUnit** 等。

### 1.2.2 产生式集合 $\mathcal{P}$

在定义了终结符与非终结符集合后, 接下来需要给出 SysY 语言的产生式规则集合, 以描述 SysY 的各个语言特性。需要注意的是, 在设计产生式集合的过程中, 也产生了许多**中间非终结符**, 将会在接下来的介绍中依次介绍。

结合对 SysY 官方文档中给出的巴克斯范式定义, 我们以一整个编译单元 (即一段完整的 SysY 代码片段) 作为开始符号 **CompUnit**, 对各个中间非终结符, 给出如下的产生式规则定义:

- **编译单元 (CompUnit):** 该非终结符表示一段完整的 SysY 程序代码。

不同于 C++ 语法, SysY 没有在函数外部提供诸如 `#include` 的头文件调用库、`#define` 的宏定义等等指令, 仅提供了基础的声明指令和函数定义指令。

因此, 一段完整的 SysY 代码中, 事实上仅有两个组成类型, 即声明和函数定义 (包含 `main` 函数), 一个编译单元由若干个声明语句和函数定义语句组成, 于是给出其产生式如下:

$$CompUnit \rightarrow ( Decl \mid FuncDef ) \mid CompUnit ( Decl \mid FuncDef )$$

## 声明语句的产生式集合构建

- **声明 (Decl)**: 该非终结符表示一句声明语句。

在 SysY 语言中, 仅提供了由 `const` 修饰符决定的常量声明语句 (`ConstDecl`) 和变量声明语句 (`VarDecl`) 两种声明类型。

因此, 一个声明语句可以由这两种类型组成, 于是给出其产生式如下:

$$Decl \rightarrow ConstDecl \mid VarDecl$$

- **常量声明 (ConstDecl)**: 该非终结符表示一句常量声明语句。

需要注意的是, 不能简单地认为常量声明语句是变量声明语句前加一个 `const` 即可, 这是因为变量的声明可以不赋初值, 需要分开作为两个不同的非终结符来分析判断。

这里, 我们把常量声明分为三个部分, 即修饰符 (`const`)、基本类型符号 (`BType`)、若干个常量定义语句 (`ConstDefs`)。基于此, 我们给出其产生式如下:

$$ConstDecl \rightarrow \text{const } BType \text{ ConstDefs};$$

- **基本符号 (BType)**: 该非终结符表示定义类型的关键字符号。

在 SysY 中, 仅提供了 `int` 和 `float` 的实现, 因此, `BType` 的产生式较为简单, 即如下所示:

$$BType \rightarrow \text{int} \mid \text{float}$$

- **若干常量定义 (ConstDefs)**: 该非终结符表示包含若干个由 ‘;’ 相连的单个常量定义语句 (`ConstDef`) 的语句。

$$ConstDefs \rightarrow ConstDef \mid ConstDefs, ConstDef$$

- **常量定义 (ConstDef)**: 该非终结符表示形如 `A[n]=N+1` 的常量定义语句。

对于一个常量定义语句, 我们对等号两边分别来看: 等号左边是常量标识 (`ConstIdent`), 可能存在数组结构。等号右边是常量初值 (`ConstInitVal`), 可能是运算式, 也可能是数组形式。因此, 给出如下产生式:

$$ConstDef \rightarrow ConstIdent = ConstInitVal$$

- **常量标识 (ConstIdent)**: 该非终结符表示形如 `A[n]` 的常量标识符部分。其中 `[]` 前为先前已给出产生式的标识符 (`Ident`), `[]` 内为常量表达式 (`ConstExp`)。给出如下产生式:

$$ConstIdent \rightarrow \text{Ident} \mid ConstIdent[ConstExp]$$

- **常量初值 (ConstInitVal)**: 该非终结符表示常量的初值部分, 其值可能为常量表达式或数组形式, 其中数组形式比较复杂, 这里我们定义非终结符 `ConstArray`, 其值为若干个逗号连接的表达式或数组, 来辅助我们构造产生式。

需要注意的是, 常量初值的形式需要与先前常量定义相关联, 若不为数组, 则常量初值为一个常量表达式 (`ConstExp`), 若为数组, 则需与数组维度相对应。

给出如下产生式来定义常量初值:

$$ConstInitVal \rightarrow ConstExp \mid ConstArray$$

$$ConstArray \rightarrow ConstInitVal \mid ConstArray, ConstInitVal$$

如上，我们即完成了对常量声明语句规则的的产生式构建。

- **变量声明 (VarDecl)**: 该非终结符表示一条变量声明语句，除去 `const` 修饰词，其产生式形式基本与常量声明语句相同，但其内部的非终结词含义有些许不同，给出产生式如下：

$$VarDecl \rightarrow BType \ ValDefs;$$

- **若干变量定义 (ValDefs)**: 该非终结符表示包含若干个由 ‘;’ 相连的单个变量定义语句 (ValDef) 的语句。

$$ValDefs \rightarrow ValDef \mid ValDefs, ValDef$$

- **变量定义 (ValDef)**: 该非终结符表示形如 `A[n]=N+1` 的常量定义语句。

对于变量定义语句，其形式与常量定义语句类似，但是需要注意的是，变量的定义可以只声明不赋值。因此，给出如下产生式：

$$ValDef \rightarrow ValIdent \mid (ValIdent = InitVal)$$

- **变量标识 (ValIdent)**: 该非终结符表示形如 `A[n]` 的变量标识符部分。不同于常量标识符部分，`[]` 内为可以含变量的表达式 (Exp)。于是给出如下产生式：

$$ValIdent \rightarrow Ident \mid ValIdent[Exp]$$

- **变量初值 (InitVal)**: 该非终结符表示变量的初值部分，同样为可以含变量的表达式 (Exp)

给出如下产生式来定义常量初值：

$$InitVal \rightarrow Exp \mid Array$$

$$Array \rightarrow InitVal \mid Array, InitVal$$

如上，我们又完成了对变量声明语句规则的的产生式构建。至此，声明语句的产生式已经全部构建 (其中 `Exp` 和 `ConstExp` 稍后会在表达式中给出)

## 函数定义语句的产生式集合

- **函数定义 (FuncDef)**: 该非终结符表示一句函数定义语句。

考察 SysY 中的函数定义，其形如 `c++` 中函数定义，由函数类型 (FuncType)、函数名称 (Ident)、参数表 (FuncParams) 和语句块 (Block) 组成。

于是给出定义产生式如下：

$$FuncDef \rightarrow FuncType \ Ident \ (FuncParams) \ Block$$

- **函数返回类型 (FuncType)**: 该非终结符表示函数返回类型。

函数返回类型分为 `int`, `float`, `void` 三种，需要注意的是，当返回类型为 `int`、`float` 时，函数内部的所有分支都应该有一个包含 `Exp` 的 `return` 语句；而 `void` 类型可以无 `return` 或者不带返回的 `return` 语句。

给出产生式如下：

$$FuncType \rightarrow void \mid int \mid float$$



- **函数参数表 (FuncParams)**: 该非终结符表示多个函数参数。

给出产生式如下:

$$FuncParams \rightarrow FuncParam \mid FuncParams, FuncParam$$

- **函数参数 (FuncParam)**: 该非终结符表示一个函数参数。

函数参数的定义类似于声明语句中的写法, 于是给出产生式如下:

$$FuncParams \rightarrow Btype \textbf{Ident} \mid Btype \textbf{Ident} [ \ ] \mid Btype \textbf{Ident} [ Exp ]$$

- **语句块 (Block)**: 该非终结符表示一段语句集合, 包含若干语句。

给出产生式如下:

$$Block \rightarrow BlockItem \mid Block, BlockItem$$

- **语句块 (Block)**: 该非终结符表示一个语句。

在 SysY 中, 一个语句可能有多重类型, 包含声明语句 (先前已定义)、赋值语句、if 语句、while 语句、return 语句、其他语句, 以及 continue、break 等关键字, 于是给出产生式如下:

$$BlockItem \rightarrow Decl \mid Assign \mid IfStmt \mid WhileStmt \mid ReturnStmt \mid ; \mid EXP; \mid Block \mid \textbf{break}; \mid \textbf{continue};$$

- **赋值语句 (Assign)**: 该非终结符表示一个赋值语句。

对于一个赋值语句, 需要对左值表达式 (LVal) 进行定义, 右式为一个运算式即可。于是给出产生式如下:

$$Assign \rightarrow LVal = Exp;$$

- **左值 (LVal)**: 该非终结符表示赋值语句的左值。

左值可以是单一的标识符和具体的数组元素, 于是给出产生式如下:

$$LVal \rightarrow \textbf{Ident} \mid LVal [ Exp ]$$

- **if 语句 (IfStmt)**: 该非终结符表示一个 if 语句。

SysY 中 if 语句的定义比较简单, 但是需要引入条件表达式 (Cond) 来进行定义, 还需要注意 else 语句, 给出产生式如下:

$$IfStmt \rightarrow \textbf{if} (Cond) stmt \mid \textbf{if} (Cond) stmt \textbf{else} stmt$$

- **While 语句 (WhileStmt)**: 该非终结符表示一个 While 语句。

SysY 中 while 语句的定义也比较简单, 类似于 if 语句, 给出产生式如下:

$$WhileStmt \rightarrow \textbf{while} (Cond) stmt$$

- **Return 语句 (ReturnStmt)**: 该非终结符表示一个 return 语句。

SysY 中 return 语句的定义同样比较简单, 但需要注意之前提到的可能存在没有返回值的现象, 给出产生式如下:

$$ReturnStmt \rightarrow \textbf{return}; \mid \textbf{return} (Cond) stmt;$$

## 表达式的产生式集合

对表达式的定义需要考虑之前给出的终结符集合中对应的各个运算的优先级。

在 SysY 中, 运算表达式存在加减法、乘除法和一元运算 (基本表达式、函数调用、单目运算) 三种, 优先级由低到高; 条件表达式存在逻辑或、逻辑与、逻辑相等、关系运算, 优先级由低到高。需要注意的是, 常量的运算表达式与变量相同, 但其中使用的 `ident` 标识符需要为常量。

接下来, 我们按照以上优先级规则, 对两种运算分别给出其产生式定义

- **运算表达式 (EXP)**: 该非终结符表示一个运算表达式。

根据我们先前的习题与学习, 可以认识到对于运算表达式, 需要由底向上根据优先级来构建产生式, 即按照加减法 (`AddExp`)、乘除法 (`MulExp`) 和一元运算 (`UnaryExp`), 其中一元表达式包含基本表达式 (`PrimaryExp`)、函数调用、单目运算式 (`UnaryOP`), 于是可以给出产生式集合如下:

$$EXP \rightarrow AddExp$$

$$AddExp \rightarrow MulExp \mid AddExp + MulExp \mid AddExp - MulExp$$

$$MulExp \rightarrow UnaryExp \mid MulExp * UnaryExp \mid MulExp / UnaryExp \mid MulExp \% UnaryExp$$

$$UnaryExp \rightarrow PrimaryExp \mid UnaryOp \space UnaryExp \mid \mathbf{Ident} \space () \mid \mathbf{Ident} \space (FuncParams)$$

$$PrimaryExp \rightarrow (Exp) \mid LVal \mid Number$$

$$UnaryOp \rightarrow + \mid - \mid !$$

$$Number \rightarrow \mathbf{IntConst} \mid \mathbf{floatConst}$$

- **条件表达式 (Cond)**: 该非终结符表示一个条件表达式。

同样的, 对于条件表达式, 也需要由底向上根据优先级来构建产生式, 即按照逻辑或 (`LORExp`)、逻辑与 (`LAndExp`)、逻辑相等 (`EqExp`)、关系运算 (`RelExp`)、加减法 (`AndExp`), 于是可以给出产生式集合如下:

$$Cond \rightarrow LORExp \quad LORExp \rightarrow LAndExp \mid LORExp \parallel LAndExp \quad LAndExp \rightarrow EqExp \mid LAndExp \&\&$$

$$EqExp \quad EqExp \rightarrow RelExp \mid EqExp == RelExp \mid EqExp != RelExp \quad RelExp \rightarrow AddExp \mid RelExp >$$

$$AddExp \mid RelExp < AddExp \mid RelExp >= AddExp \mid RelExp <= AddExp$$

### 1.2.3 开始符号 *S*

该文法的开始符号是编译单元: ***CompUnit***。

至此, 我们设计了 SysY 全部语言特性的产生式集合, 规定了相应的开始符号、终结符集合和非终结符, 完成了 SysY 语言子集的上下文无关文法设计。

## 2 ARM 汇编编程

在本小节, 我们将按照我们定义的编译器支持的 SysY 语言特性进行分类, 编写对应的 **armv7-a** 汇编代码, 在这之前, 有必要学习一下 **armv7** 汇编用法 [2]:

### 2.1 armv7-a 汇编学习

#### 一、寄存器组

我们主要使用通用寄存器, 在 **armv7** 架构中, 共有 16 个通用寄存器:

- r0 ~ r3: 用来传递函数参数、暂存数据
- r4 ~ r11: 用来保存被调函数的局部变量、暂存数据
- r12: 记录函数调用过程中上一次 sp 指针的值
- r13(sp): 函数堆栈寄存器
- r14(lr): 记录函数返回地址
- r15(pc): 程序计数器

## 二、文件组成

一个汇编文件可以分为若干段，常使用的段如下：

- .text: 代码段，其下面的内容都是代码
- .rodata: 只读数据段，这部分存放常量，如 const 修饰的变量和常量字符串等
- .data: 包含程序的全局变量和局部变量
- .bss: 存放一些未初始化的全局变量和局部变量，它们的初值是 0

除此之外，还有一个特定的伪指令 **.section**，它的作用是指定将后续汇编代码放置到哪个段中，使用此伪指令提供了更高的自由，也让代码更加清晰。

## 三、输入输出流

本次实验中，我们使用 SysY 官方提供的运行时库 libsysy.a 和 sylib.h 文件来进行输入输出流的实现，这些库提供了一系列 I/O 函数、计时函数等用于在 SysY 程序中表达输入/输出、计时等功能需求。

需要注意的是，在 SysY 语法中，不包括 #include 的预处理调用库指令，因此，在 SysY 源程序中不会出现对 sylib.h 的文件包含，而是由 SysY 编译器来分析和直接处理 SysY 程序中对这些函数的调用。

本次实验中，我们使用到的 SysY 运行时库函数主要包含：

- `getint()`: 输入一个整数，返回对应的整数值
- `getfloat()`: 输入一个浮点数，返回对应的浮点数值。
- `getarray(int[])`: 输入一串整数，第 1 个整数代表后续要输入的整数个数，该个数通过返回值返回；后续的整数通过传入的数组参数返回。
- `getfarray(float[])`: 输入一个整数后跟若干个浮点数，第 1 个整数代表后续要输入的浮点数个数，该个数通过返回值返回；后续的浮点数通过传入的数组参数返回
- `putint(int)`: 输出一个整数的值。
- `putfloat(float)`: 输出一个浮点数的值。
- `putarray(int,int[])`: 第 1 个参数表示要输出的整数个数 (假设为 N)，后面应该跟上要输出的 N 个整数的数组。putarray 在输出时会在整数之间安插空格。

- `putarray(int, float[])`: 第 1 个参数表示要输出的浮点数个数 (假设为  $N$ )，后面应该跟上要输出的  $N$  个整数的数组。`putarray` 在输出时会在浮点数之间安插空格。

此外，由于链接运行时库进行编译会额外输出运行时长 (在 `sylib.c` 中实现)，且部分输出函数没有换行，为了方便测试的直观性，我们对 `sylib.c` 进行了相应的修改，使其输出格式便于检验，已经上传到我们实验的[github 仓库](#)中。

## 2.2 赋值与算术运算

对于我们要实现的 SysY 编译器，其要支持整形和浮点型常量与变量，因此，在本小节中，我们将以如下的 C 程序为例，使用这些特性，并编写对应的汇编代码：

---

```

1  const int A = 1; //声明整形常量，赋了初值，位于.data 段
2  const float B = 2.2; //声明浮点型常量，赋了初值，位于.data 段
3  int e; //声明整形变量，默认赋值为 0，位于.bss 段
4  float f; //声明浮点常量，默认赋值为 0.0，位于.bss 段
5  int main()
6  {
7      int a; //声明整形变量，没有赋初值，位于.data 段
8      int b = 1; //声明整形变量，赋了初值，位于.data 段
9      float c; //声明浮点型变量，没有赋初值，位于.data 段
10     float d = 1.1; //声明浮点型变量，赋了初值，位于.data 段
11
12     a = b + A; //整形加法指令 add
13     b = A / b; //整形除法，无指令，自己实现
14
15     f = B + d; //浮点加法指令 vadd
16     f = A - B; //整形和浮点进行运算，先转换，再浮点运算
17     return 0;
18 }
```

---

首先是很多整形、浮点型常量与变量的声明与赋值语句，根据上一小节的学习，我们知道这些变量应该位于 `.bss` 段和 `.section` 段，而常量应该位于只读数据段，对应的汇编代码如下：

---

```

1  @ 选择架构
2  .arch armv7-a
3  .section .rodata
4      A: .word 1
5      B: .float 2.2
6
7  .section .bss
8      e: .word
9      f: .float
```

---

```

10
11 .section .data
12     a: .word
13     b: .word 1
14     c: .float
15     d: .float 1.1

```

然后实现整形之间的运算,和其他汇编语言一样,我们需要通过操作寄存器来进行运算。在 ARMv7 汇编语言中, `ldr` 和 `str` 是用于加载和存储数据的指令:

- `ldr`: 用于将数据从内存加载到寄存器中,常见的使用方式为:

```

1  ldr r0, =A  @ 将符号 A 的地址赋值给 r0 寄存器
2  ldr r0, [r0] @ [] 可理解为指针,将 r0 地址处的值,即 A 的值赋值给 r0

```

- `str`: 将寄存器中的数据存储到内存中,格式和 `ldr` 类似

在 ARMv7 中,加法、减法、乘法的指令格式完全一样,为 `inst rd, rs, rt`,作用是将 `rs`, `rt` 寄存器中的值取出,然后执行指令 `inst`,并将结果存到 `rd` 之中。在这里我们以加法指令 `add` 为例,实现上述代码中的 `a = b + A`;

```

1  @ 从内存中获取数据
2  ldr r1, =b
3  ldr r2, =A
4  ldr r3, =a
5  ldr r1, [r1]
6  ldr r2, [r2]
7  @ 赋值运算
8  add r4, r1, r2
9  @ 将数据存储回内存
10 str r4, [r3]

```

而对于除法运算和取模运算,在我们所选的 ARMv7 架构中,并没有直接支持的指令,我们需要手动实现一下。实现思路就是进入循环,然后检查被除数是否小于除数,如果是,跳出循环;否则被除数减去除数,然后商 +1,最后将结果存到两个寄存器中,商即为整形除法运算的结果,而余数则为整形取模运算的结果。

```

1  @ 所选 armv7 架构没有除法指令,手动实现:余数存储在 r1 中,商存储在 r0 中
2  mov r0, #0
3  division_loop:
4      @ 检查被除数是否小于除数,如果是,跳出循环
5      cmp r1, r2
6      blt division_done

```

```

7      @ 被除数减去除数，商加 1
8      sub r1, r1, r2
9      add r0, r0, #1
10     b division_loop
11 division_done:
12     @ 将返回值设置为 0
13     mov r0, #0
14     @ 退出程序并发起系统调用
15     mov r7, #1
16     swi 0

```

接下来实现浮点的运算指令。为了支持浮点运算，我们需要指定浮点处理单元 (FPU)，在 ARM 体系结构中，有多种不同的 FPU，例如 NEON 和 VFP，它们有着不同的指令集和性能特性，在本小节中我们以 neon 为例编写浮点运算汇编代码：

- NEON 寄存器：NEON 中有 32 个 64 位寄存器，以 d 开头，可存储双精度浮点数
- NEON 指令：NEON 指令用于执行浮点数向量操作，这些指令可以同时多个浮点数值执行相同的操作，从而提高并行性和性能，常见的运算指令有 vadd, vsub, vmul 等，用法和 ARMv7 相应运算指令相同
- NEON 加载与存储：对应于 ldr 和 str，NEON 加载和存储指令为 vldr 和 vstr

了解这些知识后，着手实现代码中的  $f = B + d$ ；这句单精度浮点加法运算代码：

```

1  @ 先将 f, B, d 的地址(整形)，加载到通用寄存器中
2  ldr r1, =f
3  ldr r2, =B
4  ldr r3, =d
5  @ 然后将这些地址对应的浮点值，加载到 D 寄存器中
6  vldr d1, [r1]
7  vldr d2, [r2]
8  vldr d3, [r3]
9  @ 单精度浮点数运算，32 位
10 vadd.f32 d1, d2, d3

```

最后，还有整形与浮点型进行运算的这类语句，我们需要把整形转换为浮点型，然后使用浮点型运算指令进行运算，在 ARMv7 中，整形转换为浮点型可使用 vcvrt.f32.s32(u32)，将一个四字节有(无)符号整形变量转换为单精度浮点型变量：

```

1  ldr r4, =A
2  @ 将整形变量转换为 NEON 浮点型变量
3  vcvrt.f32.u32 d5, d4
4  vsub.f32 d1, d5, d1

```

现在，我们已经完成了如上 C 代码对应的 ARM 汇编代码编写，接下来使用交叉工具链进行编译，需要注意的是，由于我们使用了 NEON 这种 FPU，需要加入参数，完整的编译命令为：

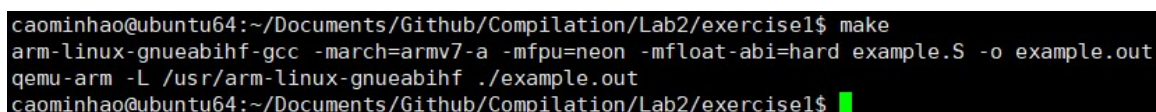
---

```
1 $ arm-linux-gnueabi-gcc -march=armv7-a -mfp=neon -mfloat-abi=hard example.S -o example.out
2 $ qemu-arm -L /usr/arm-linux-gnueabi ./.example.out
```

---

其中，`-march` 参数表示指令集架构，`-mfp` 参数表示选择的浮点计算单元，`-mfloat-abi` 参数表示参数浮点运算的 ABI，在这里指定为 `hard`，即交给硬件进行运算而非软件进行模拟。

将编译命令写入 Makefile 中，进行编译，结果如下图所示：



```
caominhao@ubuntu64:~/Documents/Github/Compilation/Lab2/exercise1$ make
arm-linux-gnueabi-gcc -march=armv7-a -mfp=neon -mfloat-abi=hard example.S -o example.out
qemu-arm -L /usr/arm-linux-gnueabi ./.example.out
caominhao@ubuntu64:~/Documents/Github/Compilation/Lab2/exercise1$
```

图 2.1: ARM 汇编：赋值与算术运算编译成功通过

本小节中的完整代码，和对应的 Makefile，请参考[example.S & Makefile](#)

## 2.3 数组声明与运算

数组是一段连续的内存空间，在 ARMv7 中，可通过数据类型后跟若干值的方式开辟一个数组，如 `word 1,2,3,4` 声明一个四元素整形数组，`.float` 声明一个单精度浮点数组等。这些声明将在 `.data` 段中分配内存以存储数组元素，并将元素初始化为指定的值。

而对于数组的访问，同其他汇编语言一样，可以使用基址寄存器加上偏移量的方式进行访问。在本小节，我们将以如下 C 代码为例，翻译为对应的 ARM 汇编程序，熟悉数组的使用方式。

---

```
1 int arr1[5]; //声明整形数组
2 int main()
3 {
4     float arr2[5] = {1.1,2.2,3.3,4.4,5.5}; //声明浮点数组
5     // 整形与浮点型运算
6     arr2[0] = arr1[1] + arr2[2];
7     arr2[1] = arr1[2] - arr2[3];
8     arr2[2] = arr1[3] * arr2[4];
9     return 0;
10 }
```

---

首先，在 `.data` 段中声明数组，由于 `arr1` 未赋初值，但给出了数组长度，因此只需要留出空间即可，`arr2` 为浮点数组，采取 `.float` 后跟各元素值的方式创建数组。

---

```
1 .section .data
2     arr1: .space 20 @ 预留空间给 arr1, 共 5 个 int, 一共 20 字节
3     arr2: .float 1.1, 2.2, 3.3, 4.4, 5.5 @ 浮点数数组
```

---

接着，我们以 `arr2[0] = arr1[1] + arr2[2]` 这条语句为例，我们需要获取两个数组的首地址，并加上偏移量（都是四字节乘以索引号），然后使用浮点运算指令进行运算。

```

1  @ 设置 r0 为 &arr2[2] 的地址
2  ldr r0, =arr2
3  @ 加载 &arr2[2] 的地址到 r0
4  ldr r0, [r0, #8]
5
6  @ 设置 r1 为 &arr1[1] 的地址
7  ldr r1, =arr1
8  @ 加上 4 以获得 &arr1[1] 的地址
9  add r1, r1, #4
10
11 @ 加载 arr1[1] 的值到 r2, arr2[2] 的值到 r3
12 ldr r2, [r1]
13 ldr r3, [r0]
14
15 @ 使用 NEON 单精度浮点加法计算 arr2[0] = arr1[1] + arr2[2];
16 vadd.f32 s0, s2, s3
17 @ 使用 NEON 存储指令存回
18 vstr s0, [r0]

```

在 Makefile 中编写新的 target，进行编译，结果如下图所示：

```

caominhao@ubuntu64:~/Documents/Github/Compilation/Lab2/exercisel$ make array
arm-linux-gnueabi-gcc -march=armv7-a -mfpu=neon -mfloat-abi=hard example.S -o example2.out
qemu-arm -L /usr/arm-linux-gnueabihf ./example2.out
caominhao@ubuntu64:~/Documents/Github/Compilation/Lab2/exercisel$ ls
example2.out example2.S example.out example.S lib Makefile
caominhao@ubuntu64:~/Documents/Github/Compilation/Lab2/exercisel$ make clean
rm -fr *.out

```

图 2.2: ARM 汇编：数组运算编译成功通过

本小节所编写的完整汇编代码，请参考 [example2.S](#)。

## 2.4 函数定义及其调用

在 ARMv7 汇编中，函数定义通常包括以下步骤：

- 声明函数入口点：使用 `.global` 指令声明函数的入口点，以便其他部分的代码可以调用它
- 保存寄存器状态：如果函数使用了某些寄存器（如 `lr` 或其他调用者保存的寄存器），则需要在函数内部保存和恢复这些寄存器的状态，以确保函数执行后不会破坏调用者的寄存器值
- 编写函数体，执行函数操作
- 函数返回：在函数完成时，通常使用 `bx lr` 指令返回到调用者

而对于函数的调用，则通常包括以下步骤：



- 传递参数：通过寄存器传递，在 2.1 节我们知道， $r0 \sim r3$  是用来传递函数参数，暂存数据的
- 调用函数：使用 bl（分支链接）指令调用函数。bl 指令将返回地址保存到 lr 寄存器，并跳转到函数入口点
- 执行函数
- 函数返回：函数执行完毕后，使用 bx lr 指令返回到调用者，同时返回值通常存储在 r0 寄存器中

了解这些知识后，我们实现一下如下的 C 程序对应的汇编代码，实现两数相加并返回，此例子涵盖了上面提到的所有过程。

---

```

1  int add(int a,int b) {
2      return a+b;
3  }
4  int main() {
5      int a = 1 , b = 2;
6      int c = add(a,b);
7      return 0;
8  }
```

---

首先实现函数 (即函数定义部分)，详细汇编代码如下：

---

```

1  .global add
2  add:
3      push {lr}    @ 保存调用者保存的寄存器
4      @ 加载参数 a 和 b 到 r0 和 r1
5      ldr r0, [sp, #4]
6      ldr r1, [sp, #8]
7      @ 函数体：执行 a + b 操作
8      add r2, r0, r1
9      @ 将结果存储在 r0 中，作为返回值
10     mov r0, r2
11     @ 恢复调用者保存的寄存器并返回
12     pop {lr}
13     bx lr
```

---

然后在 main 函数中进行函数调用，保存 lr 寄存器，返回值在 r0 寄存器：

---

```

1  @ 保存调用者保存的寄存器
2  push {lr}
3  @ 调用 add 函数，结果存储在 c 中，注意不要使用 r2 和 r3 寄存器，否则会产生内存错误
4  ldr r0, =a
5  ldr r1, =b
```

---

```

6  ldr r4, =c
7  bl add
8  str r0, [r4]
9  @ 将函数返回值设置为 0
10 mov r0, #0
11 @ 恢复调用者保存的寄存器并返回
12 pop {lr}
13 bx lr

```

---

依然可以通过编译，函数部分完整汇编代码请参考 [function.S](#)

## 2.5 分支语句与关系运算

我们要实现的 SysY 编译器中，还需要实现条件分支语句和关系运算（即条件判断）。本小节中，我们将以如下的 C 语言程序为例，对 if、while 语句和逻辑运算的语言特性进行考察，并编写对应的 arm 汇编代码：

---

```

1  int ifElseIf(){
2      int a;
3      a = getint();
4      int b;
5      b = getint();
6      // 测试 if else 语句
7      if(a <= 4) {          // 测试小于等于分支
8          if(a+10>=12){      // 测试 if 语句和大于等于语句
9              return 1;      // 测试不同位置的 return 指令
10         }
11         return 2;
12     }
13     else {
14         if(10 < 0xb) {      // 测试立即数的大于语句和十六进制表示比较
15             // 测试 if-elif-else 语句
16             if (b == 10)    // 测试等于语句
17                 a = 25;
18             else if (!(b != 5)) // 测试不等于语句和非语句
19                 a = 15;
20             else
21                 a = -10;
22         }
23         return a;
24     }
25 }
26 int main(){

```

```

27     int c = ifElseIf();
28     putint(c);
29 }

```

关于条件分支运算，ARMv7 要比 x86 简单许多。与 mips 指令集类似，ARMv7 中条件分支的实现离不开标签、cmp 指令和 beq、bne、ble、bge、bgt、blt、b 等控制流指令的有机结合。

例如如下语句，即实现了一段简单的分支跳转指令：

```

1  cmp r0, #4          @ 比较 r0 和 4
2  ble ifElseIf_LessOrEqual @ 如果 r0 <= 4, 则跳转到 ifElseIf_LessOrEqual 标签

```

其中，尽管分支语句的语法较为简单，但由于分支语句会造成函数有多个出口，且需要进行标签间的来回跳转，在生成或设计汇编代码时，需要综合考量分支间的逻辑关系，才能设计出一个好的 ARMv7 汇编代码，简化不必要的指令数。

如本次尝试翻译为 ARMv7 汇编程序的 C 语言程序中，存在大量的分支判断，且由于是函数，分支语句造成了较多的函数出口，需要我们设计一个好的转换逻辑，从而简化代码逻辑。

本次实验中，我们采用自上至下的层次遍历方式进行分析设计，这种方式能够保障分支跳转的准确性，但可能存在代码冗余，需要后续的优化。于是可以给出如下的分支判断代码：

```

1  If:
2      ....
3      cmp r4, #4          @ 比较 a 和 4
4      ble If_If @ 如果 r0 <= 4, 则跳转到 If_If 标签
5      mov r4, #10
6      cmp r4, #0xb        @ 比较 10 和 11 (0xb 的十进制表示)
7      blt Else_If_If @ 如果 10 < 11, 则跳转到 Else_If_If 标签
8  If_If:
9      add r4, r4, #10 @ a += 10
10     cmp r4, #12        @ 比较 a 和 12
11     bge if_if_return @ 如果 a >= 12, 则跳转到 if_if_return 标签
12     mov r0, #2         @ 返回 2
13     b End @ 跳转到 End 标签
14  if_if_return:
15     mov r0, #1         @ 返回 1
16     b End @ 跳转到 End 标签
17  Else_If_If:
18     cmp r0, #10        @ 比 b 和 10
19     beq Else_If_If_return @ 如果 b 等于 10, 则跳转到 Else_If_If_return 标签
20     cmp r0, #5         @ 比较 b 和 5
21     bne Else_If_Else_return @ 如果 r0 不等于 5, 则跳转到 Else_If_Else_return 标签
22     mov r0, #15        @ 如果 r0 等于 5, 则设置 r0 为 15
23     b End @ 跳转到 End 标签

```

```

24 Else_If_Else_return:
25     mov r0, #-10      @ 设置 r0 为 -10
26     b End            @ 跳转到 End 标签
27 Else_If_If_return:
28     mov r0, #25       @ 设置 r0 为 25
29     b End            @ 跳转到 End 标签
30 End:
31     pop {r4, pc}      @ 恢复 r4 并返回

```

编译该 ARMv7 程序并执行，发现程序能够正确运行。尝试给出不同的输入，可以看到程序给出了相应的正确输出，表明该分支跳转程序编写正确，实现了我们所需的功能。

分支部分完整汇编代码请参考 [If\\_Branch.S](#)。

```

lwy@ubuntu: ~/byyl/lab2
lwy@ubuntu:~/byyl/lab2$ arm-linux-gnueabi-gcc -mcpu=neon test.S sylib.c -o test -static
lwy@ubuntu:~/byyl/lab2$ qemu-arm -L /usr/arm-linux-gnueabi-hf ./test
3 2
1
lwy@ubuntu:~/byyl/lab2$ qemu-arm -L /usr/arm-linux-gnueabi-hf ./test
1 2
2
lwy@ubuntu:~/byyl/lab2$ qemu-arm -L /usr/arm-linux-gnueabi-hf ./test
5 10
25
lwy@ubuntu:~/byyl/lab2$ qemu-arm -L /usr/arm-linux-gnueabi-hf ./test
5 5
15
lwy@ubuntu:~/byyl/lab2$ qemu-arm -L /usr/arm-linux-gnueabi-hf ./test
5 3
-10
lwy@ubuntu:~/byyl/lab2$

```

图 2.3: ARM 汇编：分支跳转编译成功通过、且分支输出正确

## 2.6 循环语句与逻辑运算

类似的，while 语句的实现逻辑与 if 类似，这里不再赘述，同样给出完整汇编代码 [While\\_Branch.S](#)。其中，continue 和 break 关键字可以视作另一类标签的实现，同样也需要设计相应的分支语句，我们将在之后的逻辑运算语句中给出例子。

逻辑条件判断可以通过多个关系判断复合使用，也可以通过值的计算来达成。其中，前者可以节省指令条数，起到简化程序的作用。需要注意的是，ARMv7 中对逻辑运算（与或运算等）的分支跳转，对于一个逻辑或跳转，其采用多个关系判断组合的 ARMv7 汇编是简单的，如当 r1 大于 7 或小于 10 时继续执行 while(continue)，可以写为：

```

1  cmp r1, #10      @ 比较 j 和 10
2  blo continue    @ 如果 j 小于 10，跳到 continue

```

```

3  cmp r1, #7      @ 比较 j 和 7
4  bhi continue @ 如果 j 大于 7, 跳到 continue

```

但是，对与逻辑与跳转的实现，如当 `r1` 大于 7 且小于 10 时中断执行 `while(break)`，就需要思考一下。其实，逻辑与可以视作非逻辑或，我们在实现非逻辑时，可以通过跳转至不同的标签分支来实现，因此，可以给出如下的代码来实现逻辑与：

```

1  cmp r1, #7      @ 比较 j 和 7
2  blo run        @ 如果 j 小于 7, 跳到正常执行
3  cmp r1, #10     @ 比较 j 和 10
4  bhi run        @ 如果 j 大于 10, 跳到正常执行
5  b break        @ 如果条件都不满足, 中断

```

### 3 思考题

若要设计一个编译器，将 SysY 语言转换为汇编语言，我们仍可以将这个工作划分到编译器的各个阶段上去：

- **词法分析：**将 SysY 代码字符串分割成一系列的 token，例如标识符、关键字、运算符、常量等
  - 数据结构：词法分析器需要一种数据结构来表示程序中的各种符号，如变量名、SysY 的关键字、操作符等，可以设计一种二元组，分别表示 id 的类型和值
  - 算法：可使用有限自动机 (NFA,DFA) 和正则表达式来识别和生成 tokens
- **语法分析：**根据一定的文法规则，将词法单元组合成一棵抽象语法树，表示源代码的结构和语义
  - 数据结构：使用抽象语法树 AST
  - 算法：可以使用自顶向下或自底向上的语法分析器，如 LL(k) 或 LR(k) 分析器
- **语义分析：**对 AST 进行类型检查、作用域分析等操作，确保源代码符合语言的语义规范
  - 数据结构：需要一个符号表，用于跟踪变量、函数、类型等信息
  - 算法：符号表中需要一种高效的搜索算法，如快速排序，堆排序等
- **中间代码生成：**将 AST 转换为一种中间表示形式 (IR)，如三地址代码
  - 数据结构：中间表示形式 IR，如 LLVM，或者最简单的三地址代码
  - 算法：可以在 yacc 的产生式规则定义部分中设计一种高效的转换策略
- **代码优化：**比如消除死代码，递归优化等，提高目标代码的执行效率
  - 数据结构：各种数据流分析和控制流图等
  - 算法：各种优化技术，如常量折叠、死代码消除、循环展开等
- **目标代码生成：**将中间代码转换为目标平台的汇编代码
  - 数据结构：目标机器的汇编语言表示
  - 算法：将中间代码转换为目标机器代码，使用寄存器分配、指令选择等技术

## 4 分工情况

- **曹珉浩：**CFG 设计中负责描述终结符、非终结符集合，并给出底层非终结符的描述。ARM 汇编中负责赋值与算术运算、数组声明与运算、函数定义及其调用等方面语言特性的编程研究，此外还设计了编译的 makefile 文件。
- **李威远：**CFG 设计中负责描述产生式集合，并给出中间非终结符的描述。ARM 汇编中负责条件分支语句与关系运算、循环语句与逻辑运算等方面语言特性的编程研究。

此外，SysY 语言特性选择、输入输出流实现等其他方面，为两人共同讨论完成。

## 参考文献

- [1] Monica S.Lam Alfred V.Aho. 编译原理. 机械工业出版社, 2009.
- [2] Couvrir. Linux 进阶-arm\_v7 架构和 arm 常用汇编指令. [https://blog.csdn.net/weixin\\_47077788/article/details/128211454?](https://blog.csdn.net/weixin_47077788/article/details/128211454?), 2023.
- [3] SysY. Sysy 官方文档阅读. <https://gitlab.eduxiji.net/nscscs/compiler2023/-/blob/master/SysY2022%E8%AF%AD%E8%A8%80%E5%AE%9A%E4%B9%89-V1.pdf>, 2023.