



南開大學
Nankai University

计算机学院

编译原理报告

定义你的编译器 & 汇编编程

朱璟钰 孟笑朵

年级：2020 级

专业：计算机科学与技术

指导教师：王刚

2022 年 10 月 15 日

摘要

本次实验确定了要实现的编译器支持的 SysY 语言特性, 给出了用 CFG 描述的 SysY 语言特性形式化定义。包括整型、浮点型、字符串型的常/变量声明/初始化语句、Switch()case 等选择语句、for() 等迭代语句、跳转语句及表达式 (含算术运算、逻辑运算、关系运算、三目表达式、赋值表达式等), 并支持多维数组及函数、输入输出特性。按照 SysY 语言特性, 合作实现了支持不同特性的 ARM 汇编程序, 并调试执行检验结果正确。在结尾部分, 给出了我们对于 SysY 语法制导翻译生成对应汇编代码的思考。

关键字: SysY 语言; 上下文无关文法; ARM 汇编

目录

| | |
|-------------------------------------|-----------|
| 一、 概述 | 1 |
| (一) 项目地址 | 1 |
| (二) 实验简述与分工介绍 | 1 |
| 二、 定义你的编译器 | 1 |
| (一) SysY 语言的终结符 V_T 定义 | 1 |
| 1. 标识符 | 1 |
| 2. 注释 | 2 |
| 3. 整型 & 浮点型 & 字符串常量 | 2 |
| 4. 字符串常量 | 3 |
| (二) SysY 语言中非终结符 V_N 定义 | 4 |
| (三) SysY 语言中开始符号 S 定义 | 4 |
| (四) SysY 语言中产生式集合 P 定义 | 4 |
| 1. 语句块 & 语句 | 5 |
| 2. 函数 | 6 |
| 3. 输入输出 | 6 |
| 4. 变量 & 常量声明与初始化 | 6 |
| 5. 表达式 | 7 |
| 三、 ARM 汇编程序 | 9 |
| (一) 程序一 | 9 |
| (二) 程序二 | 13 |
| 四、 语法制导翻译实现从 SysY 到汇编 | 18 |
| 五、 总结 | 21 |

一、 概述

(一) 项目地址

可在 gitlab 仓库中找到我们项目的代码及相关文档: [项目链接](#)

(二) 实验简述与分工介绍

本次实验由朱璟钰与孟笑朵合作完成。我们设计的编译器计划支持的功能包括: **整型、浮点型、字符串型常量**、变量声明与赋值语句, 迭代语句 (`while()` 语句、`do...while()` 语句、`for()`), 跳转语句 (包括 `break`、`continue`、`return` 等), 选择语句 (`if()` 语句、`if()else` 语句、`switch()case` 语句)、包括算术运算表达式 (加减乘除、按位与或非, 三目表达式等), 逻辑运算表达式, 关系运算表达式 (等于、不等于、大于、小于、大于等于、小于等于), 赋值表达式等**表达式语句**, 并支持**多维数组**, **函数**, **输入输出**等 SysY 语言特性。

朱璟钰同学主要负责整型/浮点型/字符串型的常量定义, 语句块与控制语句 (迭代/跳转/选择), 注释、多维数组及函数定义部分的设计, 并负责汇编阶段的程序一。孟笑朵同学主要负责标识符的定义, 常量/变量的声明与初始化, 表达式语句 (算术/逻辑/关系/赋值), 以及输入输出部分的设计, 并负责汇编阶段的程序二。两人共同完成最后语法制导翻译生成汇编代码的思考。

二、 定义你的编译器

在这一部分中, 本文实现了 SysY 语言的上下文无关文法的形式化描述, 关于上下文无关文法的具体定义我们不过多赘述, 在下文进行上下文无关文法的描述时, 遵循的规则是: 用加粗黑体来标记数位, 符号和终结符号, 用斜体字符串来标记非终结符号, 以同一个非终结符号为头部的多个产生式之间用符号 | 分隔。

由于 SysY 文法定义时内容繁杂, 为了相对条理地梳理这部分内容, 本文先定义了 SysY 语言的终结符, 而后描述了非终结符以及部分 SysY 语言特性对应的产生式¹。

(一) SysY 语言的终结符 V_T 定义

终结符号是语言中用到的基本元素, 一般不可被再次分解。在 SysY 语言中, 终结符包含关键字 (Keyword), 标识符 (Identifier), 常量 (Constant), 注释 (Comment), 符号 (Punctuator), 其中关键字和符号我们直接在产生式中表达, 在这里就不做说明, 下面对其余终结符进行逐一定义。

1. 标识符

SysY 语言中标识符是以字母或下划线开头的, 后接字母下划线或数字的字符串, 根据标识符的特征可以进行如下定义, 其中 *identifier-nondigit* 表示的是字母或下划线, *digit* 表示的是数字 0-9:

CFG 1. identifier

¹在 CFG 的设计过程中我们参考了《SysY 语言定义 (2022 版)》以及官方文档 [ISO/IEC 9899](#) 的定义

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

对于标识符的作用域，局部变量可覆盖全局变量的作用域，且变量名与函数名可重复，且不产生歧义（原因在于函数定义/调用时的上下文无关无法均可与变量相区分）。

2. 注释

实际上注释将在预处理阶段进行处理，故此处给出预处理时的注释处理文法，SysY 语言中注释规范如下：

1. 单行注释：以 ‘//’ 开始，直到换行符结束，不含换行符。
2. 多行注释：以序列 ‘/*’ 开始，直到第一次出现 ‘*/’ 时结束，包含结束处 ‘*/’。

CFG 2. cmtStmt

$$cmtStmt \rightarrow /* comment * / \mid // Line$$

3. 整型 & 浮点型 & 字符串常量

目前，主要支持 int、float、string 三种类型。

CFG 3. BType

$$\begin{aligned}
 BType &\rightarrow 'int' \mid 'float' \mid 'string' \\
 constant\text{-}expression &\rightarrow integer - const \mid float - const \mid string - const
 \end{aligned}$$

支持八进制、十进制、十六进制的整型常量，其中八进制常数以 0 开头，并由数字 0 8 的串组成的；十进制由非 0 数字开头，并由数字 0 9 的串组成；十六进制以 0x 或 0X 开头，并由 0 F 的串组成。并支持有符号及无符号整型。

CFG 4. integer-const

$$\begin{aligned}
\text{sign} &\rightarrow '+' \mid '-' \mid \epsilon \\
\text{signed-integer-const} &\rightarrow \text{sign unsigned-integer-const} \\
\text{unsigned-integer-const} &\rightarrow \text{decimal-const} \mid \text{octal-const} \\
&\mid \text{hexadecimal-const} \\
\text{decimal-const} &\rightarrow \text{nonzero-digit} \mid \text{decimal-const digit} \\
\text{octal-const} &\rightarrow 0 \mid \text{octal-const octal-digit} \\
\text{hexadecimal-const} &\rightarrow \text{hexadecimal-prefix hexadecimal-digit} \\
&\mid \text{hexadecimal-const hexadecimal-digit} \\
\text{hexadecimal-prefix} &\rightarrow '0x' \mid '0X' \\
\text{nonzero-digit} &\rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\text{octal-digit} &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \\
\text{hexadecimal-digit} &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \\
&\mid 9 \mid a \mid b \mid c \mid d \mid e \mid f \mid A \mid B \mid C \mid D \mid E \mid F
\end{aligned}$$

float-const 主要有两种形式 (1) 数字序列与点结合; (2) 尾数与阶数结合。digit-sequence 即为数字序列, 而 signed-digit-sequence 表示有符号的数字序列, 该词项主要用于阶数的构成, 便于表示很大/很小的浮点数; fractional-part 与 exponent-part 分别表示浮点数的尾数与阶数, 其中考虑 .522 与 522. 分别为省略了整数 0 与尾数部分的浮点数, 同理认为 .522e-2 为浮点数。

CFG 5. float-const

$$\begin{aligned}
\text{float-const} &\rightarrow \text{fractional-part exponent-part} \\
\text{digit-sequence} &\rightarrow \text{digit} \\
&\mid \text{digit-sequence digit} \\
\text{signed-digit-sequence} &\rightarrow \text{digit-sequence} \\
&\mid '+' \text{ digit-sequence} \\
&\mid '-' \text{ digit-sequence} \\
\text{fractional-part} &\rightarrow . \text{ digit-sequence} \\
&\mid \text{digit-sequence} . \\
&\mid \text{digit-sequence} . \text{ digit-sequence} \\
\text{exponent-part} &\rightarrow e \text{ signed-digit-sequence} \\
&\mid E \text{ signed-digit-sequence} \mid \epsilon
\end{aligned}$$

4. 字符串常量

字符串是一个由零个或多个多字节字符组成的序列, 用引号括起来, 支持 raw text 与反义字符序列表达形式。CFG 中, string-character-set 指编码集除双引号、反斜杠以及换行符外支持的所有字符集合, escape-sequence 表示反义字符集合。

CFG 6. string-const

$$\begin{aligned}
\text{string-literal} &\rightarrow " \text{char-sequence} " \\
&| L " \text{char-sequence} " \\
\text{char-sequence} &\rightarrow \mathbf{char} \\
&| \text{char-sequence } \mathbf{char} \\
\text{char} &\rightarrow \text{string-character-set} \\
&| \text{escape-sequence} \\
\text{escape-sequence} &\rightarrow \backslash ' | \backslash " | \backslash ? | \backslash \backslash \\
&| \backslash a | \backslash b | \backslash f | \backslash n | \backslash r | \backslash t | \backslash v
\end{aligned}$$
(二) SysY 语言中非终结符 V_N 定义

非终结符号也被称为“语法变量”，是可以被取代的符号

| 非终结符 | 符号 | 非终结符 | 符号 | 非终结符 | 符号 |
|--------|-------------------|--------|--------------------|--------|---------------------|
| 编译单元 | <i>CompUnit</i> | 声明 | <i>Decl</i> | 常量声明 | <i>ConstDecl</i> |
| 基本类型 | <i>BType</i> | 常数定义 | <i>ConstDef</i> | 常量初值 | <i>ConstInitVal</i> |
| 变量声明 | <i>VarDecl</i> | 变量定义 | <i>VarDef</i> | 变量初值 | <i>InitVal</i> |
| 函数定义 | <i>FuncDef</i> | 函数类型 | <i>FuncType</i> | 函数形参表 | <i>FuncFParams</i> |
| 函数形参 | <i>FuncFParam</i> | 语句块 | <i>Block</i> | 语句块项 | <i>BlockItem</i> |
| 语句 | <i>Stmt</i> | 表达式 | <i>Exp</i> | 条件表达式 | <i>Cond</i> |
| 基本表达式 | <i>PrimaryExp</i> | 数值 | <i>Number</i> | 一元表达式 | <i>UnaryExp</i> |
| 单目运算符 | <i>UnaryOp</i> | 函数实参表 | <i>FuncRParams</i> | 乘除模表达式 | <i>MulExp</i> |
| 加减表达式 | <i>AddExp</i> | 关系表达式 | <i>RelExp</i> | 相等性表达式 | <i>EqExp</i> |
| 逻辑与表达式 | <i>LAndExp</i> | 逻辑或表达式 | <i>LOrExp</i> | 常量表达式 | <i>ConstExp</i> |

(三) SysY 语言中开始符号 S 定义

编译单元 *CompUnit* 为开始符号，是程序运行的起点，可由声明 *Decl*、函数定义 *FuncDef* 等组成。

CFG 7. CompUnit

$$\begin{aligned}
\text{CompUnit} &\rightarrow \text{CompUnit } \text{Decl} \\
&| \text{CompUnit } \text{FuncDef} | \text{Decl} | \text{FuncDef}
\end{aligned}$$
(四) SysY 语言中产生式集合 P 定义

为了表达方便，在这里我们沿用 SysY 文法定义采用的扩展的 Backus 范式表示，其中：

- 符号 [...] 表示方括号内包含的为可选项；
- 符号 {...} 表示花括号内包含的为可重复 0 次或多项的项；

1. 语句块 & 语句

Block 表示语句块，当语句块结束时块内声明的变量生存期也结束。当存在同名的全局变量与局部变量时，语句块隐藏块外声明，优先使用块内声明变量/常量直至语句块的作用域结束。

CFG 8. Block

$$\begin{aligned} Block &\rightarrow \{ ' \{ ' BlockItems ' \} ' \mid \{ ' ' \} ' \\ BlockItem &\rightarrow Decl \mid Stmt \end{aligned}$$

- 语句支持用';' 表示一条空语句，语句同样支持由单个表达式与';' 构成，表达式将被求值，但该值将不会储存。
- 在我们的编译器初步设计中，语句除了基本的空语句及表达式语句外，还可分为 Selection-Statement 选择语句、Iteration-Statement 迭代语句、Jump-Statement 跳转语句三种类型。

CFG 9. Statement

$$\begin{aligned} Stmt &\rightarrow LVal ' = ' Exp ';' \mid ';' \mid Exp ';' \mid Block \\ &\mid Selection-Statement \mid Iteration-Statement \mid Jump-Statement \end{aligned}$$

- return 语句支持返回表达式，同时也支持返回空。
- 语句中的 if 就近匹配，在没有约束作用域时仅对紧邻的下一条指令发生作用，else 拥有类似的性质，但当出现 else 时，其前必需出现 if，反之则不用。
- switch() 语句之后仅支持标签语句，用以表示不同条件下的语句执行情况。
- for(exp1;exp2;exp3)Stmt 语句支持循环迭代操作，其中 exp1 仅在语句开始时执行一次，exp2 从进入循环的第一次起进行判断，exp3 每完成一次循环才执行一次。

CFG 10. Basic-Statement

$$\begin{aligned} Selection-Statement &\rightarrow \mid 'if' '(' Cond ')' Stmt \\ &\quad \mid 'if' '(' Cond ')' Stmt 'else' Stmt \\ &\quad \mid 'switch' '(' Exp ')' Labeled-Statement \\ Iteration-Statement &\rightarrow 'while' '(' Cond ')' Stmt \\ &\quad \mid 'do' Stmt 'while' '(' Exp ')' ';' \\ &\quad \mid 'for' '(' ';' Exp ';' Exp ';' Exp ')' Stmt \\ Jump-Statement &\rightarrow 'break' ';' \mid 'continue' ';' \\ &\quad \mid 'return' ';' \mid 'return' Exp ';' \\ Labeled-Statement &\rightarrow 'default' ':' Stmt \\ &\quad \mid 'case' constant-expression ':' Stmt \end{aligned}$$

2. 函数

$FuncFParam$ 表示函数形参, $FuncRParams$ 则表示函数实参。函数参数支持 $BType$ 定义的所有类型及其数组, $[]$ 与 $[Exp]$ 为可选项, 支持利用 $Ident []$ 的方式传入数组, 同样支持利用 $Ident [Exp]$ 的形式传入数组内某一特定元素, 同理, 支持 $Ident [Exp][Exp]$ 传入多维数组元素。

CFG 11. FuncParam

$$\begin{aligned} FuncRParams &\rightarrow Exp \mid FuncRParams \text{ ',' } Exp \\ FuncFParams &\rightarrow FuncFParam \mid FuncFParams \text{ ',' } FuncFParam \\ FuncFParam &\rightarrow BType \text{ Ident } \mid BType \text{ Ident ' [' ']' } \\ &\quad \mid FuncFParam \text{ ArrayIndices } \\ ArrayIndices &\rightarrow ArrayIndex \\ &\quad \mid ArrayIndices \text{ ArrayIndex } \\ ArrayIndex &\rightarrow \text{ '[' } Exp \text{ ']' } \end{aligned}$$

$FuncDef$ 表示函数定义。 $FuncType$ 表示返回类型, 当函数类型为 `void` 时, 返回值必须为空, 即 `return;`; 当函数类型为 `int` 或 `float` 时, 函数内应返回对应的数据类型值。而 $FuncFParams$ 表示函数的参数列表。

CFG 12. FuncDef

$$\begin{aligned} FuncDef &\rightarrow FuncType \text{ Ident ' (' ')' } Block \\ &\quad \mid FuncType \text{ Ident ' (' } FuncFParams \text{ ')' } Block \\ FuncType &\rightarrow \text{ 'void' } \mid \text{ 'int' } \mid \text{ 'float' } \end{aligned}$$

3. 输入输出

$SysY$ 输出输出的功能是通过链接 $SysY$ 运行时库实现的, 这些库函数不用在 $SysY$ 程序中声明就可以在 $SysY$ 的函数中使用。

需要注意的是, 当运行对应的 I/O 函数时, 传递的参数 $FuncRParams$ 要与库函数中对应 I/O 函数声明语句中的 $FuncFParam$ 类型一致。

比如当程序中调用 `putfarray` 函数时, 需要传递的形参 $FuncFParam$ 为 `int n, float a[]`, 对应传入的实参也应当是这种类型的。

4. 变量 & 常量声明与初始化

常量声明 $ConstDecl$ 和变量声明 $VarDecl$ 的基本格式类似, 但是常量声明需要在对应的声明类型 $BType$ 前面添加关键字 `const`, 在 $SysY$ 语言中我们选取的类型 $BType$ 是 `int` 和 `float` 类型;

值得注意的是, 在 $SysY$ 语言中常量是在声明的时候就进行初始化, 而变量可以只声明不初始化, 可以在常量定义 $ConstDef$ 和变量定义 $VarDef$ 的产生式中看到两者的差异。

另外, 在这里我们也对数组的声明和初始化进行了定义, 使用了上面提到的 EBNS 范式。

为了方便比较, 我们在这里将常量和变量的上面及初始化定义分别描述:

CFG 13. const-declaration

$$\begin{aligned}
Decl &\rightarrow ConstDecl \mid VarDecl \\
ConstDecl &\rightarrow \text{'const'} BType ConstDef \{ \text{'}, ConstDef \} \text{'};' \\
ConstDef &\rightarrow \mathbf{Ident} \{ \text{'[' ConstExp ']' } \text{'=' ConstInitVal} \\
ConstInitVal &\rightarrow ConstExp \\
&\mid \text{'{' [ConstInitVal \{ \text{'}, ConstInitVal \}] \text{'}}'
\end{aligned}$$

语义说明:

1. 在进行初始化的时候, 我们需要注意其中前后表达式的类型和结构之间的对应关系, 比如在声明时 $BType$ 的类型为 **int** 类型的, 那么对应的初始化值 $ConstInitVal$ 就不能是数组形式的。当然 SysY 同样支持隐式数据类型的转换, 例如从 **float** 转化为 **int** 时, 小数部分将被丢弃。
2. 这里额外说明常量数组的声明和初始化, 在 SysY 语言中, 常量数组声明时各个维度的值 $ConstExp$ 必须显示给出, 这里表示的 $ConstExp$ 是可以在编译时就直接计算得到的。在初始化时, 常量数组其中的值 $ConstInitVal$ 也必须是确定的值, 其中初始化数组的长度不能超过声明数组的长度。

CFG 14. var-declaration

$$\begin{aligned}
VarDecl &\rightarrow BType VarDef \{ \text{'}, VarDef \} \text{'};' \\
VarDef &\rightarrow \mathbf{Ident} \{ \text{'[' ConstExp ']' } \\
&\mid \mathbf{Ident} \{ \text{'[' ConstExp ']' } \text{'=' InitVal} \\
InitVal &\rightarrow Exp \\
&\mid \text{'{' [InitVal \{ \text{'}, InitVal \}] \text{'}}'
\end{aligned}$$

语义说明: 我们可以对照上面实现的常量的声明与初始化来体会变量的声明与初始化:

1. 同样地, 在进行初始化的时候, 我们需要注意其中前后表达式的类型和结构之间的对应关系。比如在声明变量语句 **int a;** 时 $VarDef$ 对应的产生式是 $VarDef \rightarrow \mathbf{Ident}$, 而在之后的变量的定义当中对应的产生式应当为 $VarDef \rightarrow \mathbf{Ident} \text{'=' InitVal}$
2. 同样对于数组的声明, 数组中各个维度的值 $ConstExp$ 必须显示给出。不同之处在于, 在初始化时, 变量数组其中的值 $InitVal$ 也可以是变量, 不一定为确定的值。
3. 全局变量如果没有进行初始化, 默认的初始化为 0 或 0.0, 而没有初始化的变量初值是不确定的。

5. 表达式

表达式简单分类包括算术运算 (+、-、*、/、%) 表达式、关系运算 (==, >, <, >=, <=, !=) 表达式和逻辑运算 (&& (与)、|| (或)、! (非)) 表达式, 在定义时我们需要注意各个运算符的优先级, 根据运算符的优先级排序我们可以定义对应的表达式:

1. 括号 () 定义基本表达式 $PrimaryExp$

2. 单目运算符 !(非) -(负号) 定义一元表达式 *UnaryExp*
3. 算术运算符的乘法、除法、余数 * / % 定义乘除模表达式 *MulExp*
4. 算术运算符的加法、减法、前缀自增自减 + - ++ - 定义加减表达式 *AddExp*
5. 关系运算符的大于、大于等于、小于、小于等于 > >= < <= 定义关系表达式 *RelExp*
6. 关系运算符的等于、不等于 == != 定义相等性表达式 *EqExp*
7. 逻辑运算 AND && 定义逻辑与表达式 *LAndExp*
8. 逻辑运算 OR || 定义逻辑或表达式 *LOrExp*
9. 条件控制运算符 ?: 定义三目表达式 *conditional-expression*
10. 赋值运算符 定义赋值运算符 *assignment-expression*

在实际的使用过程中,除了运算符优先级的考量之外,我们还需要考虑一些其它因素,比如函数各种语句,需要我们对表达式进行更加全面的扩充,比如我们定义了针对条件控制语句的条件表达式 *Cond*, 定义了针对赋值运算的可修改的左值表达式 *LVal*, 定义了最简单的没有操作符的表达式基本表达式 *PrimaryExp*, 定义了包含一元运算符的一元表达式 *UnaryExp*²。

CFG 15. other-expression

$$\begin{aligned} Exp &\rightarrow AddExp \\ Cond &\rightarrow LOrExp \\ LVal &\rightarrow \mathbf{Ident} \{ '[Exp]' \} \\ PrimaryExp &\rightarrow '(Exp)' | LVal | Number \\ Number &\rightarrow \mathbf{IntConst} | \mathbf{floatConst} \end{aligned}$$

CFG 16. unary-expression

$$\begin{aligned} UnaryExp &\rightarrow PrimaryExp | \mathbf{Ident} '([FuncRParams])' \\ &\quad | UnaryOp UnaryExp \\ UnaryOp &\rightarrow '+' | '-' | '!' | '++' | '--' \\ FuncRParams &\rightarrow Exp \{ ', Exp \} \end{aligned}$$

CFG 17. arithmetic-expression

$$\begin{aligned} MulExp &\rightarrow UnaryExp | MulExp '*' UnaryExp \\ &\quad | MulExp '/' UnaryExp | MulExp '%' UnaryExp \\ AddExp &\rightarrow UnaryExp \\ &\quad | AddExp '+' UnaryExp | MulExp '-' UnaryExp \end{aligned}$$

²关于表达式的定义参考了博文 [C 语言基本概念之表达式](#)

CFG 18. relation-expression

$$\begin{aligned}
 RelExp &\rightarrow AddExp \\
 &| RelExp '<' AddExp \mid RelExp '>' AddExp \\
 &| RelExp '<=' AddExp \mid RelExp '>=' AddExp \\
 EqExp &\rightarrow RelExp \\
 &| EqExp '==' EqExp \mid EqExp '===' EqExp
 \end{aligned}$$
CFG 19. logistic-expression

$$\begin{aligned}
 LAndExp &\rightarrow EqExp \\
 &| LAndExp '&&' EqExp \\
 LOrExp &\rightarrow LAndExp \\
 &| LOrExp '||' LAndExp
 \end{aligned}$$
CFG 20. conditional-expression

$$\begin{aligned}
 conditional-expression &\rightarrow LOrExp \\
 &| LOrExp '?' LOrExp : conditional-expression
 \end{aligned}$$
CFG 21. assignment-expression

$$\begin{aligned}
 assignment-expression &\rightarrow conditional-expression \\
 &| UnaryExp assignment-operator assignment-expression \\
 assignment-operator &\rightarrow '=' \mid '*=' \mid '/=' \mid '%=' \mid '- =' \\
 &| '<<=' \mid '>>=' \mid '&=' \mid '^=' \mid '|=' \\
 expression &\rightarrow assignment-expression \\
 &| expression ',' assignment-expression \\
 ConstExp &\rightarrow AddExp
 \end{aligned}$$

三、 ARM 汇编程序

(一) 程序一

程序一主要测试了函数、浮点数、数组、跳转语句 if()else 及循环语句 for(;;) 部分的特性, C 代码如下:

程序一 C 语言程序

```

1  #include<stdio.h>
2  int N=1;
3  float func(float in []) {
4      float res=0;

```

```

5      for(int i=0;i<5;i++){
6          if(i%2==0)
7              res+=in[i]/10;
8          else
9              res+=0.5;
10     }
11     return res;
12 }
13 int main(){
14     int N=10;
15     float input[5]={N,9,8,7,6};
16     float result = func(input);
17     printf("%f \n",result);
18     return 0;
19 }

```

在手写汇编代码时遇到的最主要的问题如下:

1. **浮点数表示错误**: 在一开始写汇编的时候, 试图通过 0.5 之类的形式来给寄存器赋值, 结果赋值失败, 后来通过改写为浮点数的 IEEE 754 标准进行了赋值。(不过似乎使用版本近一些交叉编译器也可以用阶码表示)
2. **对于数组的运算不熟练**: 由于在循环中利用下标获取数组中元素的同时, 还需要获取下标的当前值, 导致计算偏移的时候总是出错, 并且中途还忘记将字节地址转换为字地址。
3. **完全没有提前给出常量值的意识**: 两个 NUM 段都是对照着 gcc 版本才更改过来的, 甚至没想到这也能算偏移 (类似的, 如果参照版本更新一点的编译器, 会发现其更倾向于在指令中直接指定)。

程序一汇编程序

```

1 N: @ 全局变量N
2     .word    1
3 NUM: @ 给除法准备的常量10
4     .word    1092616192
5 func(float*):
6     push     {fp, lr} @ 保护现场
7     add      fp, sp, #4 @ 更新fp, 转移栈帧
8     sub      sp, sp, #16 @ 给局部变量开空间, float为四个字节, 所以-16
9     str      r0, [fp, #-16] @ 将r0中的字数据写入以fp-16为地址的存储器中
10    @ float res = 0;
11    mov      r3, #0
12    str      r3, [fp, #-8]
13    @ int i = 0;
14    mov      r3, #0
15    str      r3, [fp, #-12]
16 LOOP: @ 不是for开始的地方, 但是loop的起始点
17    @ 循环条件判断
18    ldr      r3, [fp, #-12]

```

```

19      cmp     r3, #4
20      bgt     OUT
21      @ if i%2==0
22      ldr     r3, [fp, #-12]
23      and     r3, r3, #1 @ 这里学了一下gcc, 其实比比奇偶确实and一下就可以了
24      cmp     r3, #0
25      bne     ELSE @ 不满足则跳转到else
26      @ 这一小段是数组寻址
27      ldr     r3, [fp, #-12] @ 加载i
28      lsl     r3, r3, #2 @ 变成字地址
29      ldr     r2, [fp, #-16] @ 找数组首地址
30      add     r3, r2, r3 @ 找到下标指定的元素
31      ldr     r3, [r3] @ 加载元素
32      @ 除法运算
33      ldr     r1, NUM
34      mov     r0, r3
35      bl      __aeabi_fdiv
36      mov     r3, r0
37      @ 计算并返回最终结果
38      mov     r1, r3
39      ldr     r0, [fp, #-8]
40      bl      __aeabi_fadd
41      mov     r3, r0
42      str     r3, [fp, #-8]
43      b       BACK
44  ELSE:    @ 计算res+=0.5
45      mov     r1, #1056964608 @ 这里#1056964608表示0.5, 应为IEEE 754浮点数标准形式
46      ldr     r0, [fp, #-8]
47      bl      __aeabi_fadd
48      mov     r3, r0
49      str     r3, [fp, #-8]
50  BACK:    @ 完成for中的i++
51      ldr     r3, [fp, #-12]
52      add     r3, r3, #1
53      str     r3, [fp, #-12]
54      b       LOOP
55  OUT:
56      ldr     r3, [fp, #-8] @ return res
57      @ 弹出栈帧
58      mov     r0, r3
59      sub     sp, fp, #4
60      pop     {fp, lr}
61      bx      lr
62  .LC0:
63      .ascii  "%f \012\000"
64  NUMS: @ 全部为浮点数
65      .word   1091567616

```

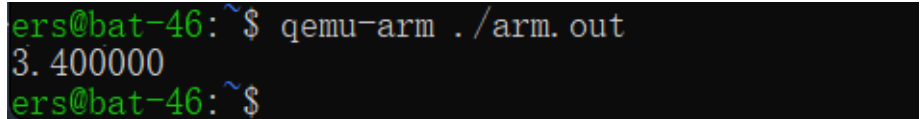
```

66      .word    1088421888
67      .word    1086324736
68      .word    .LC0
69  main:
70      push     {r4, fp, lr}
71      add      fp, sp, #8
72      sub      sp, sp, #36
73      @ int N=10
74      mov      r3, #10
75      str      r3, [fp, #-16]
76      @ 开始浮点数组的初始化
77      sub      r3, fp, #40 @ 分配空间
78      mov      r2, #0
79      str      r2, [r3]
80      str      r2, [r3, #4]
81      str      r2, [r3, #8]
82      str      r2, [r3, #12]
83      str      r2, [r3, #16]
84      @ 一边找定义好的常数，一边移动指针，一边给数组赋值
85      ldr      r3, NUMS
86      str      r3, [fp, #-36]
87      mov      r3, #109051904
88      str      r3, [fp, #-32]
89      ldr      r3, NUMS+4
90      str      r3, [fp, #-28]
91      ldr      r3, NUMS+8
92      str      r3, [fp, #-24]
93      ldr      r0, [fp, #-16]
94      bl       __aeabi_i2f
95      mov      r3, r0
96      str      r3, [fp, #-40]
97      @ 调用func函数
98      sub      r3, fp, #40
99      mov      r0, r3
100     bl       func(float*) @ 跳转到func的位置
101     str      r0, [fp, #-20]
102     @ 开始printf函数调用
103     ldr      r0, [fp, #-20]
104     bl       __aeabi_f2d
105     mov      r3, r0
106     mov      r4, r1
107     mov      r2, r3
108     mov      r3, r4
109     ldr      r0, NUMS+12
110     bl       printf @ 跳转到printf的位置
111     mov      r3, #0 @ return 0
112     @ 栈帧切换
113     mov      r0, r3

```

```
114      sub    sp, fp, #8
115      pop    {r4, fp, lr}
116      bx     lr
```

程序运行成功截图：



```
ers@bat-46:~$ qemu-arm ./arm.out
3.400000
ers@bat-46:~$
```

图 1: 程序一运行成功截图

(二) 程序二

程序二主要测试了函数、常量与变量的声明和初始化、表达式 (算术、逻辑、关系)、输入输出等部分的特性, C 代码如下:

程序二 C 语言程序

```
1  const int initVal = 1;
2  int length = 3;
3
4  int fibonacci(int n){
5      if(n==1||n==2) return initVal;
6      else return fibonacci(n-1)+fibonacci(n-2);
7  }
8
9  int main() {
10     int a,b;
11     int cal,log,ral;
12     a = getint();
13     b = fibonacci(a);
14     cal = a + b - b*2 + a/3 ;
15     log = a > 5 && (b > 10 || a > b);
16     ral = a >= b || (!a && (a + b) > 20 ) ;
17     putint(b);
18     printf("\ncal = a + b - b*2 + a/3 = %d\n",cal);
19     printf("log = a > 5 && (b > 10 || a > b) = %d\n",log);
20     printf("ral = a >= b || (!a && (a + b) > 20 ) = %d\n",ral);
21     return 0;
22 }
```

在这一部分的汇编程序代码的编写时需要注意以下几点:

1. **整数除法取模运算**: 在进行这一部分的编写时, 笔者采用了循环语句的形式, 使得最后的计算结果符合整数除不尽下舍弃小数部分, 当然在实际的 ARM 汇编程序中不是这样计算的;
2. **函数调用过程**: 调用函数时首先要保存上下文, 函数的参数一般由 r0-r3 三个寄存器给出, 最后函数返回值若无特殊说明一般都是用 r0 来返回的。

3. **输入输出语句的调用**：这一部分是笔者编写过程中频繁出错的地方，在通常，ARM 的汇编中 `printf` 语句输出字符串是需要动态计算字符串的位置的³，使得这一部分表达不是特别直观；为了简化代码的书写与方便理解，笔者这里将字符串加入到代码段中，直接输出字符串，但在实际过程中字符串应当是在只读数据段的！

为了规范起见，所有的注释都用 '@' 开头，所有的跳转标签均用 '.' 开头。

程序一汇编程序

```

1  .arch armv7-a
2  @ 数据段
3  @ 全局变量及常量的声明
4  @ const int initVal = 1;
5  .data
6  .global      initVal
7  @这里我们常量声明用只读数据表示和变量做区分
8  .section      .rodata
9  .align 2
10 @我们将符号initVal的类型设置为object
11 .type      initVal, %object
12 @指定initVal变量为4个字节
13 .size      initVal, 4
14 initVal:
15 .word      1
16 @ int length = 3;
17 .data
18 .global length
19 .align 2
20 .type      length, %object
21 .size      length, 4
22 length:
23 .word      3
24
25 .text
26 @字符串同样声明为只读数据
27 .section      .rodata
28 .align 2
29 _str0:
30 .ascii     "\012cal = a + b - b*2 + a/3 = %d\012\000"
31 .align 2
32 _str1:
33 .ascii     "log = a > 5 && (b > 10 || a > b) = %d\012\000"
34 .align 2
35 _str2:
36 .ascii     "ral = a >= b || (!a && (a + b) > 20 ) = %d\012\000"
37
38 @ 代码段
39 @ fibonacci函数

```

³参考博文 [android ARM 汇编学习——hello world](#)


```

40 @ int fibonacci(int n)
41 .text
42 .global fibonacci
43 @ 定义类型为函数
44 .type        fibonacci, %function
45 fibonacci:
46 @ 从左到右压栈r4 r7 lr 保存寄存器上下文
47 push        {r4, r7, lr}
48 sub         sp, sp, #12
49 @ 传入实参n
50 add         r7, sp, #0
51 str         r0, [r7, #4]
52 ldr         r3, [r7, #4]
53 @ n == 1
54 cmp         r3, #1
55 beq         .L1
56 @ n == 2
57 ldr         r3, [r7, #4]
58 cmp         r3, #2
59 bne         .L2
60 .L1:
61 @ 常量initVal替换为对应的值1
62 movs        r3, #1
63 b           .L3
64 .L2:
65 @ fibonacci(n-1)
66 ldr         r3, [r7, #4]
67 subs        r3, r3, #1
68 @ 传入参数
69 mov         r0, r3
70 bl          fibonacci
71 @ 函数返回值直接存入r0
72 mov         r4, r0
73 @ fibonacci(n-1)
74 ldr         r3, [r7, #4]
75 subs        r3, r3, #2
76 mov         r0, r3
77 bl          fibonacci
78 mov         r3, r0
79 @ 计算得到函数返回值
80 add         r3, r3, r4
81 .L3:
82 mov         r0, r3
83 adds        r7, r7, #12
84 mov         sp, r7
85 @ 恢复上下文
86 pop         {r4, r7, pc}
87

```

```

88 @ 代码段
89 @ main函数
90 @ int main()
91     .text
92     .align 1
93     .global main
94     .type main, %function
95 main:
96     @ 保存寄存器上下文
97     push    {r7, lr}
98     sub     sp, sp, #24
99     add     r7, sp, #0
100    @ 调用 getint()
101    bl      getint
102    @ a = getint();
103    str     r0, [r7, #4]
104    ldr     r0, [r7, #4]
105    @ b = fibonacci(a);
106    bl      fibonacci
107    str     r0, [r7, #8]
108    ldr     r2, [r7, #4]
109    ldr     r3, [r7, #8]
110    @ cal = a + b - b*2 + a/3 ;
111    @ 计算 a = a + b;
112    add     r2, r2, r3
113    ldr     r3, [r7, #8]
114    @ b = b << 1
115    lsls    r3, r3, #1
116    @ r1 = a - b;
117    subs    r1, r2, r3
118    ldr     r2, [r7, #4]
119    mov     r3, #0
120    b .CMPDIV
121    @ 整数除法
122    .INTDIV:
123    sub     r2, r2, #3
124    add     r3, r3, #1
125    .CMPDIV:
126    cmp     r2, #0
127    bgt     .INTDIV
128    @ cal = r1 + r3
129    add     r3, r3, r1
130    str     r3, [r7, #12]
131    @ 计算 log = a > 5 && (b > 10 || a > b);
132    @ 先判断 a > 5 短路原理
133    ldr     r3, [r7, #4]
134    cmp     r3, #5
135    ble     L5 @ ble 小于等于

```

```

136      @ a > 5为True 判断 b > 10
137      ldr      r3, [r7, #8]
138      cmp      r3, #10
139      bgt      .L6
140      @ a > b
141      ldr      r2, [r7, #4]
142      ldr      r3, [r7, #8]
143      cmp      r2, r3
144      ble      .L5
145  .L6: @ True
146      movs     r3, #1
147      b        .L7
148  .L5: @ False
149      movs     r3, #0
150  .L7:
151      str      r3, [r7, #16]
152      @ r1 = a >= b || (!a && (a + b) > 20 );
153      ldr      r2, [r7, #4]
154      ldr      r3, [r7, #8]
155      @ a >= b
156      cmp      r2, r3
157      bge      .L8
158      @ !a
159      ldr      r3, [r7, #4]
160      cmp      r3, #0
161      bne      .L9
162      @ (a + b) > 20
163      ldr      r2, [r7, #4]
164      ldr      r3, [r7, #8]
165      add      r3, r3, r2
166      cmp      r3, #20
167      ble      .L9
168  .L8: @ True
169      movs     r3, #1
170      b        .L10
171  .L9: @ False
172      movs     r3, #0
173  .L10:
174      str      r3, [r7, #20]
175      ldr      r0, [r7, #8]
176      @ putint(b);
177      bl       putint
178      @ printf("\ncal = a + b - b*2 + a/3 = %d\n",cal);
179      ldr      r1, [r7, #12]
180      adr      r0, __str3
181      bl       printf
182      ldr      r1, [r7, #16]
183      adr      r0, __str4

```

```

184     bl    printf
185     ldr   r1, [r7, #20]
186     adr   r0, __str5
187     bl    printf
188     @ return 0
189     movs   r3, #0
190     mov    r0, r3
191     adds   r7, r7, #24
192     mov    sp, r7
193     @ sp needed
194     pop    {r7, pc}
195
196 __str3:
197     .ascii "\012cal = a + b - b*2 + a/3 = %d\012\000"
198 __str4:
199     .ascii "log = a > 5 && (b > 10 || a > b) = %d\012\000"
200 __str5:
201     .ascii "ral = a >= b || (!a && (a + b) > 20 ) = %d\012\000"
202 __bridge:
203     .word  __str0
204     .word  __str1
205     .word  __str2
206     .section .note.GNU-stack,"",%progbits

```

注意使用 SysY 输入输出时需要链接 SysY 运行时库，执行下面的指令：

```

1 arm-linux-gnueabi-hf-gcc test2.S -o target.out libsysy.a
2 qemu-arm ./target.out

```

程序运行成功截图：

```

xiaoduo@LAPTOP-OH49GLN2:~/CompileSpace/Lab2/arm_sublab$ arm-linux-gnueabi-hf-gcc test2.S -o target.out libsysy.a
xiaoduo@LAPTOP-OH49GLN2:~/CompileSpace/Lab2/arm_sublab$ qemu-arm ./target.out
5
5
cal = a + b - b*2 + a/3 = 2
log = a > 5 && (b > 10 || a > b) = 0
ral = a >= b || (!a && (a + b) > 20 ) = 1
TOTAL: 0H-0M-0S-0us
xiaoduo@LAPTOP-OH49GLN2:~/CompileSpace/Lab2/arm_sublab$

```

图 2: 程序二运行成功截图

四、 语法制导翻译实现从 SysY 到汇编

在本小节中，我们编写了一个简单的基于 for 循环的程序样例，并绘制了相应的示意图来解释具体语法制导翻译实现 SysY 语言到汇编的流程，SysY 代码样例如下：

for 循环样例代码

```

1 int a = 0;
2 int b = 1;
3 for(int i = 0; i < 10; i++){
4     a = b + i;

```

5 } }

在词法分析和语法分析阶段，编译器会帮助我们构建符号表，定义符号表结构体如下，用以保存常量/变量标识符名、具体值、生成代码及相应的地址。在本示例当中，我们简化了符号表的设计，仅保留了标识符名称以及对应变量相应的地址。

符号表结构体

```
1 typedef struct
2 {
3     char iden[50];
4     float num;
5     long addr;
6     char code[1000];
7 }symTable;
```

接下来给出 for 语句的语法生成树及变量对应的符号表描述，当汇编程序需要用到相应的变量时，需要利用 ldr 找到变量存储的地址，并通过 str 语句进行存储。

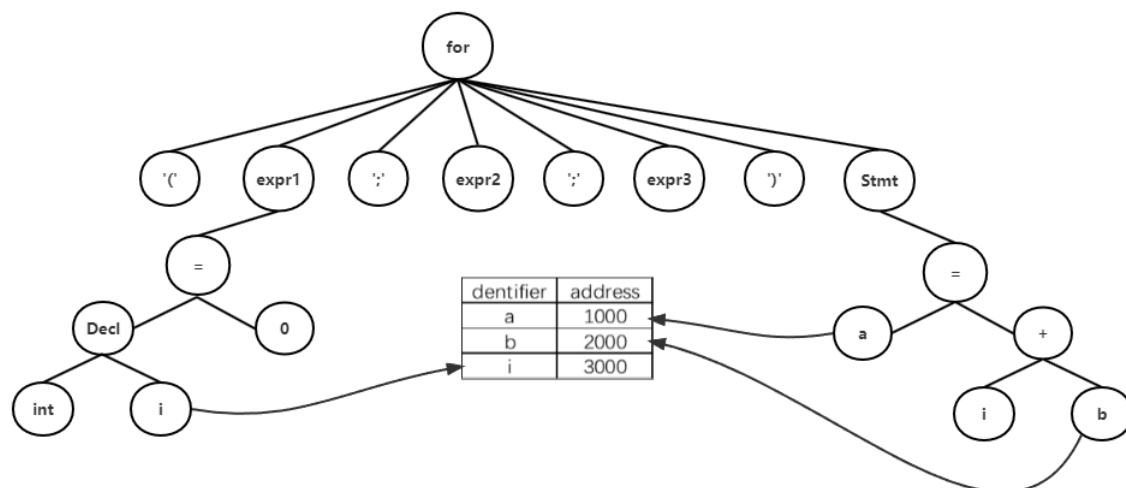


图 3: 连接符号表

CFG 1. for 控制流语句翻译

$$\begin{aligned}
 stmt &\rightarrow for (expr1; expr2; expr3) stmt1 \\
 \{ & \text{LOOP} = \text{newlabel}; \\
 & \text{OUT} = \text{newlabel}; \\
 & \text{BACK} = \text{newlabel}; \\
 stmt.code &:= expr1.code \parallel b \text{ BACK} \parallel \text{LOOP} \\
 & \parallel stmt1.code \parallel expr3.code \parallel \text{BACK} \\
 & \parallel expr2.code \parallel b \text{ LOOP} \parallel \text{OUT} \}
 \end{aligned}$$

构建语法制导翻译的语法生成树如下，我们在每一个语句下面添加了对应的汇编代码的翻译：

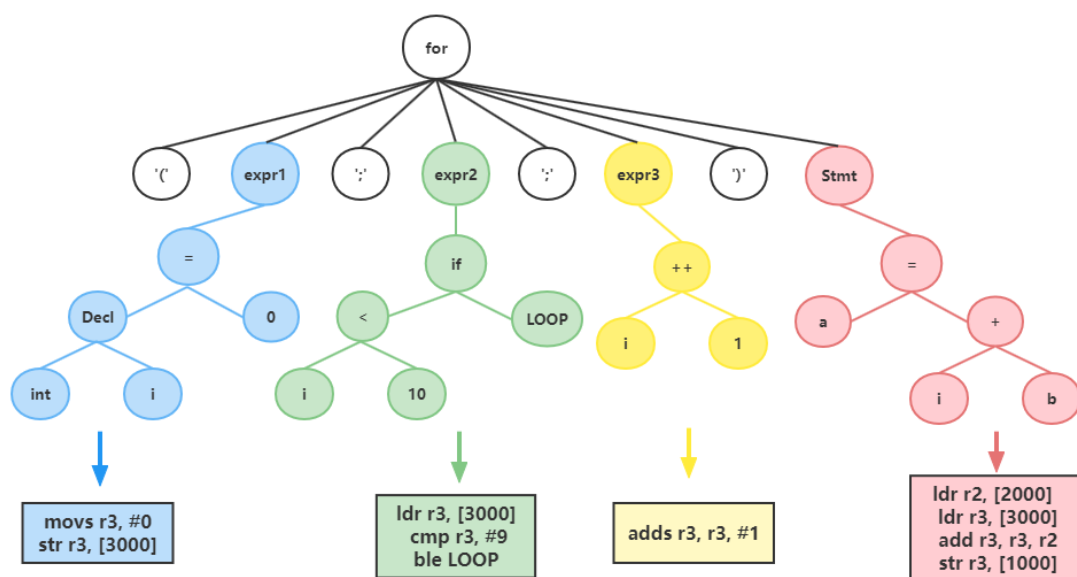


图 4: 语法生成树

结合 for 控制流语句翻译以及上面构建的语法生成树，我们可以梳理出整个 for 控制流语句的执行流程，从而编写出对应的汇编代码，如下所示为我们提供的样例程序进行语法制导翻译生成汇编代码的控制流图：

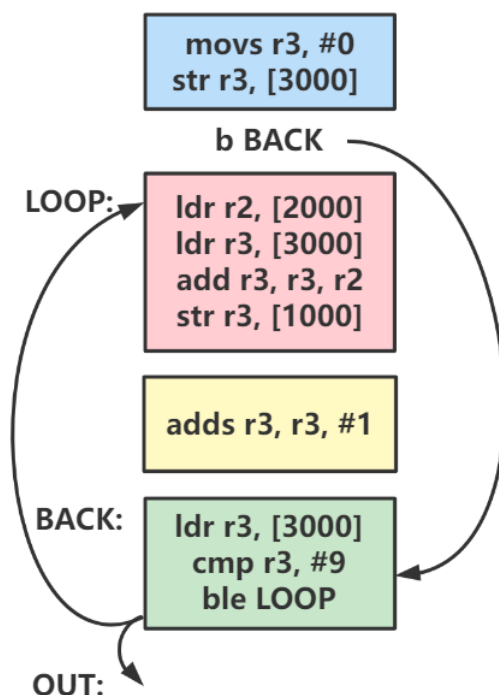


图 5: 汇编代码控制流图

结合图示以及 for 循环控制流语句的翻译，从图中我们可以发现 for 循环语句翻译为汇编代码的要注意以下几点：

1. 从汇编代码的控制流图可以看到 for 循环的 expr1 语句中对应的初始化语句只需要执行一次，在翻译时要放到循环的外边；
2. 在每次进入 for 循环内部的循环体 LOOP 之前，都需要先执行 BACK 语句块中的内容，满足条件才能进入循环内部，在翻译时要注意先 ‘b BACK’ 满足条件再进入循环（对应 ‘ble LOOP’），不满足条件则退出循环（OUT）；
3. 每次执行完循环体 LOOP 都需要执行 for 循环内部的 expr3 语句，顺次执行 BACK 语句块中内容重复上述过程。

最后附上根据上面的汇编代码的控制流图，生成的汇编程序代码：

最终生成汇编程序代码

```

1 ;for 循环之前的语句
2     movs    r3, #0
3     str     r3, [1000] ;int a = 0;
4     movs    r3, #1
5     str     r3, [2000] ;int b = 1;
6 ;for 循环中 expr1 表达式 int i = 0
7     movs    r3, #0
8     str     r3, [3000]
9     b       BACK
10 ;for 循环中循环体内部的 stmt1 语句 a = b + i;
11 LOOP:
12     ldr     r2, [2000]
13     ldr     r3, [3000]
14     add     r3, r3, r2
15     str     r3, [1000]
16 ;for 循环中 expr3 表达式 i++
17     ldr     r3, [3000]
18     adds    r3, r3, #1
19     str     r3, [3000]
20 ;for 循环中 expr2 表达式 i < 10
21 BACK:
22     ldr     r3, [3000]
23     cmp     r3, #9
24     ble     LOOP
25 ;for 循环出口位置
26 OUT:

```

五、 总结

路漫漫其修远兮，吾将上下而造编译器。