南开大学
Nankai University

南　　开　　大　　学

计 算 机 学 院 和 网 络 空 间 安 全 学 院

编译系统原理第二次实验报告

定义你的编译器 & 汇编编程

姓名：彭钰钊　姜涵

学号：2110756　2113630

年级：2021 级

专业：计算机科学与技术　信息安全

指导教师：王刚

2023 年 10 月 10 日

摘要

本次实验是在上一次预备工作的基础上进一步确定我们要实现的编译器支持哪些 SysY 语言特性，给出其形式化定义，即上下文无关文法的定义。同时设计几个 SysY 程序，编写等价的 ARM 汇编程序，用汇编器生成可执行程序，调试通过、能正常运行得到正确结果。这些程序应该尽可能全面地包含我们要支持的语言特性。

在实验分工方面，我们的第一部分工作按不同的语言特性进行分工：由姜涵负责数据类型、变量/常量声明与初始化、语句与函数部分的内容，由彭钰钊负责表达式、注释部分的内容，并且由两人一同检查并修改整体的 CFG 表达；第二部分工作按照 SysY 的不同语言特性进行分工：由姜涵负责整数运算、循环、函数内容，由彭钰钊负责浮点数、分支、函数声明与调用的内容。

关键字：编译原理、SysY 语言、上下文无关文法、ARM 汇编

目录

一、 定义编译器

(一) SysY 语言概述

SysY 语言 [1] 是 C 语言一个子集的扩展，其语法和结构与 C 语言类似，基本实现了 C 语言的相关语法；但是 SysY 语言本身没有提供输入/输出 (I/O) 的语言构造，I/O 是以运行时库方式提供，库函数可以在 SysY 程序中的函数内调用。

SysY 语言**在数据类型上**支持 int(32 位有符号整数)/float(32 位单精度浮点数) 类型以及以这两种类型为元素且按行优先存储的多维数组；**在函数实现上**，能够实现带参/无参函数，可以返回 int/float 类型的值，也可以不返回值，同时参数传递与 C 语言相似，有值传递和址传递（形参只有第一维的长度可以缺省）；**在变量/常量声明上**，需要先定义再使用；**在语句类型上**，包括赋值语句、表达式语句 (表达式可以为空)、语句块、if 语句、while 语句、break 语句、continue 语句、return 语句，其中，表达式支持基本的算术运算（+、-、*、/、%）、关系运算（==、!=、<、>、<=、>=）和逻辑运算（!、&&、||），非 0 表示真、0 表示假，而关系运算或逻辑运算的结果用 1 表示真、0 表示假。

(二) 上下文无关文法设计

1. SysY 语言特性

基础 track：

- 数据类型：int、float
- 变量声明、常量声明，常量、变量的初始化
- 语句：赋值（=）、表达式语句、语句块、if、while、return
- 表达式：算术运算（+、-、*、/、%，其中 +、-都可以是单目运算符）、关系运算（==，>，<，>=，<=，!=）和逻辑运算（&&（与）、||（或）、！（非））
- 注释
- 输入输出

进阶 track：

- 函数、语句块
  - 函数：函数声明、函数调用
  - 变量、常量作用域：在函数中、语句块（嵌套）中包含变量、常量声明的处理，break、continue 语句
- 数组：数组（一维、二维、…）的声明和数组元素访问
- 代码优化
  - 寄存器分配优化方法
  - 基于数据流分析的强度削弱、代码外提、公共子表达式删除、无用代码删除等

2. CFG 描述

一般定义：四元组表示——(V_T, V_N, S, P)

- 一组**终结符号**；也称为“词法单元”——V_T
- 一组**非终结符号**；也称为“语法变量”——V_N
- 一组**产生式**：产生式头（左部）【非终结符】→ 产生式体（右部）

  P ： 产生式集合（有限集）

  每个产生式形式 A→ ，其中 $A \in V_N, \alpha \in (V_T \cup V_N)$

  关于 A 的产生式 S 至少在某个产生式左部出现一次
- 一个特定的非终结符——S

下面给出我们的 SysY 语言上下文无关文法描述。

1. 数据类型

   Type → **int** | **float**

2. 变量声明、常量声明，常量、变量的初始化

   - 变量声明

     Idlist → *Idlist*, **id** | **id**

     Decl → *Type Idlist*
   - 变量初始化

     Vari-init → *Type* **id** = *Exp*
   - 常量声明及初始化

     Con-decl → CONST *Type* **id**=*literal*

3. 语句：赋值（=）、表达式语句、语句块、if、while、return

   - 赋值语句

     Stmt → *LVal* '=' *Exp*';'
   - 表达式语句

     Stmt → [*Exp*]
   - 语句块

     Stmt →*Block*

     Block → '{' { *BlockItem* } '}'

     BlockItem → *Decl* | *Stmt*
   - if 语句

     Stmt → **if**(*Cond*) *Stmt* [**else** *Stmt*]
   - while 语句

     Stmt → **while** (*Cond*) *Stmt*
   - return 语句

     Stmt → **return** [*Exp*]

4. 表达式

- 表达式

  $Exp \rightarrow AddExp$

- 数值

  $Number \rightarrow [\ \textbf{-}\ ]\ Digit\ \{\ Digit\ \}\ [\ \textbf{.}\ Digit\ \{\ Digit\ \}\ ]$

  $Digit \rightarrow \textbf{0} \mid \textbf{1} \mid ... \mid \textbf{9}$

- 基本表达式

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

- 左值表达式

  $LVal \rightarrow \textbf{id}$ {'[' Exp ']'}

- 算数表达式 $AddExp \rightarrow MulExp \mid AddExp$ '+' $MulExp \mid AddExp$ '−' $MulExp$

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

- 一元表达式

  $UnaryExp \rightarrow PrimaryExp \mid \textbf{id}$ '(' $[ParameterList]$ ')' $\mid UnaryOp\ UnaryExp$

- 单目运算符

  $UnaryOp \rightarrow$ '+' | '−' | '**!**' 【注：'!' 仅出现在条件表达式中】

- 关系表达式

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

  $EqExp \rightarrow RelExp \mid EqExp$ '==' $RelExp \mid EqExp$ '!=' $RelExp$

- 逻辑表达式

  $LAndExp \rightarrow EqExp \mid LAndExp$ '&&' $EqExp$

  $LOrExp \rightarrow LAndExp \mid LOrExp$ '||' $LAndExp$

- 条件表达式

  $Cond \rightarrow LOrExp$

- 常量表达式

  $ConstExp \rightarrow AddExp$

5. 注释

$Comment \rightarrow$ '\\'$Comment\text{-}text$'\n' | '*'$Comment\text{-}text$'*\'

$Comment\text{-}text \rightarrow$ **[^\n]***

6. 函数

- 函数声明

  FunDecl → **function** *id* ([*ParameterList*])

  ParameterList → *Parameter* { , *Parameter* }

  Parameter → *Type id*

- 函数调用

  FunCall → *id* ([*ArgumentList*])

  ArgumentList → *expr* { , *expr* }

  expr → *Literal* | *id* | *FunCall* | ...

  Literal → **true** | **false** | *Number* | *String* | ...

二、 汇编编程

(一) ARM 汇编概述

ARM（Advanced RISC Machine）汇编语言是一种精简指令集计算机（RISC）架构的汇编语言，它包括一组简单而强大的指令集，可用于执行各种计算、控制和数据操作。ARM 汇编语言通常包括数据处理、分支、跳转、加载/存储等操作指令，以及条件执行的支持。

ARM 汇编语言常被用于实现嵌入式系统开发、编写底层驱动程序、进行性能优化、辅助逆向工程和漏洞研究等。在有些情况下，编写 ARM 汇编能实现实现更高级的控制和性能优化、为开发者提供更多的工具和技能、解决复杂的嵌入式和系统级编程问题。我们本学期实验的目标也是设计编译器将高级语言程序转换成 ARM 汇编。

在 ARM 汇编程序中，指令、伪指令、伪操作、寄存器名等既可以全部使用大写也可以全部使用小写，但不可以大小写混用。在该系统中，与 X86 类似预先定义了一些段名:

定义	含义
.text	代码段
.data	初始化数据段
.bss	未初始化数据段
.rodata	只读数据段

当然还可以使用 ‘.section .testsection @ 定义一个 testsetcion 段 ‘的方式来自定义段。

(二) SysY 源程序

1. Test 1

```
1  #include<stdio.h>
2  const float pai = 3.14159;
3
4  float getCircumference(int type, float r)
5  {
6      float circumference;
```

```
    if(type == 0) {
        circumference = 2 * pai * r;
    }
    else {
        circumference = 4 * r;
    }
    return circumference;
}

float getArea(int type, float r)
{
    float area;
    if(type == 0) {
        area = pai * r * r;
    }
    else {
        area = r * r;
    }
    return area;
}

int main()
{
    int type;
    printf("Input the type (0 for circle, others for square): ");
    scanf("%d", &type);
    float r;
    printf("Input a number: ");
    scanf("%f", &r);
    if(r < 0) {
        printf("Invalid number. Please enter a non-negative value.\n");
    }
    else {
        float circumference = getCircumference(type, r);
        float area = getArea(type, r);
        if(type == 0) {
            printf("Circumference of the square: %.4f\n", circumference);
            printf("Area of the circle: %.4f\n", area);
        }
        else{
            printf("Circumference of the square: %.4f\n", circumference);
            printf("Area of the circle: %.4f\n", area);
        }
    }
```

```
    return 0;
}
```

2. Test 2

```
#include <stdio.h>
#define MAX_COUNT 10

int calculateSum(int n);

int main() {
    int counter = 1;
    int result = 0;
    while (counter <= MAX_COUNT) {
        result += calculateSum(counter);
        counter++;
    }
    printf("Total Sum of the first %d numbers is: %d\n", MAX_COUNT, result);
    return 0;
}

int calculateSum(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += i;
    }
    return sum;
}
```


(三) ARM 汇编程序

1. Test 1

```
.data
.global pai         @ 声明全局标识符 "pai"
pai:
    .float 3.14159265  @ 初始化 "pai" 变量为 3.14159265

.text
    .global getCircumference  @ 声明全局函数 "getCircumference"
    .type getCircumference, %function  @ 定义 "getCircumference" 函数的元信息
getCircumference:
    str fp, [sp, #-4]!  @ 保存当前栈帧的栈指针到栈上，并更新栈指针
```

```
11      add fp, sp, #0      @ 设置新的栈帧指针
12      sub sp, sp, #20     @ 分配 20 字节的栈空间用于局部变量
13
14      str r0, [fp, #-16]    @ 将参数 r0 存储到栈帧上的位置
15
16      vstr.32 s0, [fp, #-20]  @ 将单精度浮点寄存器 s0 的值存储到栈帧上的位置
17
18      ldr r3, [fp, #-16]  @ 加载参数 r0 到 r3 寄存器
19      cmp r3, #0          @ 比较 r3 和 0
20      bne .NotEqualZeroLabel @ 如果不等于 0, 则跳转到 .NotEqualZeroLabel 标签
21
22      ldr r1, =pai
23      vldr.32 s15, [r1]     @ 加载单精度浮点常数到 s15 寄存器
24      vadd.f32 s15, s15, s15  @ 将 s15 寄存器的值加倍
25      vldr.32 s14, [fp, #-20]  @ 加载栈帧上的单精度浮点数到 s14 寄存器
26      vmul.f32 s15, s14, s15  @ 计算 s14 * s15 的结果并存储到 s15 寄存器
27      vstr.32 s15, [fp, #-8]  @ 将结果存储到栈帧上的位置
28      b .EndLabel               @ 跳转到 .EndLabel 标签
29
30  .NotEqualZeroLabel:
31      vldr.32 s15, [fp, #-20]  @ 加载栈帧上的单精度浮点数到 s15 寄存器
32      vmov.f32 s14, #4.0e+0     @ 将单精度浮点常数 4.0 存储到 s14 寄存器
33      vmul.f32 s15, s15, s14    @ 计算 s15 * s14 的结果并存储到 s15 寄存器
34      vstr.32 s15, [fp, #-8]    @ 将结果存储到栈帧上的位置
35
36  .EndLabel:
37      ldr r3, [fp, #-8]    @ 加载栈帧上的单精度浮点数到 r3 寄存器
38      vmov s15, r3          @ 将 r3 寄存器的值存储到单精度浮点寄存器 s15 中
39      vmov.f32 s0, s15      @ 复制 s15 寄存器的值到 s0 寄存器
40      add sp, fp, #0        @ 恢复栈指针
41      @ sp needed
42      ldr fp, [sp], #4      @ 恢复栈帧指针
43      bx lr                 @ 返回
44
45      .global getArea  @ 声明全局函数 "getArea"
46      .type getArea, %function  @ 定义 "getArea" 函数的元信息
47  getArea:
48      str fp, [sp, #-4]!  @ 保存当前栈帧的栈指针到栈上, 并更新栈指针
49      add fp, sp, #0      @ 设置新的栈帧指针
50      sub sp, sp, #20     @ 分配 20 字节的栈空间用于局部变量
51
52      str r0, [fp, #-16]    @ 将参数 r0 存储到栈帧上的位置
53
54      vstr.32 s0, [fp, #-20]  @ 将单精度浮点寄存器 s0 的值存储到栈帧上的位置
```

```
55
56      ldr r3, [fp, #-16]   @ 加载参数 r0 到 r3 寄存器
57      cmp r3, #0           @ 比较 r3 和 0
58      bne .NotEqualZeroLabel2 @ 如果不等于 0, 则跳转到 .NotEqualZeroLabel2 标签
59
60      ldr r1, =pai
61      vldr.32 s14, [r1]    @ 加载单精度浮点常数到 s14 寄存器
62      vldr.32 s15, [fp, #-20]   @ 加载栈帧上的单精度浮点数到 s15 寄存器
63      vmul.f32 s15, s14, s15   @ 计算 s14 * s15 的结果并存储到 s15 寄存器
64      vldr.32 s14, [fp, #-20]   @ 再次加载栈帧上的单精度浮点数到 s14 寄存器
65      vmul.f32 s15, s14, s15   @ 计算 s14 * s15 的结果并存储到 s15 寄存器
66      vstr.32 s15, [fp, #-8]   @ 将结果存储到栈帧上的位置
67      b .EndLabel2         @ 跳转到 .EndLabel2 标签
68
69  .NotEqualZeroLabel2:
70      vldr.32 s15, [fp, #-20]   @ 加载栈帧上的单精度浮点数到 s15 寄存器
71      vmul.f32 s15, s15, s15   @ 计算 s15 * s15 的结果并存储到 s15 寄存器
72      vstr.32 s15, [fp, #-8]    @ 将结果存储到栈帧上的位置
73
74  .EndLabel2:
75      ldr r3, [fp, #-8]    @ 加载栈帧上的单精度浮点数到 r3 寄存器
76      vmov s15, r3         @ 将 r3 寄存器的值存储到单精度浮点寄存器 s15 中
77      vmov.f32 s0, s15     @ 复制 s15 寄存器的值到 s0 寄存器
78      add sp, fp, #0       @ 恢复栈指针
79      @ sp needed
80      ldr fp, [sp], #4     @ 恢复栈帧指针
81      bx lr                @ 返回
82
83  .section .rodata
84  .InputTypePrompt:
85      .ascii "Input the type (0 for circle, others for square): \000"   @ 提示消息
86      .align 2
87  .FormatInteger:
88      .ascii "%d\000"   @ 格式化字符串
89      .align 2
90  .InputNumberPrompt:
91      .ascii "Input a number: \000"   @ 提示消息
92      .align 2
93  .FormatFloat:
94      .ascii "%f\000"   @ 格式化字符串
95      .align 2
96  .InvalidNumberMessage:
97      .ascii "Invalid number. Please enter a non-negative value.\000"   @ 提示消息
98      .align 2
```

```
99  .CircumferenceMessage:
100     .ascii "Circumference of the square: %.4f\012\000"  @ 提示消息
101     .align 2
102 .AreaMessage:
103     .ascii "Area of the circle: %.4f\012\000"  @ 提示消息
104
105     .text
106     .align 2
107     .global main  @ 声明全局函数 "main"
108     .type main, %function  @ 定义 "main" 函数的元信息
109 main:
110     @ 栈帧设置和局部变量分配
111     push {fp, lr}       @ 保存当前函数的栈帧指针和返回地址
112     add fp, sp, #4      @ 设置新的栈帧指针
113     sub sp, sp, #24     @ 分配 24 字节的栈空间用于局部变量
114
115     @ 读取输入类型
116     ldr r2, .bridge         @ 加载 .bridge 标签的地址到 r2 寄存器
117 .LPIC9:
118     add r3, pc, r3      @ 计算全局偏移地址
119     str r3, [fp, #-8]   @ 将 r3 寄存器的值存储到栈帧上的位置
120     mov r3, #0          @ 将常数 0 存储到 r3 寄存器
121     ldr r3, .bridge+4       @ 加载 .bridge+4 标签的地址到 r3 寄存器
122 .LPIC0:
123     add r3, pc, r3      @ 计算全局偏移地址
124     mov r0, r3          @ 将 r3 寄存器的值存储到 r0 寄存器（用于 printf 调用）
125     bl printf       @ 调用 printf 函数以打印消息
126     sub r3, fp, #24     @ 计算栈帧上局部变量的地址并存储到 r3 寄存器
127     mov r1, r3          @ 将 r3 寄存器的值存储到 r1 寄存器
128     ldr r3, .bridge+8       @ 加载 .bridge+8 标签的地址到 r3 寄存器
129 .LPIC1:
130     add r3, pc, r3      @ 计算全局偏移地址
131     mov r0, r3          @ 将 r3 寄存器的值存储到 r0 寄存器（用于 scanf 调用）
132     bl scanf  @ 调用 scanf 函数以读取输入
133     ldr r3, .bridge+12      @ 加载 .bridge+12 标签的地址到 r3 寄存器
134 .LPIC2:
135     add r3, pc, r3      @ 计算全局偏移地址
136     mov r0, r3          @ 将 r3 寄存器的值存储到 r0 寄存器（用于 printf 调用）
137     bl printf       @ 调用 printf 函数以打印消息
138     sub r3, fp, #20     @ 计算栈帧上局部变量的地址并存储到 r3 寄存器
139     mov r1, r3          @ 将 r3 寄存器的值存储到 r1 寄存器
140     ldr r3, .bridge+16      @ 加载 .bridge+16 标签的地址到 r3 寄存器
141 .LPIC3:
142     add r3, pc, r3      @ 计算全局偏移地址
```

```
143     mov r0, r3              @ 将 r3 寄存器的值存储到 r0 寄存器（用于 scanf 调用）
144     bl scanf    @ 调用 scanf 函数以读取输入
145
146     vldr.32 s15, [fp, #-20]   @ 加载栈帧上的单精度浮点数到 s15 寄存器
147     vcmpe.f32 s15, #0         @ 比较 s15 和 0
148     vmrs APSR_nzcv, FPSCR     @ 获取浮点标志寄存器的状态
149     bpl .getCircumferenceAndArea
150     @ 如果 s15 大于等于 0，则跳转到 .getCircumferenceAndArea 标签
151
152     ldr r3, .bridge+20      @ 加载 .bridge+20 标签的地址到 r3 寄存器
153 .LPIC4:
154     add r3, pc, r3      @ 计算全局偏移地址
155     mov r0, r3          @ 将 r3 寄存器的值存储到 r0 寄存器（用于 puts 调用）
156     bl puts         @ 调用 puts 函数以打印错误消息
157     b .ExitLabel                @ 跳转到 .ExitLabel 标签
158
159 .getCircumferenceAndArea:
160     ldr r3, [fp, #-24]    @ 加载栈帧上的整数到 r3 寄存器
161     vldr.32 s15, [fp, #-20]   @ 加载栈帧上的单精度浮点数到 s15 寄存器
162     vmov.f32 s0, s15    @ 复制 s15 寄存器的值到 s0 寄存器
163     mov r0, r3          @ 将 r3 寄存器的值存储到 r0 寄存器（用于 getCircumference 调用）
164     bl getCircumference   @ 调用 getCircumference 函数计算周长
165     vstr.32 s0, [fp, #-16]    @ 将结果存储到栈帧上的位置
166
167     ldr r3, [fp, #-24]    @ 加载栈帧上的整数到 r3 寄存器
168     vldr.32 s15, [fp, #-20]   @ 加载栈帧上的单精度浮点数到 s15 寄存器
169     vmov.f32 s0, s15    @ 复制 s15 寄存器的值到 s0 寄存器
170     mov r0, r3          @ 将 r3 寄存器的值存储到 r0 寄存器（用于 getArea 调用）
171     bl getArea      @ 调用 getArea 函数计算面积
172     vstr.32 s0, [fp, #-12]    @ 将结果存储到栈帧上的位置
173
174     ldr r3, [fp, #-24]    @ 加载栈帧上的整数到 r3 寄存器
175     cmp r3, #0          @ 比较 r3 和 0
176     bne .SquareLabel                @ 如果不等于 0，则跳转到 .SquareLabel 标签
177
178     vldr.32 s15, [fp, #-16]   @ 加载栈帧上的单精度浮点数到 s15 寄存器
179     vcvt.f64.f32 d7, s15    @ 将 s15 寄存器的单精度浮点数转换为双精度浮点数
180     vmov r2, r3, d7         @ 将双精度浮点数的高位存储到 r2 寄存器
181     ldr r1, .bridge+24          @ 加载 .bridge+24 标签的地址到 r1 寄存器
182 .LPIC5:
183     add r1, pc, r1          @ 计算全局偏移地址
184     mov r0, r1              @ 将 r1 寄存器的值存储到 r0 寄存器（用于 printf 调用）
185     bl printf           @ 调用 printf 函数以打印消息
186
```

```
187     vldr.32 s15, [fp, #-12]   @ 加载栈帧上的单精度浮点数到 s15 寄存器
188     vcvt.f64.f32 d7, s15    @ 将 s15 寄存器的单精度浮点数转换为双精度浮点数
189     vmov r2, r3, d7         @ 将双精度浮点数的高位存储到 r2 寄存器
190     ldr r1, .bridge+28          @ 加载 .bridge+28 标签的地址到 r1 寄存器
191 .LPIC6:
192     add r1, pc, r1          @ 计算全局偏移地址
193     mov r0, r1              @ 将 r1 寄存器的值存储到 r0 寄存器（用于 printf 调用）
194     bl printf           @ 调用 printf 函数以打印消息
195     b .ExitLabel                @ 跳转到 .ExitLabel 标签
196
197 .SquareLabel:
198     vldr.32 s15, [fp, #-16]   @ 加载栈帧上的单精度浮点数到 s15 寄存器
199     vcvt.f64.f32 d7, s15    @ 将 s15 寄存器的单精度浮点数转换为双精度浮点数
200     vmov r2, r3, d7         @ 将双精度浮点数的高位存储到 r2 寄存器
201     ldr r1, .bridge+32          @ 加载 .bridge+32 标签的地址到 r1 寄存器
202 .LPIC7:
203     add r1, pc, r1          @ 计算全局偏移地址
204     mov r0, r1              @ 将 r1 寄存器的值存储到 r0 寄存器（用于 printf 调用）
205     bl printf           @ 调用 printf 函数以打印消息
206
207     vldr.32 s15, [fp, #-12]   @ 加载栈帧上的单精度浮点数到 s15 寄存器
208     vcvt.f64.f32 d7, s15    @ 将 s15 寄存器的单精度浮点数转换为双精度浮点数
209     vmov r2, r3, d7         @ 将双精度浮点数的高位存储到 r2 寄存器
210     ldr r1, .bridge+36          @ 加载 .bridge+36 标签的地址到 r1 寄存器
211 .LPIC8:
212     add r1, pc, r1          @ 计算全局偏移地址
213     mov r0, r1              @ 将 r1 寄存器的值存储到 r0 寄存器（用于 printf 调用）
214     bl printf           @ 调用 printf 函数以打印消息
215
216 .ExitLabel:
217     mov r0, r3              @ 将 r3 寄存器的值存储到 r0 寄存器
218     sub sp, fp, #4          @ 恢复栈指针
219     @ sp needed
220     pop {fp, pc}            @ 弹出栈帧指针和返回地址
221
222 .bridge:
223     .word _GLOBAL_OFFSET_TABLE_-(.LPIC9+8)   @ 全局偏移表地址
224     .word .InputTypePrompt-(.LPIC0+8)   @ .InputTypePrompt 地址
225     .word .FormatInteger-(.LPIC1+8)   @ .FormatInteger 地址
226     .word .InputNumberPrompt-(.LPIC2+8)   @ .InputNumberPrompt 地址
227     .word .FormatFloat-(.LPIC3+8)   @ .FormatFloat 地址
228     .word .InvalidNumberMessage-(.LPIC4+8)   @ .InvalidNumberMessage 地址
229     .word .CircumferenceMessage-(.LPIC5+8)   @ .CircumferenceMessage 地址
230     .word .AreaMessage-(.LPIC6+8)   @ .AreaMessage 地址
```

```
231     .word .CircumferenceMessage-(.LPIC7+8)  @ .CircumferenceMessage 地址
232     .word .AreaMessage-(.LPIC8+8)  @ .AreaMessage 地址
```

2. Test 2

```
1   .global main          @ 定义全局标签 main，表示程序的入口点。
2
3   .section .data        @ 数据段，定义MAX_COUNT 及 format 标签
4   MAX_COUNT:
5       .word 10
6   format:
7       .asciz "Total Sum of the first %d numbers is: %d\n"
8
9   .section .bss         @ .bss 段定义两个四字节空间的变量，分别存储计数器的值和计算结果
10  counter:
11      .space 4
12  result:
13      .space 4
14
15  .text
16  main:
17  @ main 是程序入口点，
18  @ 初始化counter（计数器）和result（结果）变量
19  @ 将计数器counter 初始值设置为 1
20      ldr r0, =counter        @ 将 counter 变量的地址加载到 r0 寄存器中
21      mov r1, #1        @ 将值1存储在 r1 寄存器中，表示计数器的初始值。
22      str r1, [r0]          @ 将 r1 寄存器中的值存储到 counter 变量的地址
23  @ 将结果result 初始值设置为 0
24      mov r0, #0        @ 将值0存储在 r0 寄存器中，表示结果的初始值
25      ldr r4, =result       @ 将 result 变量的地址加载到 r4 寄存器中
26      str r0, [r4]          @ 将r0 寄存器中的值存储到 r4 寄存器中指定的地址
27      @ str r0, [result]
28
29  loop_start:
30      @ 循环开始，程序将在这里进入循环来计算总和。\
31
32   @ 将 counter 变量的值加载到 r1 寄存器中，以便程序可以在寄存器中操作该值。
33      ldr r0, =counter          @ 将counter 变量地址加载到 r0
34      ldr r1, [r0]
35      @ 将 counter 变量的值加载到了 r1 寄存器中，
36      @ 以便后续可以使用 r1 寄存器中的值进行操作。
37
38      @ 比较计数器和MAX_COUNT（循环结束条件判断）
```

```
39      ldr r2, =MAX_COUNT
40      ldr r2, [r2]          @ 加载 MAX_COUNT 的值到 r2 寄存器中
41      cmp r1, r2
42      bgt loop_end          @ 如果 r1 大于 r2, 则跳转到 loop_end, 结束循环。
43
44      @ 调用calculateSum 函数
45      bl calculateSum
46
47      @ 将返回值添加到结果中
48      @ 将计数器的值加到结果中
49      ldr r4, =result       @ 将result 标签的地址加载到 r4 寄存器中
50      ldr r2, [r4]
51  I   add r2, r2, r0
52  I   str r2, [r4]
53      @ 增加计数器
54      ldr r0, =counter
55      ldr r1, [r0]
56      add r1, r1, #1
57      str r1, [r0]
58
59      @ 比较计数器和MAX_COUNT
60      cmp r1, r3
61      ble loop_start
62      @ 继续循环
63      @   b loop_start
64
65  loop_end:
66      @ 准备参数并调用printf 函数来打印结果
67      ldr r0, =format
68      @ 将 format 变量的内存地址存储在 r0 寄存器中,
69      @ 以便后续可以将该地址传递
70      printf 函数。
71      ldr r3, =MAX_COUNT
72      ldr r1, [r3]          @ 将 MAX_COUNT 变量的值存储在 r1 寄存器中
73      ldr r4, =result       @ 将 result 变量的内存地址存储在 r4 寄存器中
74      ldr r2, [r4]          @ 加载result 变量的值到 r2 寄存器中
75      bl printf    @ printf 函数, 向其传递参数 r0、r1 和 r2
76
77      @ 退出程序
78      mov r0, #0        @ 将零存储在 r0 寄存器中, 用于表示程序的返回状态正常
79      bx lr
80
81  calculateSum:
82      @ 计算和
```

```
83      mov r2, #0          @ 初始化总和sum = 0
84      mov r3, #1          @ 初始化循环计数器i = 1
85      @ldr r4, =MAX_COUNT
86      ldr r4, [r0]        @ 加载 n 的值到 r4 寄存器中
87  loop_sum:
88      cmp r3, r4          @ 比较i 和 MAX_COUNT
89      bgt loop_end_sum
90      @ i 大于 MAX_COUNT, 则跳转到 loop_end_sum 标签, 循环结束
91      add r2, r2, r3      @ sum += i
92      add r3, r3, #1      @ i++
93      b loop_sum
94
95  loop_end_sum:
96      mov r0, r2
97      @ 将 r2 寄存器中的值 (sum) 复制到 r0 寄存器中, 用于返回结果
98      bx lr           @ 用 bx 指令跳转回到函数调用点
```

(四) 代码测试

1. Test 1

为了方便进行调试，我编写了如下 Makefile 文件：

```
1  .PHONY: test1, clean
2  test1:
3      arm-linux-gnueabihf-gcc Test1.s -o Test1.out -static -march=armv7-a -mfpu=vfpv3
4      qemu-arm ./Test1.out
5  clean:
6      rm -fr *.out
```

运行结果如图1所示

```
yuzhao-peng@yuzhao-peng:~/Compiler_Sys/Lab0-2/final$ make test1
arm-linux-gnueabihf-gcc Test1.s -o Test1.out -static -march=armv7-a -mfpu=vfpv3
qemu-arm ./Test1.out
Input the type (0 for circle, others for square): 0
Input a number: 1
Circumference of the square: 6.2832
Area of the circle: 3.1416
yuzhao-peng@yuzhao-peng:~/Compiler_Sys/Lab0-2/final$ make test1
arm-linux-gnueabihf-gcc Test1.s -o Test1.out -static -march=armv7-a -mfpu=vfpv3
qemu-arm ./Test1.out
Input the type (0 for circle, others for square): 1
Input a number: 1
Circumference of the square: 4.0000
Area of the circle: 1.0000
```

图 1: Test1 运行结果

可以看见我们的程序正确实现了输入、运算与输出的功能。现在我们来分析一下程序主要包括了哪些 SysY 语言特性（如下）：

- 数据类型：整数（int）、浮点数（float）
- 变量声明（type、r）、常量声明（pai），常量的初始化
- 语句：表达式语句，if、return
- 表达式：算术运算、关系运算、条件表达式
- 输入输出
- 函数（getCircumference、getArea）：函数声明、函数调用

2. Test 2

运行结果如图2所示

```
hanj@hanj-virtual-machine:~/Lab2$ arm-linux-gnueabihf-gcc TOtalSUM16.S -o TOtalS
UM16.out
hanj@hanj-virtual-machine:~/Lab2$ qemu-arm -L /usr/arm-linux-gnueabihf ./TOtalSU
M16.out
Total Sum of the first 10 numbers is: 220
```

图 2: Test2 运行结果

程序输出正确，本程序实现的 SysY 语言特性如下：

- 数据类型：整数（int）
- 变量声明（counter、result）、常量声明（MAX_COUNT），常量的初始化、变量的初始化
- 语句：表达式语句，while、for、if、return

- 表达式：算术运算、关系运算、条件表达式
- 输入输出
- 函数（calculateSum）：函数声明、函数调用

参考文献

[1] CSC-Compiler. SysY 语言定义, 2023.