



南开大学
Nankai University

南 开 大 学

网 络 空 间 安 全 学 院

编译系统原理实验报告

预备工作 2 定义编译器 & ARM 汇编编程

张丛 刘国民

年级：2021 级

专业：信息安全

指导教师：王刚

2023 年 10 月 10 日

摘要

本次实验我们参考 miniSysY 的全部文法，结合我们想要实现的 SysY 语言特性设计了上下无关文法。通过四元组对 CFG 进行了描述，同时为了熟悉目标代码——ARM 汇编语言，我们编写了斐波那契数列和阶乘程序的汇编代码，程序能够正确执行。其中 CFG 描述部分由刘国民和张丛同学共同完成，ARM 汇编部分刘国民同学负责斐波那契数列程序的编写，张丛同学负责阶乘程序的编写。

关键字：ARM 汇编；上下无关文法

目录

一、 定义编译器

(一) 编译器支持的 SysY 语言特性

基础 track:

- 数据类型: int
- 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

竞赛 track:

- 数组
- 变量、常量作用域——在语句块中包含变量、常量声明, break、continue 语句
- 函数
- 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

设计 CFG 时我们将以上语言特性均考虑在内, 在后续实验中会尝试实现所有语言特性

(二) CFG 描述 SysY 语言特性

我们利用上下无关文法对编译器所支持的 SysY 语言特性子集进行形式化定义, CFG 包括终结符集合 V_T , 非终结符集合 V_N , 开始符号 S 和产生式集合 P 四个部分。

1. 终结符集合 V_T

- 标识符

$$id \rightarrow id_nondigit$$

$$| id\ id_nondigit$$

$$| id\ digit$$

$$id_nondigit \rightarrow _ | a | b | c | d | e | f | g | h | i | j | k | l$$

$$| m | n | o | p | q | r | s | t | u | v | w | x | y$$

$$| z | A | B | C | D | E | F | G | H | I | J | K | L$$

$$| M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z$$

$$digit \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$$

- 数值常量

$$\begin{aligned} integer_const \rightarrow & decimal_const \\ & | octal_const \\ & | hex_const \end{aligned}$$

$$\begin{aligned} decimal_const \rightarrow & nonzero_digit \\ & | decimal_const\ digit \end{aligned}$$

$$octal_const \rightarrow 0 | octal_const\ octal_const$$

$$\begin{aligned} hex_const \rightarrow & hex_prefix\ hex_digit \\ & | hex_const\ hex_digit \end{aligned}$$

$$\begin{aligned} hex_prefix \rightarrow & '0x' | '0X' \\ nonzero_digit \rightarrow & 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \\ octal_digit \rightarrow & 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 \end{aligned}$$

$$\begin{aligned} hex_digit \rightarrow & 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 \\ & | 9 | a | b | c | d | e | f \\ & | A | B | C | D | E | F \end{aligned}$$

- 运算符

$$\{ +, -, *, /, \%, =, !, >, <, >=, <=, ==, !=, \&\&, || \}$$

- 关键字

$$\{ \text{void, int, if, else, while, break, const, return} \}$$

- 基本符号

$$\{ ;, [,], (,), ,, /*, */, // \}$$

2. 非终结符集合 V_N

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

3. 开始符号 S

开始符号: CompUnit

4. 产生式集合 P

- $\text{CompUnit} \rightarrow [\text{CompUnit}] (\text{Decl} \mid \text{FuncDef})$
- $\text{Decl} \rightarrow \text{ConstDecl} \mid \text{VarDecl}$
- $\text{ConstDecl} \rightarrow \text{'const' BType ConstDef ',' ConstDef ';'}$
- $\text{BType} \rightarrow \text{'int'}$
- $\text{ConstDef} \rightarrow \text{Ident '[' ConstExp ']' '=' ConstInitVal}$
- $\text{ConstInitVal} \rightarrow \text{ConstExp} \mid \text{'[ConstInitVal ',' ConstInitVal]'}$
- $\text{VarDecl} \rightarrow \text{BType VarDef ',' VarDef ';'}$
- $\text{VarDef} \rightarrow \text{Ident '[' ConstExp ']' } \mid \text{Ident '[' ConstExp ']' '=' InitVal}$
- $\text{InitVal} \rightarrow \text{Exp} \mid \text{'[InitVal ',' InitVal]'}$
- $\text{FuncDef} \rightarrow \text{FuncType Ident '(' [FuncFParams] ')' Block}$
- $\text{FuncType} \rightarrow \text{'void' } \mid \text{'int'}$
- $\text{FuncFParams} \rightarrow \text{FuncFParam ',' FuncFParam}$
- $\text{FuncFParam} \rightarrow \text{BType Ident '[' ']' '[' Exp ']' }$
- $\text{Block} \rightarrow \text{'[BlockItem]'}$
- $\text{BlockItem} \rightarrow \text{Decl} \mid \text{Stmt}$

- Stmt \rightarrow LVal '=' Exp ';' | [Exp] ';' | Block | 'if' '(' Cond ')' Stmt ['else' Stmt] | 'while' '(' Cond ')' Stmt | 'break' ';' | 'continue' ';' | 'return' [Exp] ';'
 - Exp \rightarrow AddExp
 - Cond \rightarrow LOrExp
 - LVal \rightarrow Ident '[' Exp ']'
 - PrimaryExp \rightarrow '(' Exp ')' | LVal | Number
 - UnaryExp \rightarrow PrimaryExp | Ident '(' [FuncRParams] ')' | UnaryOp UnaryExp
 - UnaryOp \rightarrow '+' | '-' | '!' // 注：保证'!' 仅出现在 Cond 中
 - FuncRParams \rightarrow Exp ',' Exp
 - MulExp \rightarrow UnaryExp | MulExp ('*' | '/' | '%') UnaryExp
 - AddExp \rightarrow MulExp | AddExp ('+' | '-') MulExp
 - RelExp \rightarrow AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
 - EqExp \rightarrow RelExp | EqExp ('==' | '!=') RelExp
 - LAndExp \rightarrow EqExp | LAndExp '&&' EqExp
 - LOrExp \rightarrow LAndExp | LOrExp '||' LAndExp
 - ConstExp \rightarrow AddExp // 在语义上额外约束这里的 AddExp 必须是一个可以在编译期求出值的常量

二、 ARM 汇编编程

ARM 汇编编程部分我们以斐波那契数列和阶乘求解为例，编写了 SysY 程序和对应的 ARM 汇编程序。熟悉了源语言和目标语言的语言特性，为后续实现编译器打好基础。

(一) 斐波那契数列

SysY 程序与 C 程序类似，具体代码如下：

1. 斐波那契数列 SysY 程序

斐波那契数列 SysY 程序

```

1 #include<stdio.h>
2 int main()
3 {
4     int a, b, i, t, n;
5     a = 0;
6     b = 1;
7     i = 1;
8     printf("Please enter the number of items in the Fibonacci sequence:");

```

```
9 scanf("%d",&n);
10 printf("The result is:\n%d\n",b);
11 while (i < n){
12     t = b;
13     b = a + b;
14     printf("%d\n",b);
15     a = t;
16     i = i + 1;
17 }
18 return 0;
19 }
```

由于 SysY 语言是 C 语言的子集, 故源代码采用.c 格式保存。通过以下命令编译链接后, 程序可以正确执行。

```
1 gcc fib.c -o fib
2 .\fib
```

2. 斐波那契 ARM 汇编程序

汇编代码如下所示, 其中注释以 @ 开头

斐波那契数列 ARM 汇编代码

```
1 .arch armv5t           @表示使用 ARMv5t 指令集
2
3
4 @ 数据段
5 .comm n, 4             @ 未初始化变量
6 .data
7     a: .word 0          @ arm架构下word表示32bits,与x86(16 bits)相区别
8     b: .word 1
9     i: .word 1
10    t: .word 0
11
12
13 @ 只读数据段
14 .rodata:
15     .align 2
16 info:
17     .asciz "Please enter the number of items in the Fibonacci sequence:"
18     .align 2
19 input:
20     .asciz "%d"
21     .align 2
22 output1:
23     .asciz "The result is:\n%d\n"
24     .align 2
25 output2:
26     .asciz "%d\n"
```

```

27     .align 2
28
29
30 @ 代码段
31 .text
32     .align 2
33     .global main
34 main:
35     push    {fp, lr}                @ 从左到右压入栈中，作用是保存返回地址
        和栈基地址
36     add     fp, sp, #4              @ 开辟函数栈帧
37     ldr     r0, _bridge+20          @ 传入参数
38     bl      printf                 @ 调用输出函数
39
40
41     ldr     r1, _bridge              @ r1=&n
42     ldr     r0, _bridge+24          @ r0=input
43     bl      __isoc99_scanf          @ scanf("%d",&n)
44
45     ldr     r0, _bridge+4
46     ldr     r1, [r0]                @ r1=b
47     ldr     r0, _bridge+28          @ r0=output1
48     bl      printf
49
50 LOOP:
51     ldr     r0, _bridge+8
52     ldr     r1, [r0]                @ r1=i
53
54     ldr     r0, _bridge
55     ldr     r2, [r0]                @ r2=n
56
57     cmp     r1, r2
58     bge     L1                      @ 退出循环
59
60     ldr     r0, _bridge+4
61     ldr     r3, [r0]                @ r3=b
62
63     ldr     r0, _bridge+32
64     str     r3, [r0]                @ t=r3
65
66     ldr     r0, _bridge+12
67     ldr     r4, [r0]                @ r4=a
68     add     r3, r3, r4              @ r3=r3+r4
69     ldr     r0, _bridge+4
70     str     r3, [r0]                @ b=r3
71
72     ldr     r1, [r0]                @ r1=b
73     ldr     r0, _bridge+16          @ r0=output2

```



```
74     bl    printf
75
76     ldr    r0, _bridge+32
77     ldr    r1, [r0]                @ r1=t
78
79     ldr    r0, _bridge+12
80     str    r1, [r0]                @ a=r1
81
82     ldr    r0, _bridge+8
83     ldr    r2, [r0]                @ r2=i
84     add    r2, r2, #1              @ r2=r2+1
85     str    r2, [r0]                @ i=r2
86     b      LOOP
87
88
89 L1:
90     mov    r0, #0
91     pop    {fp, pc}                @ return 0
92
93 _bridge:
94     .word  n
95     .word  b
96     .word  i
97     .word  a
98     .word  output2
99     .word  info
100    .word  input
101    .word  output1
102    .word  t
103
104    .section .note.GNU-stack,"",%progbits
```

编写完汇编代码后，通过交叉编译将汇编代码编译链接成可执行文件：

```
1 arm-linux-gnueabihf-gcc -c fib.S -o fib.o
2 arm-linux-gnueabihf-gcc fib.o -o fib
```

打开文件后进行测试，程序成功执行且结果正确，如下图所示：

```
liu1114@liu1114-virtual-machine:~$ arm-linux-gnueabi-gcc -c fib.S -o fib.o
liu1114@liu1114-virtual-machine:~$ arm-linux-gnueabi-gcc fib.o -o fib
liu1114@liu1114-virtual-machine:~$ ./fib
Please enter the number of items in the Fibonacci sequence:12
The result is:
1
1
2
3
5
8
13
21
34
55
89
144
```

图 1: fib.S

(二) 阶乘

1. 阶乘 SysY 程序

sysY 语言与 C 语言类似。在阶乘的 sysY 程序中，尽可能地包含了支持的语言特性，设置有全局变量、分支、循环、函数调用等等。代码如下：

```
1 #include<stdio.h>
2
3 int n=0; //全局变量
4
5 int factorial(int n) {
6     int re=1;
7
8     if(n>=1) //分支
9     {
10         while(n) //循环
11         {
12             re*=n;
13             n--;
14         }
15     }
16     return re;
17 }
18
19 int main() {
20     scanf("%d",&n);
21     int result = factorial(n); //函数调用
22     printf("%d\n",result);
```

```

23     return 0;
24 }

```

2. 阶乘 ARM 汇编程序

在 Ubuntu 中编写写好 factorial.S 文件后, 使用命令:

```
1 arm-linux-gnueabi-gcc factorial.S -o fac
```

得到可执行程序 fac。

然后使用命令:

```
1 qemu-arm -L /usr/arm-linux-gnueabi/ ./fac
```

运行可执行文件可进行阶乘求解。如图:

```

root@LAPTOP-Q1FT8048:/mnt/c/Users/zc/test2# ls
factorial.S
root@LAPTOP-Q1FT8048:/mnt/c/Users/zc/test2# arm-linux-gnueabi-gcc factorial.S -o fac
root@LAPTOP-Q1FT8048:/mnt/c/Users/zc/test2# qemu-arm -L /usr/arm-linux-gnueabi/ ./fac
7
result is 5040
root@LAPTOP-Q1FT8048:/mnt/c/Users/zc/test2# ls
fac factorial.S
root@LAPTOP-Q1FT8048:/mnt/c/Users/zc/test2# |

```

图 2: fact.S

分别验证 0! 和 12!, 结果无误:

```

root@LAPTOP-Q1FT8048:/mnt/c/Users/zc/test2# qemu-arm -L /usr/arm-linux-gnueabi/ ./fac
0

```

图 3: fac0

```

root@LAPTOP-Q1FT8048:/mnt/c/Users/zc/test2# qemu-arm -L /usr/arm-linux-gnueabi/ ./fac
12
result is 479001600

```

图 4: fac12

arm 汇编代码如下:

```

1     .arch armv7-a           @架构
2     .comm a,4               @全局变量a, 占4字节
3     .text
4     .align 2
5     .section .rodata
6     .align 2
7     __str0:                  @字符串str0和str1
8     .ascii "%d\0"
9     .align 2
10    __str1:
11    .ascii "result is %d\n"
12    .text

```

```

13     .align 2
14
15     .global fac
16 fac:                                @ 函数 int factorial(int n)
17     str fp,[sp,#-4]!                 @表示将寄存器 fp 压入栈中, 同时更新 sp 指针
18     mov fp,sp                       @将 sp 指针设置为当前栈指针
19     sub sp,sp,#12                    @开辟栈空间
20     str r0,[fp,#-12]                 @ r0 = n
21     mov r8,#1                        @保存结果
22     cmp r0,#1
23     blt .end_fac
24
25     .loop:                           @循环
26         mul r8,r0
27         sub r0,#1
28         cmp r0,#0
29         bne .loop
30         b .end_fac
31
32     .end_fac:
33         add sp,fp,#0                 @恢复栈空间, 恢复寄存器
34         ldr fp,[sp],#4
35         bx lr                        @返回
36
37     .global main
38 main:
39     push {fp,lr}
40     add fp,sp,#4
41     ldr r1,_bridge                   @*r1=n
42     ldr r0,_bridge+4                 @*r0="%d\n"
43     bl __isoc99_scanf
44     ldr r3,_bridge                   @r3=&n
45     ldr r0,[r3]                      @r0=n
46     bl fac
47     mov r1,r8
48     ldr r0,_bridge+8                 @"result is %d\n"
49     bl printf
50     mov r0,#0
51     pop {fp,pc}
52
53     _bridge:
54         .word a
55         .word __str0
56         .word __str1
57
58     .section .note.GUN-stack,"",%progbits    @避免运行时出现的一些问题

```

参考文献