南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

编译系统原理实验报告

定义你的编译器、汇编编程 & 熟悉辅助工具

许积君 2111954 信息安全-法学

哈博儒 2111453 信息安全-法学

年级：2021 级

指导教师：李忠伟

2023 年 10 月 10 日

摘要

本次实验旨在对即将要编写的 SysY 语言进行形式化定义，从而为词法分析器的构建奠定坚实的基础。同时，为了更深入地理解 SysY 语言的程序结构，我们采用 ARM 汇编语言对 SysY 语言的代码进行了全面的重写和优化，以实现更高效的执行。这一过程将有助于我们更好地理解 SysY 语言的设计理念，为进一步的开发工作提供了宝贵的经验和见解。

关键字：SysY 语言；形式化定义；ARM 汇编语言

目录

一、 形式化定义 **1**

（一） 变量声明 1

（二） 表达式 1

（三） 循环、分支 2

（四） 注释 2

（五） 函数 2

二、 等价 ARM 汇编程序 **2**

（一） 阶乘 2

1. SysY 语言实现阶乘: 2
2. ARM 汇编语言实现阶乘: 3

（二） GCD 5

1. SysY 语言实现 gcd 5
2. ARM 汇编语言实现 gcd 6

（三） 异或加密 8

1. SysY 语言实现异或加密 8
2. ARM 汇编语言实现异或加密 9

三、 分工 **11**

一、 形式化定义

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 次或多次的项;

(一) 变量声明

```
基本类型:    Bytpe->'int' | 'float'
常量声明:    con_declaration ->'const' BType con_difine { ','con_define
   } ';'
常量定义:    con_define ->identifier {'[' const_exp ']'} '='con_init
常量初始化表:  con_init -> const_exp
                        |'{' [const_init { ',' con_int }] '}'

变量声明:    var_declaration -> Btype var_define {',' var_define}';'
变量定义:    var_define->identifier {'[' const_exp ']'}
                       |identifier {'[' const_exp ']'} '=' con_init
标识符:      identifier -> (letter|_) {letter|digit|_}
字母:        letter -> 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z'
数字符:       digit -> 0 | 1 | 2 | ... | 9
数字:         Number -> decimal-const | octal-const | hexadecimal-const
10进制:        decimal-const -> nonzero-digit | decimal-const digit
8进制:         octal-const -> '0' | octal-const octal-digit
16进制:        hexadecimal-const  -> hexadecimal-prefix hexadecimal-digit
                                   | hexadecimal-const hexadecimal-digit
            hexadecimal-prefix -> '0x' | '0X'
            nonzero-digit      -> '1' | '2' | ... | '9'
            octal-digit        -> '0' | '1' | ... | '7'
            digit              -> '0' | nonzero-digit
            hexadecimal-digit  -> '0' | '1' | ... | '9'
                                  | 'a' | 'b' | 'c' | 'd' | 'e' | 'f'
                                  | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
```

(二) 表达式

```
常量表达式:     const_exp->add_exp   //在语义上额外约束这里的 add_exp
   必须是一个可以在编译期求出值的常量
表达式:         expression ->add_exp
加法表达式:    add_exp ->mul_exp | add_exp ('+' | '-') mul_exp
乘除取模表达式:  mul_exp -> unary_exp
                          |mul_exp ('*' | '/' | '%') unary_exp
一元表达式:     unary_exp -> primary_exp
                          | identifier '(' [FuncRParams] ')'
                          | unary_op unary_exp
基础表达式:     primary_exp->'(' expression ')'       //子表达式
                          | left_val                  //左值表达式
                          | Number                    //字面值
左值表达式:     left_val->identifier{'[' expression ']'}  //变量名or数组
```

```
单目运算符:     unary_op->'+' | '-' | '!'

条件表达式:     condtion_exp->or_exp
逻辑或表达式:   or_exp->and_exp | or_exp '||' and_exp
逻辑与表达式:   and_exp->Eq_exp | and_exp '&&' Eq_exp
相等性表达式:   Eq_exp -> rel_exp| Eq_exp ('==' | '!=') rel_exp
关系表达式:     rel_exp->add_exp | rel_exp ('<' | '>' | '<=' | '>=')
    add_exp
```

(三) 循环、分支

```
语句: statement -> identifier '=' expression ';'
                |  'if' '(' condition ')' statement {'else if' statement
                    } ('else' statement)?
                | 'while' '(' condition ')' statement
                |'break' ';'
                |'continue' ';'
                |'return' expression ';'
condition->expression
```

(四) 注释

```
Comment -> SingleLineComment | MultiLineComment
SingleLineComment -> '//' [^'\n']* '\n'
MultiLineComment -> '/*' [^'*']* ('*' [^'/']* [^'*']*)* '*' '/'
```

(五) 函数

```
函数定义:    FuncDef->FuncType Ident'(' [FuncFParams] ')'Block
函数类型:    FuncType->Bytype
形参列表:    FuncFParams->FuncFParam
                        |FuncFParams','FuncFParam
形参定义:    FuncFParam->type identifier {'[' [const_exp] ']'}
```

二、 等价 ARM 汇编程序

(一) 阶乘

1. SysY 语言实现阶乘:

```
1 func int factorial(int n) {
2     if (n <= 1) {
3         return 1;
4     } else {
5         return n * factorial(n - 1);
6     }
```

```
}

func void main() {
    int result;
    int n;
    n = 5;  // 计算5的阶乘, SysY没有io, 自行设置
    result = factorial(n);
    print(result);
}
```

2. ARM 汇编语言实现阶乘：

arm 汇编-函数实现：

```
.data
input: .asciz "Input a number: "
format:   .asciz "%d"
output: .asciz "The factorial of %d is %d\n"

.text
factorial:
    str lr, [sp,#-4]!
    str r0, [sp,#-4]!

    cmp r0, #0
    bne L1
    mov r0, #1
    b end

L1:

    sub r0, r0, #1
    bl factorial
    ldr r1, [sp]
    mul r0, r1

end:
    add sp, sp, #+4
    ldr lr, [sp], #+4
    bx lr

.global main
main:
    str lr, [sp,#-4]!
    sub sp, sp, #4


    ldr r0, address_of_input  /*输出提示输入语句*/
    bl printf
```

```
36
37
38      ldr r0, address_of_format
39      mov r1, sp
40      bl scanf                        /*读数据*/
41
42      ldr r0, [sp]
43      bl factorial                      /*调用*/
44      mov r2, r0
45
46      ldr r1, [sp]
47      ldr r0, address_of_output     /*输出结果*/
48      bl printf
49
50      add sp, sp, #+4
51      ldr lr, [sp], #+4
52      bx lr
53
54  address_of_input: .word input
55  address_of_output: .word output
56  address_of_format: .word format
```

```
ubuntu@ubuntu-virtual-machine:~/yacc/lab2 定义词法分析/ARM汇编程序/阶乘$ arm-lin
ux-gnueabi-as -o factorial.o factorial.s
ubuntu@ubuntu-virtual-machine:~/yacc/lab2 定义词法分析/ARM汇编程序/阶乘$ arm-lin
ux-gnueabi-gcc -o factorial factorial.o -lm -lc -static
ubuntu@ubuntu-virtual-machine:~/yacc/lab2 定义词法分析/ARM汇编程序/阶乘$ qemu-ar
m -L /usr/arm-linux-gnueabi/ ./factorial
Input a number: 5
The factorial of 5 is 120
```

图 1: 阶乘 _ 函数汇编器和验证结果

arm 汇编-非函数实现：

```
1   .data
2   input: .asciz "Input a number: "
3   format: .asciz "%d"
4   output: .asciz "The factorial of %d is %d\n"
5
6   .text
7   .global main
8
9   main:
10      str lr, [sp,#-4]!
11      sub sp, sp, #4
12
13      ldr r0, address_of_input  /* 输出提示输入语句 */
14      bl printf
15
16      ldr r0, address_of_format
17      mov r1, sp
```

```
    bl scanf                    /* 读数据 */

    ldr r0, [sp]
    cmp r0, #0
    beq end


    mov r3, #1
    mov r2, #1

loop:
    mov r4, r3
    mul r3, r4, r2
    add r2, r2, #1
    cmp r2, r0
    ble loop

    ldr r1, [sp]
    mov r2, r3
    ldr r0, address_of_output
    bl printf

end:
    add sp, sp, #4
    ldr lr, [sp], #4
    bx lr

address_of_input: .word input
address_of_output: .word output
address_of_format: .word format
```

```
ubuntu@ubuntu-virtual-machine:~/yacc/lab2 定义词法分析/ARM汇编程序/阶乘$ arm-lin
ux-gnueabi-as -o factorial.o factorial.s
ubuntu@ubuntu-virtual-machine:~/yacc/lab2 定义词法分析/ARM汇编程序/阶乘$ arm-lin
ux-gnueabi-gcc -o factorial factorial.o -lm -lc -static
ubuntu@ubuntu-virtual-machine:~/yacc/lab2 定义词法分析/ARM汇编程序/阶乘$ qemu-ar
m -L /usr/arm-linux-gnueabi/ ./factorial
Input a number: 4
The factorial of 4 is 24
```

图 2: 阶乘 _ 无函数汇编器和验证结果

(二) GCD

1. SysY 语言实现 gcd

```
    int result;
int a;
int b;

void gcd() {
```

```
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    result = a;
}

int main() {
    a = readInt();
    b = readInt();

    gcd();

    print(result);

    return 0;
}
```

2. ARM 汇编语言实现 gcd

```
.data
input_prompt1:    .asciz "Enter the first number: "
input_prompt2:    .asciz "Enter the second number: "
output_format:    .asciz "The GCD of %d and %d is %d\n"
format:           .asciz "%d"
input_buffer1:    .word 0
input_buffer2:    .word 0
.text
.global main
.global gcd

main:
    str lr, [sp, #-4]!
    sub sp, sp, #4

    @ 输入第一个数字
    ldr r0, =input_prompt1
    bl printf

    ldr r0, =format
    ldr r1, =input_buffer1
    bl scanf

    @ 输入第二个数字
    ldr r0, =input_prompt2
    bl printf

```

```
28      ldr r0, =format
29      ldr r1, =input_buffer2
30      bl scanf
31
32      @ 调用 GCD 函数计算最大公约数
33      ldr r0, =input_buffer1
34      ldr r1, =input_buffer2
35      bl gcd
36      mov r3, r1
37      @ 输出结果
38      ldr r0, =output_format
39      ldr r1, =input_buffer1
40      ldr r1, [r1]
41      ldr r2, =input_buffer2
42      ldr r2, [r2]
43      bl printf
44
45      @ 退出程序
46      add sp, sp, #4
47      ldr lr, [sp], #4
48      bx lr
49
50  gcd:
51      str lr, [sp,#-4]!
52      sub sp, sp, #4
53
54      ldr r0,[r0]
55      ldr r1,[r1]
56      cmp r0, r1
57      movls r2, r1
58      movls r1, r0
59      movls r0, r2
60
61  loop:
62      sdiv r2, r0, r1
63      mul r3, r2, r1
64      sub r3, r0, r3
65      cmp r3, #0
66      beq end
67      mov r0, r1
68      mov r1, r3
69      b loop
70
71  end:
72      @ 返回
73      add sp, sp, #4
74      ldr lr, [sp], #4
75      bx lr
```

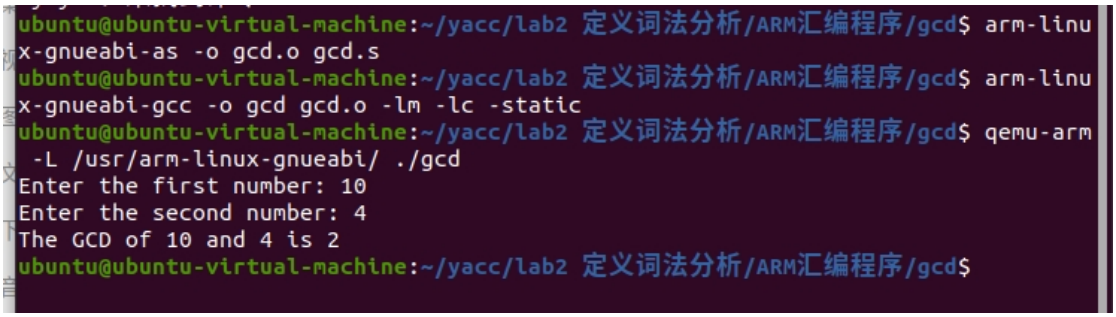

图 3: gcd 汇编器和验证结果

(三) 异或加密

1. SysY 语言实现异或加密

```
char* encryptDecrypt(char *message, char key) {
    // 获取消息的长度
    int len = 0;
    while (message[len] != '\0') {
        len++;
    }

    // 对消息进行异或加密或解密
    for (int i = 0; i < len; i++) {
        message[i] = message[i] ^ key;
    }

    return message;
}

int main() {
    char message[100];
    char key;

    printf("Enter message: ");
    scanf("%99[^\n]", message); // 读取包含空格的整行输入
    getchar(); // 清除输入缓冲区中的换行符

    printf("Enter encryption key: ");
    scanf("%c", &key);

    char* encryptedMessage = encryptDecrypt(message, key);
    printf("Encrypted message: %s\n", encryptedMessage);

    return 0;
}
```

2. ARM 汇编语言实现异或加密

```
    .comm key, 1


.section .bss

message:
    .space 100


.section .rodata
message_prompt:
    .asciz "Enter message: "
key_prompt:
    .asciz "Enter encryption key: "
format_string:
    .asciz "%99[^\n]"
key_format:
    .asciz "%d"
result_message:
    .asciz "Encrypted message: %s\n"

.section .text
.global encryptDecrypt
encryptDecrypt:
    @ 寄存器使用规则
    @ r0: 消息指针
    @ r1: 密钥
    @ r2: 消息长度
    @ r3: 临时变量
    @ r4: 循环计数器

    push {r4, lr}           @ 保存 r4 和 lr 寄存器

    mov r4, #0              @ 初始化循环计数器
loop:
    ldrb r3, [r0, r4]       @ 读取消息中的一个字节
    cmp r3, #0              @ 比较读取的字符和 0
    beq done                @ 如果相等, 跳转到 done
    add r4, r4, #1          @ 增加循环计数器
    b loop                  @ 无条件跳转到 loop

done:
    mov r2, r4              @ 保存消息长度到 r2
    mov r4, #0              @ 初始化循环计数器
loop_encrypt:
    cmp r4, r2          @ 比较循环计数器和消息长度
```

```
47      bge loop_encrypt_done     @ 如果仍然小于消息长度，则继续循环
48
49      ldrb r3, [r0, r4]  @ 读取消息中的一个字节
50      eor  r3, r1        @ 使用异或运算对字节进行加解密
51      strb r3, [r0, r4]  @ 将加解密后的字节写回消息
52      add r4, r4, #1     @ 增加循环计数器
53      b loop_encrypt
54  loop_encrypt_done:
55      pop {r4, lr}     @ 恢复 r4 和 lr 寄存器
56      bx lr            @ 返回
57
58
59  .global main
60  main:
61      push {r4, lr}   @ 保存 r4 和 lr 寄存器
62
63      ldr r0, addr_message_prompt       @ 输入密钥提示
64      bl printf
65
66      ldr r0, addr_format_string  @ 设置格式字符串的地址
67      ldr r1, addr_message              @ 设置密钥缓冲区地址
68      bl scanf                  @ 读取密钥
69
70      ldr r0, addr_key_prompt  @ 输入字符串提示
71      bl printf
72
73      ldr r0, addr_key_format  @ 设置格式字符串的地址
74      ldr r1, addr_key         @ 设置消息缓冲区地址
75      bl scanf                 @ 读取消息
76
77      ldr r1, addr_key
78      ldrb r1, [r1]            @ 从密钥缓冲区中获取一个字节
79      ldr r0, addr_message          @ 获取消息指针
80      bl encryptDecrypt        @ 调用加解密函数
81
82      mov r1, r0
83      ldr r0, addr_result_message
84      bl printf                @ 输出加密后的消息
85
86      mov r0, #0          @ 设置返回值为 0
87      pop {r4, pc}        @ 恢复 r4 和 pc 寄存器，并返回
88
89  addr_message_prompt:
90      .word message_prompt
91  addr_key_prompt:
92      .word key_prompt
93  addr_format_string:
94      .word format_string
```

```
95  addr_message:
96      .word message
97  addr_key_format:
98      .word key_format
99  addr_key:
100     .word key
101 addr_result_message:
102     .word result_message
```

```
haboru@haboru-virtual-machine:~/CS/lab2$ arm-linux-gnueabihf-gcc main.s -o main
haboru@haboru-virtual-machine:~/CS/lab2$ qemu-arm ./main
Enter message: 123
Enter encryption key: 1
Encrypted message: 032
```

图 4: 异或加密汇编器和验证结果

三、 分工

许积君：CFG 形式化描述：变量声明，循环、分支，注释。arm 编程：阶乘、gcd

哈博儒：CFG 形式化描述：变量声明，表达式, 函数。arm 编程：异或加密

参考文献