南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

编译原理实验报告

预备工作 2——定义你的编译器 & 汇编编程

谢畅　唐文涛

年级：2021 级

专业：计算机科学与技术

指导教师：王刚

2023 年 10 月 10 日

摘要

本文首先基于我们要实现的 SysY 语言特性，使用上下文无关文法给出了形式化定义。接下来设计了四个 SysY 程序以尽可能包含将要支持的语言特性，并编写等价的 ARM 汇编程序，之后结合对应的 C 语言程序运行结果进行了正确性验证。

关键字：编译器 CFG SysY arm 汇编

目录

1 实验分工、平台与链接

1.1 实验分工

唐文涛负责：1、SysY 语言的形式化定义中的变量声明、基础表达式、算数表达式、注释、作用域和数组相关的 CFG 编写，2、ARM 汇编编程中程序一和程序二的相关工作。

谢畅负责：1、SysY 语言的形式化定义中的常量声明、支持语句、左值表达式、常量表达式、逻辑表达式和函数相关 CFG 的编写，2、ARM 汇编编程中程序三和程序四的相关工作。

1.2 程序的实验平台

- VMware 虚拟机
- X86 架构 Linux 平台
- 操作系统：Ubuntu 22.04.3
- 编译器版本：g++ (Ubuntu 11.4.0-1ubuntu1 22.04) 11.4.0

1.3 arm 程序项目链接

见这里

2 SysY 语言的形式化定义

2.1 实现 SysY 语言特性

在预备工作 1 提出了要实现的 SysY 语言特性，如下：

- 支持 int 和 float 两种数据类型。
- 可以声明变量和常量，进行初始化。
- 支持赋值操作 (=)、表达式语句、语句块、if 条件语句、while 循环和 return 语句。
- 表达式可以进行算术运算（+、-、*、/、%、«、»）、关系运算（==、>、<、>=、<=、!=）和逻辑运算（&&、||、!）。
- 注释
- 输入输出
- 可以声明、调用函数
- 支持变量和常量的作用域，包括在函数和语句块中声明的变量和常量。
- 可以处理一维和二维数组，包括声明和访问数组元素。同时，支持 break 和 continue 语句。

2.2 CFG 描述

我们利用 CFG 对所选 SysY 语言特性子集进行形式化定义，CFG 主要包括终结符集合 V_T，非终结符集合 V_N，开始符号 N，产生式集合 P 。

- 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

2.2.1 开始符号

CompUnit 为开始符号

2.2.2 终结符集合 V_T

终结符或者是单引号括起的串，或者是 ***ID***、**IntConst**、**floatConst** 这样的记号，用于表示不同类型的标识符和常量。其中 ID 表示标识符，而 IntConst 表示整型常量、floatConst 表示浮点数常量。

- 标识符 (ID)

  ID → ID_char | ID ID_char | ID digit

  ID_char → _ | 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 → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- 数值常量

  ndigit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

  IntConst → digit | ndigit { digit }

  floatConst → IntConst[’.’ IntConst]

- 运算符号

  {+ , - , * , / , % , = , ! , < , > , <= , >= , == , != , && , || } (这里的逗号仅为分隔作用)

- 关键字

  { void , int , float , ID , const , IntConst , floatConst , if , while , break , continue , return , else }

- 特殊符号

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

2.2.3 非终结符集合 V_N

非终结符即一些语法变量，是从开始符号到终结符的过渡。在下面的产生式中左部均为非终结符。

2.2.4 产生式集合 P

- 编译单元 CompUnit → [CompUnit] (Decl | FuncDef)
- 声明 Decl → ConstDecl | VarDecl
- 数据类型

  BType → ’**int**’ | ’**float**’

- 变量（数组）声明以及初始化

  变量声明 VarDecl → BType VarDef { ’,’ VarDef } ’;’

  变量定义 VarDef → **ID** { ’[’ ConstExp ’]’ } | **ID** { ’[’ ConstExp ’]’ } ’=’ InitVal

  变量初值 InitVal → Exp | ’{’ [InitVal { ’,’ InitVal }] ’}’

  ({} 代表可以多选，即多维数组)

- 支持常量（数组）声明以及初始化

  常量声明 ConstDecl → ’**const**’ BType ConstDef { ’,’ ConstDef } ’;’

  常数定义 ConstDef → **ID** { ’[’ ConstExp ’]’ } ’=’ ConstInitVal

  常量初值 ConstInitVal → ConstExp | ’{’ [ConstInitVal { ’,’ ConstInitVal }] ’}’

- 左值表达式

  LVal → **ID** {’[’ Exp ’]’}

- 一些基础表达式

  基本表达式 PrimaryExp → ’(’ Exp ’)’ | LVal | Number

  数值 Number → **IntConst** | **floatConst**

  一元表达式 UnaryExp → PrimaryExp | **ID** ’(’ [FuncRParams] ’)’ | UnaryOp UnaryExp

  单目运算符 UnaryOp → ’+’ | ’−’ | ’!’ 注：’!’ 仅出现在条件表达式中

- 算术表达式

  Exp → AddExp

  加减表达式 AddExp → MulExp | AddExp (’+’ | ’−’) MulExp

  乘除模表达式 MulExp → UnaryExp | MulExp (’*’ | ’/’ | ’%’) UnaryExp

- 常量表达式

  ConstExp → AddExp 注：使用的 ID 必须是常量

- 关系表达式

  RelExp → AddExp | RelExp (’<’ | ’>’ | ’<=’ | ’>=’) AddExp

- 逻辑表达式条件表达式 Cond → OrExp

  逻辑或表达式 OrExp → AndExp | OrExp ’||’ AndExp

  逻辑与表达式 AndExp → EqExp | AndExp ’&&’ EqExp

  相等性表达式 EqExp → RelExp | EqExp (’==’ | ’!=’) RelExp

- 支持注释

  anno → ’//’ stmt | ’/*’ stmt ’*/’

- 语句 Stmt → LVal ’=’ Exp ’;’ | [Exp] ’;’ | Block | ’**if**’ ’(Cond ’)’ Stmt [’**else**’ Stmt] | ’**while**’ ’(’ Cond ’)’ Stmt | ’**break**’ ’;’ | ’**continue**’ ’;’ | ’**return**’ [Exp] ’;’ | ’**for**’ (Exp ; Exp ;Exp) Stmt ’;’

- 语句块

  语句块 Block → '{' { BlockItem } '}'

  语句块项 BlockItem → Decl | Stmt

- 函数

  函数定义 FuncDef → FuncType **ID** '(' [FuncFParams] ')' Block

  函数类型 FuncType → '**void**' | '**int**' | '**float**'

  函数形参表 FuncFParams → FuncFParam { ',' FuncFParam }

  函数形参 FuncFParam → BType **ID** ['[' ']' { '[' Exp ']' }]

  函数实参表 FuncRParams → Exp { ',' Exp }

- 变量、常量作用域

  Scope → '{' Stmt'}'

3 ARM 汇编编程

在这一节，我们设计了四个 SysY 程序以尽可能包含将要支持的语言特性，并编写等价的 ARM 汇编程序。由于在 ARM 汇编中无法直接调用 SysY 运行时库，所以我们采用 C 语言中的函数进行输入输出。

ARM 汇编程序编写完成后，使用下面的脚本进行编译运行：

```
;batch.sh
arm-linux-gnueabihf-gcc test.S -o test.out
qemu-arm -L /usr/arm-linux-gnueabihf ./test.out
```

验证正确性时，我们先编写相应的 C 语言程序，结合 C 语言程序运行结果进行验证。

3.1 程序一

程序一为计算 n 的阶乘，其中 n 为用户输入的正整数。主要覆盖了变量声明和定义、输入输出函数调用、循环语句、表达式和赋值语句等 SysY 语言特性。

3.1.1 SysY 程序

```
int main()
{
    int i,n,f;
    //n=getint();
    //使用 scanf 替代
    scanf("%d",&n);
    i=2;
    f=1;
```

```
    while(i<=n)
    {
        f=f*i;
        i=i+1;
    }
    //putint(f);
    //使用 printf 替代
    printf("%d\n",f);
}
```

3.1.2 ARM 汇编程序

在编写对应的 ARM 汇编程序时，首先将常量字符串存入.rodata 数据段。由于 main 函数中会调用其他函数，所以需要保存 lr 寄存器。分配好栈空间后，就可以正式开始翻译对应的 C 语言程序。代码及编写思路如下:

```
        .arch armv7-a
        .text
        .align 2
        .section .rodata @定义常量字符串
        .align 2
_str0:
        .ascii "%d"
        .align 2
_str1:
        .ascii "%d\n"

        .text
        .align 2

        .global main @将main声明为全局变量
main:
        push {fp,lr} @将fp、lr入栈，顺序与寄存器序号有关
        add fp,sp,#4 @fp寄存器指向栈底
        sub sp,sp,#16 @为局部变量分配16字节栈空间
        sub r1,fp,#12 @函数的第二个参数：变量n的地址
        ldr r0,_bridge @函数的第一个参数：格式化字符串_str0的地址
        bl __isoc99_scanf @调用scanf函数

        @为局部变量i和f赋值
        ldr r0,=0x2
        str r0,[fp,#-8]
        ldr r0,=0x1
        str r0,[fp,#-16]

.L1:
```

```
31          @首先判断循环条件
32          ldr r0,[fp,#-8] @i
33          ldr r1,[fp,#-12] @n
34          cmp r0,r1 @i<=n
35          bgt .L2
36
37          @算数运算
38          ldr r0,[fp,#-8] @加载i
39          ldr r1,[fp,#-16] @加载f
40          mul r1,r1,r0 @f=f*i
41          str r1,[fp,#-16] @回写f
42
43          ldr r0,[fp,#-8] @加载i
44          add r0,r0,#1 @i=i+1
45          str r0,[fp,#-8] @回写i
46
47          b .L1
48  .L2:
49          ldr r1,[fp,#-16] @printf()的第二个参数: f
50          ldr r0,_bridge+4 @printf()的第一个参数: 格式化字符串_str1的地址
51          bl printf
52
53          mov r0,#0 @main()函数的返回值
54          add sp,fp,#-4 @恢复sp指针
55          pop {fp,pc} @表示将 fp 寄存器弹出并恢复, 并更新 pc 指针
56
57  _bridge:
58          .word _str0
59          .word _str1
60
61          .section .note.GNU-stack,"",%progbits
```

3.1.3 正确性验证

编译 ARM 汇编代码 fact.S，生成可执行文件 fact.out。执行 fact.out，分别测试 n=4，5，6，程序分别输出 24,120,720，答案正确。

```
inata@inata-virtual-machine:~/compiler/lab2$ arm-linux-gnueabihf-gcc fact.S -g
-o fact.out
inata@inata-virtual-machine:~/compiler/lab2$ qemu-arm -L /usr/arm-linux-gnueabih
f ./fact.out
4
24
inata@inata-virtual-machine:~/compiler/lab2$ qemu-arm -L /usr/arm-linux-gnueabih
f ./fact.out
5
120
inata@inata-virtual-machine:~/compiler/lab2$ qemu-arm -L /usr/arm-linux-gnueabih
f ./fact.out
6
720
inata@inata-virtual-machine:~/compiler/lab2$
```

图 1: 程序一正确性验证

3.2 程序二

由于程序一所包含的语言特性有限，故设计了程序二。主要覆盖了数组、函数定义、函数调用、常量表达式、变量声明和定义、基本表达式等 SysY 语言特性。

3.2.1 SysY 程序

```
int num=1;
int func(int x,int y)
{
    int z=x*y;
    return z;
}

int main(){
    const float zero=0;

    int a[3];
    for(int i=zero;i<3;i++)
    {
        a[i]=i;
        num+=a[i];
    }
    num=num+func(2,3);
    printf("%d\n",num);
}

```

3.2.2 ARM 汇编程序

程序二对应 ARM 汇编的编写重点在于函数的定义与调用，以及数组的实现。

func 函数没有调用其他函数，属于 leaf procedure，故无需保存 lr 寄存器。而 main 函数调用了多个函数，需要保存 lr 寄存器。

main 函数中的数组 a[3] 属于局部变量，位于栈上，所以处理方式与一般局部变量类似，在函数开头分配相应的栈空间即可。在访问时，需要计算好相应的偏移量。

代码及编写思路如下:

```
        .arch armv7-a

@ ---- Non-zero Initialized data ----
        .data
        .align 2
num:
```

```
7           .word 1
8
9           .text
10          .align 2
11          .section .rodata
12          .align 2
13  _str0:
14          .ascii "%d\n"
15
16          .text
17          .align 2
18
19          .global func
20  func:
21          str fp,[sp,#-4]! @leaf procedure 只需保存fp
22          mov fp,sp
23          mul r0,r0,r1 @z=x*y z为返回值, 保存在r0
24          add sp,fp,#0 @恢复sp指针
25          ldr fp,[sp],#4 @从栈中弹出fp, 并更新sp
26          bx lr @恢复sp寄存器
27
28          .global main
29  main:
30          push {fp,lr} @将fp、lr入栈
31          add fp,sp,#4 @fp寄存器指向栈底
32
33          sub sp,sp,#20 @为局部变量分配20字节栈空间
34          ldr r0,=0x0 @r0=0
35          str r0,[fp,#-8] @局部变量zero赋值为0
36
37          @for循环
38          str r0,[fp,#-24] @循环变量i初始化为0
39  .L1:
40          ldr r0,[fp,#-24] @加载i到r0
41          ldr r1,=0x3
42          cmp r0,r1
43          bge .L3 @i>=3则跳出循环
44  .L2:
45          @a[i]的地址为fp-12-i*4
46          sub r2,fp,#12 @fp-12
47          mov r3,r0,lsl #2 @i*4
48          sub r2,r2,r3 @r2中保存a[i]的地址
49          str r0,[r2]@a[i]=i
50
51          ldr r3,_bridge @加载num的地址到r3
52          ldr r1,[r3] @加载num到r1
53          add r1,r1,r0 @num+=1
54          str r1,[r3] @回写num
```

```
55
56          add r0,r0,#1 @i+=1
57          str r0,[fp,#-24] @回写
58          b .L1
59
60  .L3:
61          ldr r1,=0x3 @func的第二个参数
62          ldr r0,=0x2 @func的第一个参数
63          bl func
64
65          ldr r3,_bridge @加载num的地址到r3
66          ldr r1,[r3] @加载num到r1
67          add r1,r1,r0 @num+=func(2,3)
68          str r1,[r3] @回写num
69          ldr r0,_bridge+4 @printf的第一个参数，第二个参数num已经加载到r1
70          bl printf
71
72          mov r0,#0 @main函数的返回值
73          add sp,fp,#-4 @恢复sp指针
74          pop {fp,pc} @将 fp 寄存器弹出并恢复，并更新 pc 指针
75
76
77  _bridge:
78          .word num
79          .word _str0
80
81          .section .note.GNU-stack,"",%progbits
```

3.2.3 正确性验证

编译 ARM 汇编代码 test.S，生成可执行文件 test.out。执行 test.out，程序输出 10。使用 gcc 编写对应 C 语言代码 test.c，执行后输出 10。正确性得到验证。

```
inata@inata-virtual-machine:~/compiler/lab2$ arm-linux-gnueabihf-gcc test.s -g
-o test.out
inata@inata-virtual-machine:~/compiler/lab2$ qemu-arm -L /usr/arm-linux-gnueabih
f ./test.out
10
inata@inata-virtual-machine:~/compiler/lab2$ gcc test.c -o test
inata@inata-virtual-machine:~/compiler/lab2$ ./test
10
inata@inata-virtual-machine:~/compiler/lab2$
```

图 2: 程序二正确性验证

3.3 程序三

程序三针对整型变量、全局一维数组的存储以及读取，函数、循环语句，各种算术指令 (移位、模运算、加减乘除)，逻辑运算 (大小的比较)、跳转语句，最后循环内部的 continue 等功能也有实现。总而言之，对整型的运算进行较为全面的覆盖

3.3.1 SysY 程序

```c
#include<stdio.h>
#define constant 2
int m[10];
int caculate(int a,int b,int c){
    int temp=b<<c;
    a=a+temp;
    return a;
}
int main(){
    int n;
    scanf("%d",&n);
    int i,j,k;
    i=5;j=constant;k=constant;
    if(i<j+k){
        i++;//make i bigger
    }
    else{
        i--;
    }
    n=n+caculate(i,j,k);
    for(i=0;i<10;i++){
        m[i]=i%5;
        if(i==5){
            m[i]+=2;
        }
        else{
            continue;
        }
    }
    n/=m[5];
    printf("%d",n);
}
```

3.3.2 Arm 汇编程序

在这里我们需要定义一个**全局变量数组**，因为不需要初始化，直接.comm m,40 即可定义一个 int m[10] 的数组。在访问此数组时，先通过符号表项，用 ldr 得到 m 的首地址，再通过偏移

值访问不同索引的 m[i] 即可。

在该程序中，重点需要解决众多运算。**加减法**非常简单，直接 add/sub 即可。但是，整型变量的**除法**似乎不支持直接 div/udiv。所以在这里调用 arm 一个自带函数 ___aeabi_idiv，得出的商在 r0 中。

而这里**移位**运算采取的是 lsl，即可得出移位的结果值；**模运算**在这里较为复杂，由于上述除法的问题，必须首先调用 ___aeabi_idiv，得出商的值，再将商与除数做乘法，最后用被除数-商 * 除数，得出的即为模的结果；**乘法**运算在这里用 mul 即可。

最后这里的相关条件判断，用 cmp+ 相关 ble/bge/beq 等语句即可，根据 cmp 执行完设置好的标识符，即可进行相关条件分支跳转。

```
1     .arch armv7-a
2     .comm m,40  @全局变量数组 int m[10]
3     .text
4     .align 2
5     .section .rodata
6     .align 2
7  _str0:
8     .ascii "%d\0"  @整型格式化字符串的常量字符串，在scanf和printf中使用到
9     .text
10    .align 2
11
12    .global calculate
13 calculate:  @int caculate(int a,int b,int c)
14    str fp,[sp,#-4]! @这里为叶过程,不需要将lr入栈,将fp入栈即可
15    mov fp,sp    @fp栈底,sp栈顶
16    sub sp,sp,#12 @分配空间给局部变量
17    str r0,[fp,#-4]   @r0=[fp,#-4]=a
18    str r1,[fp,#-8]   @r1=[fp,#-8]=b
19    lsl r3,r1,r2   @r3=r1<<r2
20    str r3,[fp,#-12] @b<<c储存
21    ldr r1,[fp,#-12]
22    ldr r0,[fp,#-4]
23    add r0,r0,r1   @a+b<<c
24    add sp,fp,#0       @将栈指针sp恢复到初始值
25    ldr fp,[sp],#4    @从栈中弹出fp寄存器,更新sp指针 sp=sp+4
26    bx lr   @恢复pc指针
27
28    .global main
29 main:
30    push {fp, lr}
31    add fp,sp,#0      @fp指向栈底
32    sub sp,sp,#24    @预留24个字节空间
33    sub r1,fp,#4      @函数第二个参数,n的地址
34    ldr r0,_bridge   @printf第一个参数,整型的格式化字符串
35    bl __isoc99_scanf       @scanf("%d",&n)
36    @为i,j,k赋值
37    mov r0,#5
38    str r0,[fp,#-8] @i=5
```

```
39      mov r0,#2
40      str r0,[fp,#-12]@j=2
41      mov r0,#2
42      str r0,[fp,#-16]@k=2
43      @if else语句
44      ldr r0,[fp,#-12] @加载j到r0
45      ldr r1,[fp,#-16] @加载k到r1
46      add r0,r0,r1   @r0中的值为j+k
47      ldr r1,[fp,#-8]  @加载i到r1
48      cmp r1,r0    @比较i与j+k大小
49      blt .L1      @i<j+k 跳转至.L1
50      ldr r0,[fp,#-8]    @i>=j+k 开始i--
51      add r0,r0,#-1
52      str r0,[fp,#-8]
53      b .L2
54  .L1:                    @i++操作
55      ldr r0,[fp,#-8]
56      add r0,r0,#1
57      str r0,[fp,#-8]
58  .L2:
59      ldr r2,[fp,#-16]
60      ldr r1,[fp,#-12]
61      ldr r0,[fp,#-8]
62      bl calculate     @准备calculate的三个参数a,b,c
63      ldr r1,[fp,#-4]  @取出n
64      add r1,r0,r1
65      str r1,[fp,#-4] @n=n+caculate(i,j,k)
66      @for 循环,准备i计数器
67      mov r0,#0
68      str r0,[fp,#-8]  @计数器赋值为0,i=0
69  .L3:
70      mov r1,#10
71      ldr r0,[fp,#-8]
72      cmp r0,r1  @比较i与10
73      bge .L7  @i>=10,跳出循环
74  .L4:
75      ldr r0,[fp,#-8]    @取出i
76      mov r1,#5
77      bl __aeabi_idiv  @ i/5 商在r0中
78
79      mul r0,r0,r1 @商*5
80
81      ldr r1,[fp,#-8]
82      sub r3,r1,r0   @i-商*5 余数
83
84      ldr r1,[fp,#-8]
85      ldr r2,_bridge+4    @将m的起始地址赋给r2
86      add r2,r2,r1,lsl #2    @i*4偏移,获取m[i]地址
```

```
87      str r3,[r2]  @m[i]=i%5,r3中为模结果
88
89      ldr r0,[fp,#-8]  @r0赋值为i
90      cmp r0,r1    @比较i和5
91      beq .L5  @如果i==5,需要进行m[i]+=2
92      b .L6    @i!=5,直接continue
93  .L5:
94      ldr r3,[r2]
95      add r3,r3,#2
96      str r3,[r2]
97      b .L6
98  .L6:      @计数器i的更新,跳转L3,检查下一次循环的条件
99      ldr r0,[fp,#-8] @i
100     add r0,r0,#1
101     str r0,[fp,#-8]  @i++
102     b .L3
103 .L7:      @跳出for循环后
104     ldr r0,_bridge+4
105     add r0,r0,#20  @得到m[5]的地址
106     ldr r3,[r0] @将m[5]取出, 赋值给r3
107
108     mov r1,r3    @r1中存的是m[5]
109     ldr r0,[fp,#-4] @r0中存的是n
110     bl __aeabi_idiv @n/m[5] 进行整型除法运算, 商在r0中
111
112     mov r1,r0
113     ldr r0,_bridge
114     bl printf    @打印最后的结果
115
116     mov r0,#0 @返回值
117     add sp,sp,#24 @恢复sp
118     pop {fp,pc}
119
120 _bridge:
121     .word _str0
122     .word m
```

3.3.3 正确性验证

```
prayer@prayer-virtual-machine:~/compile/lab2$ bash f.sh
1
6prayer@prayer-virtual-machine:~/compile/lab2$ bash f.sh
2
7prayer@prayer-virtual-machine:~/compile/lab2$ bash f.sh
10
11prayer@prayer-virtual-machine:~/compile/lab2$ bash f.sh
20
16prayer@prayer-virtual-machine:~/compile/lab2$ bash f.sh
50
31prayer@prayer-virtual-machine:~/compile/lab2$
```

图 3: 程序三正确性验证

由 SysY 代码逻辑，程序最终输出为 (n+i-1+j«k)/m[5]，即 **(n+12)/2**，当 n=1 时，结果为 6.5，化为整型是 6；当 n=2 时，结果为 7；当 n=10 时，结果为 11；当 n 为 20 时，结果为 16；当 n=50 时，结果为 31。与上图程序的结果一致，arm 程序正确。

3.4 程序四

程序四针对二维数组进行探究，并对多重循环语句进行相关研究，主要涉及的是跳转语句，与全局内存的访存，对比一维数组与二维数组的差异性，以及多重循环的实现

3.4.1 SysY 程序

```
#include<stdio.h>
int m[10][10];
int main(){
    int n,k;
    for(int i=0;i<10;i++){
        for(int j=0;j<10;j++){
            m[i][j]=i+j;
        }
    }
    n=m[2][6];
    k=m[3][4];
    printf("n=%d\n",n);
    printf("k=%d\n",n);

}
```

3.4.2 Arm 汇编程序

设计思路如下:

针对二维数组，这里同样是开辟一个全局的变量，大小为 400 字节，正好是 int m[10][10] 对应的大小。因此我们将二维数组映射到一个连续的内存空间，在访问时计算出对应的偏移地址即可。也就是，首先通过符号表项数 _bridge 先获取到 m 的地址，再计算出 m[i][j] 相对于 m 的偏移地址，即可通过 m+ 偏移地址访问到对应的二维数组元素。

而对于多重循环，这里的编写思路是，最开始初始化外层循环的计数器值，然后先定义一个外层循环，先判断循环终止条件是否满足，然后初始化内层循环的计数器值，在内层循环开始位置判断循环条件，终止那么跳入外层循环的计数器更新位置；继续，则完成内层循环的一系列运算以及赋值语句。在外层循环的计数器更新位置，还要继续跳转至外层循环开始处。

最后循环结束，跳出外层循环，开始打印 n 与 K 的值。在结束所有操作后，执行返回值的赋值操作，然后恢复 sp，释放栈的空间，恢复 fp，更新 pc 值。

```
1         .arch armv7-a
2         .comm m,400  @全局变量数组 int m[10][10]
3     .text
4     .align 2
5         .section         .rodata
6         .align  2
7 _str0:
8     .ascii "n=%d\n"  @整型格式化字符串的常量字符串,在scanf和printf中使用到
9     .align 2
10 _str1:
11    .ascii "k=%d\n"  @整型格式化字符串的常量字符串,在scanf和printf中使用到
12    .text
13    .align 2
14    .global main
15 main:
16    push {fp, lr}
17    add fp,sp,#0     @fp指向栈底
18    sub sp,sp,#16    @预留16个字节空间
19    mov r0,#0
20    str r0,[fp,#-8] @i=0
21 .L1:                @外层循环
22    mov r1,#10
23    ldr r0,[fp,#-8] @取出i
24    cmp r0,r1    @i与10比较
25    bge .L5    @跳出外层循环
26 .L2:
27    ldr r0,[fp,#-8] @取出i
28    mov r1,#0
29    str r1,[fp,#-12] @j=0 内层循环计数器
30    b .L3  @进入内层循环
31 .L3:
32    ldr r1,[fp,#-12]  @取出j
33    mov r2,#10      @j与10
34    cmp r1,r2   @j与10比大小
35    bge .L4   @j>=10跳出内层循环
36    ldr r0,[fp,#-8] @取出i
37    add r2,r0,r1 @i+j储存至r2中
```

```
38      ldr r3,_bridge+8  @取出m的地址
39      @ i*10*4+j*4
40      mov r1,#40
41      mul r0,r0,r1   @i*40
42      ldr r1,[fp,#-12] @取出j
43      add r0,r0,r1,lsl #2 @i*40+4*j
44      add r3,r3,r0   @m[i][j]的地址
45      str r2,[r3]   @赋值操作,m[i][j]=i+j
46      @内层循环计数器j更新
47      ldr r1,[fp,#-12]   @取出j的值
48      add r1,r1,#1   @j++
49      str r1,[fp,#-12] @j++
50      b .L3
51  .L4:  @外层循环计数器i更新
52      ldr r0,[fp,#-8]
53      add r0,r0,#1
54      str r0,[fp,#-8]   @i++
55      b .L1
56  .L5:    @跳出循环后打印m[2][6]和m[3][4]
57      ldr r0,_bridge+8
58      mov r1,#80    @2*10*4
59      mov r2,#24    @6*4
60      add r1,r1,r2
61      add r0,r0,r1
62      ldr r2,[r0]
63      str r2,[fp,#-4]   @赋值给n
64      ldr r1,[fp,#-4]
65      ldr r0,_bridge
66      bl printf     @打印n
67
68      ldr r0,_bridge+8
69      mov r1,#120   @3*10*4
70      mov r2,#16    @4*4
71      add r1,r1,r2
72      add r0,r0,r1
73      ldr r2,[r0]
74      str r2,[fp,#-16]   @赋值给k
75      ldr r1,[fp,#-16]
76      ldr r0,_bridge+4
77      bl printf    @打印k
78
79      mov r0,#0 @返回值
80      add sp,sp,#16 @恢复sp
81      pop {fp,pc}    @弹出fp恢复, 更新pc
82
83  _bridge:
84      .word _str0
85      .word _str1
```

```
86      .word m
```

3.4.3 正确性验证

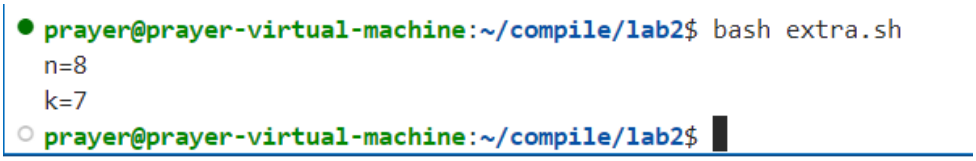

图 4: 程序四正确性验证

如上图，我们可以看到正确输出 n 与 K 的值。n 的值是 m[2][6]，而 m[2][6] 的值是 8；k 的值是 m[3][4]，而 m[3][4] 的值是 7, 说明我们的 arm 程序正确

4 思考

如果不是人“手工编译”，而是要实现一个计算机程序（编译器）来将 SysY 程序转换为汇编程序，应该如何做 (这个编译器程序的数据结构和算法设计)?

我们在预备工作 1 已经了解到了整个编译过程会分为词法分析、语法分析、语义分析、中间代码生成、代码优化和代码生成这几个阶段，因为不太了解后面的阶段，所以在这里重点分析一下词法分析和语法分析、语义分析阶段。

- 首先利用上下文无关文法将输入的 SysY 程序源代码转换成词法单元。数据结构上，创建一个符号表来跟踪变量、函数和其他标识符的信息。符号表用于查找变量的地址、类型信息等。算法设计上，通过有限状态自动机 DFA、正则表达式来识别不同类型的词法单元，构建 token 列表，具体的算法可以是基于 DFA 的模式匹配器等。
- 利用语法制导翻译，数据结构采用语法分析树的形式，将语义动作嵌入树的节点中。算法设计上，有递归下降分析、LL 分析或 LR 分析等算法来构建 AST。可采用递归下降分析的预测分析法，根据对下一个单词的首单词选择候选式。找到终结符，对每一个非终结符编写递归函数；左递归改为右递归。
- 接下来在抽象语法树上进行语义分析，检查程序是否符合语义规则，如类型检查、变量声明检查、函数调用检查等。其中符号表是关键数据结构，用于存储有关变量、函数、类型等的信息。每个符号表项包括标识符名称、类型、作用域等信息。
- 最后设计语法制导定义/翻译模式实现 SysY 程序到汇编程序的翻译。代码生成算法将 AST 节点转换为目标汇编代码，涉及到寄存器分配、指令选择、指令调度等。

参考文献