南開大學
Nankai University

计算机学院

编译系统原理实验报告

定义语言特性及汇编编程

成员 1：熊宇轩

学号：2010056MINOR

成员 2：王天鹏

学号：2013013

2022 年 10 月

Abstract

在上次实验中，我们了解了编译器的工作流程，并对 LLVM IR 这种中间生成结果有了初步的了解。在这一次实验中，我们将使用 CFG（上下文无关文法，Context-free Grammar）来定义我们自己的编译器所处理的语言的特性，并通过手动编写 ARM 和 X86 汇编程序，来实现对我们编译器生成结果进一步的认识。

关键字： 上下文无关文法；ARM 汇编；X86 汇编

目录

1 CFG 定义

1.1 变量定义

首先是最基础的变量定义：

```
varDeclare -> type idList
type       -> int
idList     -> id | id=num | idList, id | idList, id = num
num        -> digit*
digit      -> [0-9]

// VT = {int}
// VN = {varDeclare, type, idList}
// S = varDeclare
```

其中，num为满足条件的整数，这里简单定义为了单个数字的闭包。idList则是定义变量的标识符列表，其中我们还定义了对定义变量同时赋值的操作。

1.2 各类表达式

定义了变量后，我们还需要对其进行运算，这就是表达式所需要做的工作了，其定义如下：

```
expr      -> varAssign
varAssign -> eqExpr | id = eqExpr | id = varAssign

eqExpr    -> cmpExpr | eqExpr == cmpExpr | eqExpr != cmpExpr
cmpExpr   -> addExpr |
             -> cmpExpr > addExpr | cmpExpr < addExpr |
             -> cmpExpr >= addExpr | cmpExpr <= addExpr
addExpr   -> mulExpr | addExpr + mulExpr | addExpr - mulExpr
mulExpr   -> unExpr | mulExpr * unExpr | mulExpr / unExpr | mulExpr % unExpr
unExpr    -> (expr) | num | id | id++ | id-- | funCall

// VT = {id, num}
// VN = {varAssign, eqExpr, cmpExpr, addExpr, mulExpr, unExpr}
// S = expr
```

通过 CFG 定义表达式时，尤其需要注意运算的优先级，在书写产生式 R 时，我们需要按照运算优先级的高低由下向上书写，也需要注意运算符的左结合性。例如，最下面的unExpr代表了括号、后缀增减、函数调用表达式，mulExpr则表示乘除模运算，addExpr表示加减运算，comExpr表示大小比较运算，eqExpr表示判等、判不等运算。除了一般运算符表达式外，我们还需要实现较为特殊的赋值

表达式，这便是我们产生式 R 中的`varAssign`。编写该 CFG 时小组成员参考了 CppReference 上的运算符优先级。[1]

表达式 CFG 的产生式 R 比较复杂，考虑到这次实验还属于准备工作的范畴，我们并没有一一列出全部打算实现的运算符。

1.3 控制语句

以下是循环和分支语句的 CFG 定义：

```
ctrlStmt  -> ifStmt | whileStmt | forStmt
ifStmt    -> if(expr) stmt else stmt
whileStmt -> while(expr) stmt
forStmt   -> for(expr; expr; expr) stmt

// VT = {for, while, if}
// VN = {ifStmt, whileStmt, forStmt}
// S = ctrlStmt
```

除了循环和分支这样的控制语句，我们的语句还应该包含复合语句，其 CFG 如下：

```
stmt      -> stmt | ctrlStmt | compStmt | singStmt
compStmt  -> {multiStmt}
multiStmt -> singStmt | singStmt multiStmt
singStmt  -> expr; | return expr; | return;

// VT = {return}
// VN = {singStmt, multiStmt, compStmt}
// S = stmt
```

需要注意的是，返回语句并不是表达式，需要单独定义一下，包含到`stmt`中，以方便进行下文函数 CFG 的定义。

1.4 函数定义和调用

以下是函数定义的 CFG：

```
func -> type fName (args) stmt
type -> int
args -> arg | args, arg | epsilon
arg  -> type id | type id = num
// VT = {int}
// VN = {arg, args, type}
// S = func
```

其中，fName为函数名，args为形参表，arg为单个形参。需要注意的是，形参表可以为空。对于有默认值的形参，也需要做一下处理。

有了函数定义，还需要对其进行调用，函数调用的 CFG 如下：

```
funCall -> fName(paras)
paras -> id | num | paras, id | paras, num | epsilon
// VT = epsilon
// VN = {paras}
// S = funCall
```

需要注意的是，函数同样可以是无传入参数的。

2 汇编编程

2.1 C 函数源代码

对于预备工作 1 中的阶乘程序进行该改写来体现体现 C 语言中的阶乘特性，并考虑了相关的边界情况。主要改动是将变量均改变成全局变量，阶乘使用 factorial 函数进行计算，main 函数仅负责输入输出。改写后的 C 代码如下：

```c
#include <stdio.h>
int i = 2, f = 1;
int n=0;
int factorial(int n)
{

    if (n < 0)
        return -1;
    while (i <= n)
    {
        f = f * i;
        i = i + 1;
    }
    return f;
}

int main()
{
    scanf("%d", &n);
    n = factorial(n);
    printf("%d", n);
```

```
22      return 0;
23  }
```

依照上述 c 语言代码编写其汇编语言版本, 分为 X86 体系的 AT&T 汇编和 ARM 体系下的 arm 汇编两种方式进行实验。

2.2 ARM 汇编

小组成员依据实验指导要求, 在 arm 汇编中出于编程方便, 将涉及的参数 f、i、n 定义为全局变量, 提取出常量%d, 将程序主体部分专门写成一个函数 factorial 在 main 主程序段中调用 scanf、factorial、printf 等函数实现对阶乘程序的输入输出, 其中编写的 arm 汇编程序如下。

```
1   # 赋值全局变量
2   i:
3   .word   2
4   f:
5   .word   1
6   n:
7   .zero   4
8   # 定义 factorial 函数
9   factorial(int):
10  # 为函数分配 16 个字节长度的栈帧内存。
11  sub     sp, sp, #16
12  str     w0, [sp, 12]
13  ldr     w0, [sp, 12]
14  #if (n < 0)
15  cmp     w0, 0
16  bge     .L4
17  #return -1;
18  mov     w0, -1
19  b       .L3
20  .L5:
21  adrp    x0, f
22  add     x0, x0, :lo12:f
23  ldr     w1, [x0]
24  adrp    x0, i
25  add     x0, x0, :lo12:i
26  #f = f * i;
27  ldr     w0, [x0]
28  mul     w1, w1, w0
29  adrp    x0, f
30  add     x0, x0, :lo12:f
```

```
31  str     w1, [x0]
32  adrp    x0, i
33  add     x0, x0, :lo12:i
34  #i = i + 1;
35  ldr     w0, [x0]
36  add     w1, w0, 1
37  adrp    x0, i
38  add     x0, x0, :lo12:i
39  str     w1, [x0]
40  .L4:
41  #x0 寄存器赋值成 i，并放在 w0 中保存
42  adrp    x0, i
43  add     x0, x0, :lo12:i
44  ldr     w0, [x0]
45  #w1 保存 n 的值
46  ldr     w1, [sp, 12]
47  #i<=n
48  cmp     w1, w0
49  # 满足条件，跳转至 L5 段
50  bge     .L5
51  # 不满足条件，函数返回 f 值
52  adrp    x0, f
53  add     x0, x0, :lo12:f
54  ldr     w0, [x0]
55  .L3:
56  add     sp, sp, 16
57  ret
58  # 保存全局变量%d
59  .LC0:
60  .string "%d"
61  main:
62   # 传入 LC0 段的值，调用 scanf 函数输入 n 的值
63  stp     x29, x30, [sp, -16]!
64  add     x29, sp, 0
65  adrp    x0, n
66  add     x1, x0, :lo12:n
67  adrp    x0, .LC0
68  add     x0, x0, :lo12:.LC0
69  bl      scanf
70  #w0 储存 n 的值传入并调用 factorial 函数
71  adrp    x0, n
72  add     x0, x0, :lo12:n
```

```
ldr     w0, [x0]
bl      factorial(int)
# 给 n 赋值 factorial 的返回值
mov     w1, w0
adrp    x0, n
add     x0, x0, :lo12:n
str     w1, [x0]
adrp    x0, n
# 传入%d,n 并调用 printf 函数输出
add     x0, x0, :lo12:n
ldr     w1, [x0]
adrp    x0, .LC0
add     x0, x0, :lo12:.LC0
bl      printf
#return
mov     w0, 0
ldp     x29, x30, [sp], 16
ret
```

小组成员也通过 gcc 编译器输出了 arm 汇编代码如下：

```

i:
        .word   2
f:
        .word   1
n:
factorial(int):
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        str     r0, [r7, #4]
        ldr     r3, [r7, #4]
        cmp     r3, #0
        bge     .L4
        mov     r3, #-1
        b       .L3
.L5:
        movw    r3, #:lower16:f
        movt    r3, #:upper16:f
        ldr     r2, [r3]
        movw    r3, #:lower16:i
```

```
22          movt    r3, #:upper16:i
23          ldr     r3, [r3]
24          mul     r2, r3, r2
25          movw    r3, #:lower16:f
26          movt    r3, #:upper16:f
27          str     r2, [r3]
28          movw    r3, #:lower16:i
29          movt    r3, #:upper16:i
30          ldr     r3, [r3]
31          adds    r2, r3, #1
32          movw    r3, #:lower16:i
33          movt    r3, #:upper16:i
34          str     r2, [r3]
35  .L4:
36          movw    r3, #:lower16:i
37          movt    r3, #:upper16:i
38          ldr     r3, [r3]
39          ldr     r2, [r7, #4]
40          cmp     r2, r3
41          bge     .L5
42          movw    r3, #:lower16:f
43          movt    r3, #:upper16:f
44          ldr     r3, [r3]
45  .L3:
46          mov     r0, r3
47          adds    r7, r7, #12
48          mov     sp, r7
49          ldr     r7, [sp], #4
50          bx      lr
51  .LC0:
52          .ascii  "%d\000"
53  main:
54          push    {r7, lr}
55          add     r7, sp, #0
56          movw    r1, #:lower16:n
57          movt    r1, #:upper16:n
58          movw    r0, #:lower16:.LC0
59          movt    r0, #:upper16:.LC0
60          bl      __isoc99_scanf
61          movw    r3, #:lower16:n
62          movt    r3, #:upper16:n
63          ldr     r3, [r3]
```

```
64          mov     r0, r3
65          bl      factorial(int)
66          mov     r2, r0
67          movw    r3, #:lower16:n
68          movt    r3, #:upper16:n
69          str     r2, [r3]
70          movw    r3, #:lower16:n
71          movt    r3, #:upper16:n
72          ldr     r3, [r3]
73          mov     r1, r3
74          movw    r0, #:lower16:.LC0
75          movt    r0, #:upper16:.LC0
76          bl      printf
77          movs    r3, #0
78          mov     r0, r3
79          pop     {r7, pc}
```

不难看出，gcc 输出的 ARM 汇编程序在分段和跳转等重要程序步骤和小组成员手工编写的汇编程序表现一直。存在一些细节上的差异，如小组成员编写的汇编程序为函数分配 16 个字节长度的栈帧内存，而 gcc 输出的 ARM 汇编程序为函数分配 12 个字节长度的栈帧内存。小组成员使用到了 W0、W1 等 ARM64 下新增的的 32 位零寄存器，而 gcc 输出的 ARM 汇编程序只使用了传统的 r0、r1、r2 等传统 ARM32 位寄存器。在函数调用这一方面，小组成员对于 scanf 函数直接调用 scanf 进行输入，而 gcc 输出的 ARM 汇编程序对于 scanf 的调用处理上有所不同，使用 ___isoc99_scanf 函数进行输入。除此之外在寄存器的使用方面也存在一些差异。[2]

2.3 X86 汇编

小组成员依据实验指导要求，将全局变量，常量%d 和函数 factorial 写成汇编代码形式，然后在 main 主程序段中调用 scanf、factorial、printf 等函数实现对阶乘程序的输入输出，同样实现了 AT&T 格式下 X86 汇编代码，具体程序如下。

```
1   # 函数 factorial
2       .text
3       .globl    factorial
4       .type    factorial, @function
5   factorial:
6       # if n<=0
7           movl n, %eax
8           cmpl $0, %eax
9           jl L2
10      #while (i <= n)
11      movl i, %ebx
```

```
12      cmpl %ebx,%eax
13      jle L3
14      ret
15      #return -1;
16          L2:
17                  movl -1(%esp), %eax
18                  ret
19      #f=f*i  i++
20          L3:  movl f, %eax
21                  movl 4(%esp), %ebx
22                  imull 8(%esp), %eax
23                  addl $1, i
24
25  # 常量
26      .section .rodata
27  STR0:
28      .string "%d"
29  # 全局变量 将 f、i、n 均设为全局变量
30      .globl    f
31      .data
32      .align 4
33      .type    f, @object
34      .size    f, 4
35  f:
36      .long    i
37
38      .globl    i
39      .data
40      .align 4
41      .type    i, @object
42      .size    i, 4
43  i:
44      .long    2
45
46      .globl    n
47      .data
48      .align 4
49      .type    n, @object
50      .size    n, 4
51  n:
52      .long    0
53  # 主函数
```

```
    .text
    .globl main
    .type main, @function
main:
# scanf("%d", &n);
    pushl $n
    pushl $STR0
    call scanf
    addl $8, %esp
#n = factorial(n);
    pushl n
    call factorial
        addl $8, %esp
# printf("%d", n);
    pushl n
        pushl $STR0
        call printf
        addl $8, %esp
# return 0;
        xorl %eax, %eax
        ret
# 可执行堆栈段
    .section .note.GNU-stack,"",@progbits
```

其中编写汇编代码过程中最初遇到错误出现错误，因此考虑将全局变量不能放置在汇编程序开头单独声明为全局标识符，并使用.long 标签对 32 位整型变量进行初始化，同时加上.global 标识符将该变量设为全局变量，便于后续代码调用。

小组成员也对阶乘程序使用 gcc -s 命令输出了 gcc 的编译结果，经过比较后发现，gcc 的汇编代码更长，且加入了更多的编译器标记。在代码逻辑上与小组成员编写的代码基本一致，但进行了相关的优化。值得注意的是，gcc 编译出的汇编代码在程序结束返回部分额外增加了堆栈完整性检查，因此 gcc 给出的汇编程序存在两个 ret 指令，分别是正常返回（return 0）和堆栈异常时的返回。

```
.file   "jiecheng.c"
.text
.globl   factorial
.type   factorial, @function

factorial:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
```

```
11      .cfi_offset 6, -16
12      movq    %rsp, %rbp
13      .cfi_def_cfa_register 6
14      movl    %edi, -20(%rbp)
15      movl    $2, -8(%rbp)
16      movl    $1, -4(%rbp)
17      cmpl    $0, -20(%rbp)
18      jns     .L4
19      movl    $-1, %eax
20      jmp     .L3
21  .L5:
22      movl    -4(%rbp), %eax
23      imull    -8(%rbp), %eax
24      movl    %eax, -4(%rbp)
25      addl    $1, -8(%rbp)
26  .L4:
27      movl    -8(%rbp), %eax
28      cmpl    -20(%rbp), %eax
29      jle     .L5
30      movl    -4(%rbp), %eax
31  .L3:
32      popq    %rbp
33      .cfi_def_cfa 7, 8
34      ret
35      .cfi_endproc
36  .LFE0:
37      .size    factorial, .-factorial
38      .section    .rodata
39  .LC0:
40      .string    "%d"
41      .text
42      .globl    main
43      .type    main, @function
44  main:
45  .LFB1:
46      .cfi_startproc
47      pushq    %rbp
48      .cfi_def_cfa_offset 16
49      .cfi_offset 6, -16
50      movq    %rsp, %rbp
51      .cfi_def_cfa_register 6
52      subq    $16, %rsp
```

```
53      movq    %fs:40, %rax
54      movq    %rax, -8(%rbp)
55      xorl    %eax, %eax
56      leaq    -12(%rbp), %rax
57      movq    %rax, %rsi
58      leaq    .LC0(%rip), %rdi
59      movl    $0, %eax
60      call    __isoc99_scanf@PLT
61      movl    -12(%rbp), %eax
62      movl    %eax, %edi
63      call    factorial
64      movl    %eax, -12(%rbp)
65      movl    -12(%rbp), %eax
66      movl    %eax, %esi
67      leaq    .LC0(%rip), %rdi
68      movl    $0, %eax
69      call    printf@PLT
70      movl    $0, %eax
71      movq    -8(%rbp), %rdx
72      xorq    %fs:40, %rdx
73      je    .L8
74      call    __stack_chk_fail@PLT
75  .L8:
76      leave
77      .cfi_def_cfa 7, 8
78      ret
79      .cfi_endproc
80  .LFE1:
81      .size    main, .-main
82      .ident    "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
83      .section    .note.GNU-stack,"",@progbits
```

3 小组分工

小组目前暂定分工为：熊宇轩同学负责变量声明、赋值、运算符等特性，王天鹏同学负责循环、分支和函数调用等特性，由于小组成员目前对编译相关知识了解还不多，后期相关任务分配可能会有调整。

在这次的实验中，小组两名同学都参与了各自负责部分的 CFG、汇编程序的设计与编写，并一同参与了调试与测试工作。

参考文献

[1] Cppreference. C++ 运算符优先级 - cppreference.com. https://zh.cppreference.com/w/cpp/language/operator_precedence.

[2] Azeria Labs. Writing arm assembly. https://azeria-labs.com/writing-arm-assembly-part-1/.