南开大学
Nankai University

南　开　大　学

网络空间安全学院

编译原理实验报告

期末作业设计

陈豪斌 1911397 许家威 1911500

年级：2019 级

专业：信息安全

指导教师：王刚

2021 年 10 月 10 日

摘要

本文以一个简单的 C++ 小程序为例，其中使用了宏展开、嵌套循环、函数递归调用、全局变量、静态变量、多态等多个方面的语言特性，并利用 LLVM-Clang 作为编译器对 C++ 程序进行编译，以此查看编译器对程序编译完成的一个流程。

关键字：Compiler; LLVM-Clang

This report takes a simple C++ program as an example that uses numerous language characteristics including macro expansion, nested loop, function recursive calls, global variables, static variables and polymorphism and uses LLVM-Clang as the compiler for the program, to figure out the full procedure of compilation.

Keywords：Compiler; LLVM-Clang

目录

一、 编译器概述

(一) 编译器语言设定

我们的编译器使用 SysY 语言，SysY 语言是 C 语言的一个子集。每个 SysY 程序的源码存储在一个扩展名为 sy 的文件中。该文件中有且仅有一个名为 main 的主函数定义，还可以包含若干全局变量声明、常量声明和其他函数定义。我们也将对 SysY 的常规功能进行扩展。

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，编译器中的 I/O 部分我们将会通过 SysY 运行时库进行实现。

(二) 语言结构和编译器功能概述

首先，我们的编译器将实现变量的声明与定义。变量用于开辟一片内存区域，在条件允许的情况下可以直接使用寄存器直接实现，以加快程序运行速度。我们也将支持变量对应的数组和指针。变量的不同类型对应着不同的运算方式，详细支持的变量类型可见后文。

函数是结构化组织程序的重要功能，是我们编译器必须要实现的内容。函数可以带参数也可以不带参数，参数可以是任何变量类型，对于数组类的参数我们将只传递指针，并且只有第一维的长度可以空缺。函数可以返回各种变量类型的值，或者不返回值。函数体由若干变量声明和语句组成。

语句是程序语言的重要组成部分，语句包括赋值语句，表达式语句、语句块、if 语句、while 语句、break 语句、continue 语句、return 语句等，同时也包含空语句。语句块中可以包含若干变量声明和语句。我们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。算符的优先级和结合性以及计算规则 (含逻辑运算的“短路计算”) 与 C 语言一致，具体可见后文。

二、 编译器功能详细说明

(一) 变量类型

我们将支持如下基本变量类型。

- int 类型，此类型表示有符号整型数据，只能储存整数，由 4 个字节构成，最高位为符号位。
- float 类型，此类型表示有符号浮点型数据，可以存储小数，由 4 个字节构成，我们将调用 FPU 的相关指令进行实现。
- char 类型，此类型表示字符类型数据，使用的字符集为 ASCII，由 1 个字节构成。
- bool 类型，此类型表示布尔类型数据，只有真和假两种状态，由 1 个字节构成。
- 指针类型，此类型存储了各个变量的内存地址，根据计算机实际情况可以为 4 个字节或者 8 个字节构成。

我们也将计划支持如下特殊数据类型。

- 结构体 struct，结构体是一个各种数据类型的集合，可以按照集合中所有数据的总体大小来分配相应大小内存。

- 数组，数组是多个相同数据类型变量的序列，占用内存和此序列中变量个数相关，数组的访问通过首地址进行访问。
- 多维数组，多维数组及数组的数组，实现方法和普通数组相同。
- 字符串 char*，字符串为一个固定长度的字符变量数组，用 0 表示结尾，实现方式和数组相同。

(二) 函数

函数包括了如下部分。

- 函数名，用于标记这个函数的标识符。
- 参数，可以为任意的变量类型，若数组作为参数，那么只传递数组的首地址。
- 返回值类型，表示为函数的返回值情况，如果没有返回值则为 void，如果为数组，则返回数组的首地址。
- 函数体，表示了此函数需要执行的内容。

我们将根据时间情况，尝试实现函数的多态性功能，以及函数的重载功能，以此加入 C++ 语言的部分特性。

(三) 语句

基本语句包括

- 表达式，包括条件表达式，运算表达式等，此部分可以详见本章的表达式部分。
- 变量及函数声明语句，用于声明函数的结构和变量的存在。
- if 语句，其中包括 if-else 嵌套结构，用于进行条件分支。
- while 与 for 语句，用于构造循环结构。
- break 语句，用于直接跳出循环。
- continue 语句，用于跳过此循环部分，进入下一次循环。
- return 语句，用于标识函数的返回值，同时结束本函数的运行。
- 空语句，不执行任何内容。

特殊语句包括语句块，语句块中局部变量的生命周期只在本语句块执行时间内。同时我们也将视时间情况，对 switch 语句进行实现。

(四) 表达式

对于表达式，我们将实现如下运算符。

- 算术运算符 $+, -, *, /, \%$
- 关系运算符 $==, !=, >, <, <=, >=$

- 位运算符 $\&, |, \hat{}, \sim$
- 逻辑运算符 $!, \&\&, ||$
- 赋值运算符 $=$
- 解引用运算符 $*$
- 成员选择运算符 $., \rightarrow$
- 取地址运算符 $\&$
- 一元正负号运算符 $-, +$
- 数组下标 []
- 函数调用运算符 ()
- 逗号 ,

运算符结合律和优先级如表1所示。

三、 语言的 Backus 语言范式及文法

SysY 语言的文法采用扩展的 Backus 范式（EBNF，Extended Backus-Naur Form）表示，其中：

- 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
- 终结符或者是由单引号括起的串，或者是 Ident、InstConst 这样的记号

具体的定义文法表示将在附录里呈现，此处暂不赘述。

1. 语言的终结符特征

1. 语言中标识符 **Ident** 的规范如下：

$$id \rightarrow id\_non\_digit \mid id\ non\_digit \mid id\ digit$$

其中 id_non_digit 为大写、小写字母或下划线，id_digit 为数字。注意我们的语言中，变量可以在不同的作用域内重名，而且函数和变量名重复是没有关系的。

2. 语言中注释的规范如下：
   - 单行注释 //：以此符号为首的所有字符都被编译器忽略，但不包括换行；
   - 多行注释/* */：必须配对，被它们包含的所有字符都被编译器忽略。
3. 语言中关于数值常量的规范如下：
   - $integer\_const \rightarrow decimal\_const \mid octal\_const \mid hexadecimal\_const$
   - $decimal\_const \rightarrow nonzero\_digit \mid decimal\_const\ digit \mid digit`.'decimal\_const$
   - $oct\_const \rightarrow 0 \mid oct\_constoct\_digit$
   - $hex\_const \rightarrow hex\_prefix \mid hex\_consthex\_digit$ 且 $hex\_prefix \rightarrow' 0x' \mid' 0X'$

其中 $digit$ 为数字 0-9，oct_digit 为数字 0-7，hex_digit 为数字 0-9 和字符 [A-F] | [a-f]。

运算符	结合律	描述
[]	左	数组下标
()	左	函数调用
.	左	成员选择
→	左	成员选择
~	右	位求反
!	右	逻辑非
−	右	一元负号
+	右	一元正号
*	右	解引用
&	右	取地址
*	左	乘法
/	左	除法
%	左	取模
+	左	加法
−	左	减法
<	左	小于
>	左	大于
<=	左	小于等于
>=	左	大于等于
==	左	等于
!=	左	不等于
&	左	位与
^	左	位异或
|	左	位或
&&	左	逻辑与
||	左	逻辑或
=	右	赋值
,	左	逗号

表 1: 运算符优先级和结合律（同一横线内的运算符同级）

四、 SysY 程序及其 ARM 汇编代码编写

在此小节，我们将设计几个简单地 SysY 程序，并编写等价的 ARM 汇编程序，并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。

(一) 递归计算阶乘的函数 Factorial

我们设计的第一个 SysY 程序就是用来计算一个数的阶乘的。该程序事实上存在两个部分，第一个部分是主函数 main，主要调用了库函数 printf 来帮助输出调试结果信息；另一个部分为函数 factorial，两者分别通过 gcc-arm 编译生成目标 obj 文件，随后通过 ld 指令生成 qemu-arm 可以运行的 ARM 二进制文件。

其中 main.c 函数的定义如下：

main.c

```c
#include <stdio.h>
extern int factorial(int number); // 链接时寻找该函数即可。
int main() {
    int input;
    scanf("\%d", &input);
    int res = factorial(input);
    printf("\%d", res);
    return 0;
}
```

我们使用 gcc-arm 对其编译，输出 main.o：

```
gcc-arm -c -o main.o main.s
```

接下来，我们需要手写 ARM 汇编版本的阶乘计算程序，而且由于该函数用到了递归函数，因此我们更需要小心分析函数的工作流程。我们先给出高级语言版本的 factorial 函数内容：

高级语言版本的 factorial 程序

```c
int factorial(int number) {
    if (number == 1 || number == 0) {
        return 1;
    } else {
        return number * factorial(number - 1);
    }
}
```

我们先从最简单的分支判断条件来进行分析。首先此处只是一个 if-else 分支，但是其判断条件和入栈的参数 number 有关。查询 ARM 汇编手册可知，对于入栈的参数，我们可以利用寄存器 r0 来进行读取；而对于 else 分支，number 的值会发生修改，因此还需要额外做一步备份。根据以上分支我们不难写出如下的 ARM 汇编代码：

上述 if-else 对应的 ARM 汇编指令

```
@ if else语句
ldr r3, [fp, #-8]
cmp r3, #1
beq branch1
ldr r3, [fp, #-8]
cmp r3, #0
```

```
bne branch2

@ branch1 对应的是number == 1 || number == 0
@ -> return 1 (r3)
@ factorial不允许随便跳转到branch1 :))
branch1:

@ branch2 对应else的情况
branch2:

@ end 对应函数返回
end:
```

接下来我们具体分析每个 branch 都在做些什么。

首先是最简单的递归出口的情况，我们只需要将 r3 设置成 1 然后跳转到 end 的地方即可。即

branch1

```
@ if number == 1 || number == 0
branch1:
    mov r3, #1
    b end
```

其次，我们需要考虑调用递归的第二个分支的情况。由于此处我们使用到 r3 寄存器用来存储返回值，那么假设上一层递归函数已经正确设置好 r3 的值了，我们只需要获取 r3 寄存器的值，并重写即可。即首先我们要从备份的栈上取出 number 的值，减去 1 之后调用 factorial 函数，然后获取 r3 的值并进行重写。

branch2

```
branch2:
    ldr r3, [fp, #-8]
    sub r3, r3, #1

    @ 改变一下arg0并递归调用
    mov r0, r3
    bl factorial

    @ 计算上一个factorial返回的值和当前值的结果，然后保存
    mov r8, r0
    ldr r3, [fp, #-8] @ 上个函数已经修改了 :)
    mul r3, r8, r3 @ r3将在end处使用到
```

最后一部分便是函数的调用和栈帧切换的汇编代码处理了。我们知道，函数调用的逻辑是这样的：

- 参数从右向左（或者从左向右，根据目标平台而决定）入栈；
- 函数返回地址入栈；
- 备份栈帧指针；
- 栈指针下移，开辟局部变量空间；
- 函数处理；
- 恢复栈帧；

- 返回到存储的返回地址处。

根据以上处理逻辑我们不难写出对应的 ARM 汇编代码：

函数调用的汇编指令

```
@ 备份上一个函数的栈帧情况，并保存返回地址到栈上
push { fp, lr }

@ 因为push过一次，sp多减了一个，所以本函数的栈帧指针寄存器fp = sp + 4
add fp, sp, #4
@ 开辟两个变量的空间作返回值。
sub sp, sp, #8

@ 保存参数number，后续可能会修改，需要从栈上去取
str r0, [fp, #-8]
```

最后完整代码如下：

完整代码

```
@ 定义目标架构为 ARM 平台
.arch armv7-a
.arm

.text
@ 定义字段对齐长度为1
.align 1

@ 定义一个全局名称（函数名）
.global factorial
@ factorial是一个函数类型，而非变量
.type factorial %function
@ 文件名为factorial
.file "factorial.sy"

.syntax unified

.fpu vfpv3-d16

factorial:
    @ 函数原型是 int factorial(int number)
    @ 递归函数入口 factorial
    @ 需要利用寄存器 sp 和 bp 为函数开辟栈上空间，用来存一些临时变量等。
    @ ---------
    @ arg0      <----- fp - 8
    @ ---------
    @ ret地址
    @ ---------
    @ XXXX      <--- fp
    @ ---------

    @ 备份上一个函数的栈帧情况，并保存返回地址到栈上
    push { fp, lr }

    @ 因为push过一次，sp多减了一个，所以本函数的栈帧指针寄存器fp = sp + 4
    add fp, sp, #4
```

```
    @ 开辟两个变量的空间作返回值。
    sub sp, sp, #8

    @ 保存参数number, 后续可能会修改, 需要从栈上去取
    str r0, [fp, #-8]

    @ if else语句
    ldr r3, [fp, #-8]
    cmp r3, #1
    beq branch1
    ldr r3, [fp, #-8]
    cmp r3, #0
    bne branch2

@ branch1 对应的是number == 1 || number == 0
@ -> return 1 (r3)
@ factorial不允许随便跳转到branch1 :))
branch1:
    mov r3, #1
    b end

@ branch2 对应
branch2:
    ldr r3, [fp, #-8]
    sub r3, r3, #1

    @ 改变一下arg0并递归调用
    mov r0, r3
    bl factorial

    @ 计算上一个factorial返回的值和当前值的成绩, 然后保存
    mov r8, r0
    ldr r3, [fp, #-8] @ 上个函数已经修改了 :)
    mul r3, r8, r3 @ r3将在end处使用到

@ 函数返回
end:
    mov r0, r3
    sub sp, fp, #4
    pop { fp, lr }
    bx lr
    .size       factorial, .-factorial
```

接下来我们使用 gcc-arm 汇编这个代码，然后和 main.o 连接到一起，再使用 qemu-arm 测试结果。

```
gcc-arm -c -o factorial.o factorial.s
gcc-arm -o test.bin factorial.o main.o -static
qemu-arm test.bin
```

运行结果如下：

(二) 循环计算斐波那契数列的某项值

我们设计的第二个 SysY 程序用来计算斐波那契数列的某项的值，此程序将计算斐波那契数列计算程序直接写在了 main 函数中，调用库函数实现输出与输入，然后通过 gcc-arm 编译成目

```
(base) darren@darren-MS-7C82:~/606/chb$ gcc-arm factorial.o main.o -static -o test.bin
(base) darren@darren-MS-7C82:~/606/chb$ qemu-arm test.bin
12
result:479001600(base) darren@darren-MS-7C82:~/606/chb$
```

图 1: 阶乘程序运行结果

标文件，最后通过通过 qemu-arm 运行此二进制文件。

SysY 程序代码如下。

```
#include <stdio.h>
int main() {
    int a = 1;
    int b = 1;
    int c = 1;
    int input;
    scanf("%d", &input);
    input = input - 2; // 用于处理斐波那契数列的前2位
    while(input > 0) {
        a = b;
        b = c;
        c = a + b;
        input = input - 1;
    }
    printf("result:%d", input);
}
```

此处为了加快程序运行速度，我们将程序中的局部变量直接放入了寄存器中，同时，因为主函数是一个函数结构，所以我们需要完成保存 lr 寄存器，恢复栈的操作，转换成 ARM 汇编代码如下。

```
.arch armv7-a
.arm

.global main
.type main, %function
.text
main:
    @ 保存lr和fp
    push {fp, lr}

    @ 调用scanf获取用户输入
    ldr r0, =format
    sub sp, sp, #4
    mov r1, sp
    bl  scanf
    ldr r3, [sp, #0]
    add sp, sp, #4

    @ 准备局部变量
    sub r3, r3, #2
    mov r0, #1
    mov r1, #1
    mov r2, #1

LOOP:
```

```
26          cmp r3 , #0
27          ble END
28
29          @ 计算
30          mov r0, r1
31          mov r1, r2
32          add r2, r0, r1
33
34          @ 计数
35          sub r3, r3, #1
36          b    LOOP
37      END:
38          @ 打印结果
39          ldr r0, =output
40          mov r1, r2
41          bl   printf
42
43          @ 恢复栈和lr
44          pop {fp, lr}
45          bx lr
46
47
48      .data
49      format:
50          .asciz "%d"
51
52      output:
53          .asciz "result: %d \n"
```

我们通过如下指令编译并运行此汇编代码。

```
gcc-arm -static fab.S -o fab
qemu-arm ./fab
```

最终我们得到如下结果。

```
(base) darren@darren-MS-7C82:~/compiler$ gcc-arm -static fab.S -o fab
(base) darren@darren-MS-7C82:~/compiler$ qemu-arm ./fab
10
result: 55
```

图 2: 斐波那契数列程序运行结果

A 分工情况

- 陈豪斌：负责了 ARM 汇编的阶乘代码设计；负责了上下文无关文法的设计 [1]；
- 许家威：负责了 ARM 汇编的 Fibonacci 代码设计；负责了本小组语言支持的语法特性定义和语言特性介绍。

B 文法定义

此小节将介绍我们设计的语言的上下文无关文法（Context-Free Grammar）的具体定义，如下所示，其中 CompUnit 是起始符号，编译单元。

编译单元	CompUnit → [CompUnit](Decl | FuncDef | StructDef)
声明	Decl → ConstDecl | VarDecl | StructDecl
常量声明	ConstDecl → **‘const’** BType ConstDef{‘,’ConstDef}‘;’
基本类型	BType → (**‘int’** | **‘char’** | **‘float’** | **‘double’**)[‘*’]
常数定义	ConstDef → ItemType {‘[’ ConstExp ‘]’}‘=’ConstInitVal
常量初值	ConstInitVal → ConstExp | ‘{’[ConstInitVal{‘,’ConstInitVal}]‘}’
变量声明	VarDecl → BType VarDef{‘,’ VarDef}‘;’
变量定义	VarDef → Ident{‘[’ConstExp‘]’} | Ident‘[’{ConstExp}‘]’‘=’InitVal
变量初值	InitVal → Exp | ‘{’[InitVal{‘,’InitVal}]‘}’
函数定义	FuncDef → FuncType Ident‘(’[FuncFParams]‘)’ Block
函数类型	FuncType → **‘void’** | **‘int’** | **‘char’** | **‘float’** | **‘double’**
函数形参表	FuncFParams → FuncFParam{‘,’FuncFParam}
函数形参	FuncFParam → BType ItemType[‘[’‘]’{‘[’Exp‘]’}]
结构体声明	StructDecl → **‘struct’** ItemType ‘;’
结构体定义	StructDef → **‘struct’** ItemType ‘{’CompUnit‘};’
语句块	Block → ‘{’{BlockItem}‘}’
语句块项	BlockItem → Decl | Stmt
语句	Stmt → LVal‘=’Exp‘;’ | [Exp]‘;’ | Block | **‘if’**‘(’Cond‘)’Stmt[**‘else’**Stmt] | **‘while’**‘(’Cond‘)’Stmt | **‘break’**‘;’ | **‘continue’**‘;’ | **‘return’**[Exp]‘;’ | **‘for’**‘(’[Exp]‘;’Cond‘;’Exp‘)’ Stmt
表达式	Exp → BitExp
条件表达式	Cond → LOrExp
左值表达式	LVal → ItemType{‘[’Exp‘]’
基本表达式	PrimaryExp → ‘(’Exp‘)’ | LVal | Number
数值	Number → **IntConst** | **CharConst** | **FloatConst** | **DoubleConst**

一元表达式	UnaryExp → PrimaryExp | ItemType‘(’[FuncRParams]‘)’ | UnaryOp UnaryExp
单目运算符	UnaryOp → ‘+’ | ‘-’ | ‘!’ | ‘∼’ | ‘&’ | ‘*’
函数实参表	FuncRParams → Exp{‘,’Exp}
乘除模表达式	MulExp → UnaryExp | MulExp(‘*’ | ‘/’ | ‘%’)UnaryExp
加减表达式	AddExp → MulExp | AddExp(‘+’ | ‘-’) MulExp
位运算表达式	BitExp → AddExp | BitExp (‘&’ | ‘|’ | ‘^’) AddExp
关系表达式	RelExp → AddExp | RelExp(‘<’ | ‘>’ | ‘>=’ | ‘<=’) AddExp
相等性表达式	EqExp → RelExp | EqExp(‘==’ | ‘!=’) RelExp
逻辑与表达式	LAndExp → EqExp | LAndExp‘&&’EqExp
逻辑或表达式	LOrExp → EqExp | LOrExp‘||’LAndExp
常量表达式	ConstExp → AddExp
基本单元	ItemType → **Ident**‘.’ItemType | **Ident**‘→’ItemType | **Ident**

C GitLab 工程链接

https://gitlab.com/c1934/compiler-homework/-/tree/main/Homework_ARM

参考文献

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.