



南開大學
Nankai University

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

组员：朱景博 米奕霖
学号：2111451 2111566
专业：计算机科学与技术

2023 年 10 月 10 日

摘要

本文参考 miniSysY 定义了支持变量声明、赋值语句、循环语句及分支语句、函数定义的 CFG，并在此 CFG 的基础上设计了尽量覆盖上述 SysY 功能的程序，且手动编写其 ARM 汇编程序并调试通过，运行并得到正确的结果。

关键字：CFG、SysY、ARM 汇编

目录

1 背景介绍	2
1.1 SysY 语言	2
1.2 上下文无关文法	2
1.2.1 基本概念	2
1.2.2 语法规则	2
1.2.3 应用领域	3
2 CFG 相关设计	3
2.1 CFG 及相关注解	3
3 ARM 汇编	4
3.1 斐波那契数列	4
3.2 阶乘	10
4 小组分工	12
4.1 CFG 设计部分	12
4.2 ARM 编程部分	12
5 思考与总结	13

1 背景介绍

1.1 SysY 语言

SysY 语言是一个用于教育和研究的 C 语言子集。它的设计旨在简化 C 语言的复杂性，使之更易于学习和理解。SysY 语言具有 C 语言的基本语法结构，但去掉了一些复杂和容易引起错误的特性，同时添加了一些新的特性，以提高代码的可读性和可维护性。

SysY 语言最初由中国清华大学的计算机系开发，旨在用于计算机体系结构与操作系统课程的教学。它被设计为一种简单而强大的语言，使得学生可以更容易地理解计算机程序的基本概念，同时也方便了教师进行教学和学生进行学习。

SysY 语言的语法和语义与 C 语言非常相似，但做了一些简化和限制，以降低学习门槛。它支持整数和浮点数类型，具有基本的控制结构（例如循环和条件语句），并且可以进行函数的定义和调用。与 C 语言不同的是，SysY 语言没有指针和数组的概念，这样可以避免一些常见的编程错误。此外，SysY 语言还支持多维数组和结构体，以便更方便地组织和处理数据。

总的来说，SysY 语言是一个旨在帮助学生更好地理解计算机编程基本概念的教学工具。它的简单性和易学性使得它成为初学者学习编程的良好选择。

1.2 上下文无关文法

上下文无关文法（Context-Free Grammar，简称 CFG）是一种形式文法，用于描述一类形式语言的语法结构。这种文法在计算机科学和语言学领域中被广泛使用，用于描述编程语言的语法、自然语言的语法结构等。

1.2.1 基本概念

CFG 由四个元素组成：

- ** 非终结符 (Non-terminal symbols): ** 非终结符是语法规则中的符号，它们代表语法结构的各个组成部分。非终结符通常用大写字母表示，如 S、A、B 等。
- ** 终结符 (Terminal symbols): ** 终结符是最终出现在字符串中的符号，也就是语言的基本元素。终结符可以是字母、数字、标点符号等，它们是语言中的实际字符。
- ** 产生式 (Production rules): ** 产生式定义了非终结符如何被替换为终结符和/或其他非终结符的规则。一个产生式表示了一种替换规则，形式通常为 $A \rightarrow$ ，其中 A 是非终结符，是由终结符和/或非终结符组成的字符串。
- ** 开始符号 (Start symbol): ** 开始符号是文法中的一个特殊非终结符，表示语言的起始点。文法的推导 (Derivation) 从开始符号开始，通过不断地应用产生式，最终生成符合语法规则的字符串。

1.2.2 语法规则

上下文无关文法的语法规则非常简洁，每个产生式都表达了一种替换关系，例如：

- $S \rightarrow \text{if } E \text{ then } S \text{ else } S$
- $E \rightarrow \text{id} + E$
- $E \rightarrow \text{id}$

这些规则描述了一个简单的条件语句和带加法的表达式的语法结构。

1.2.3 应用领域

- ** 编程语言设计: ** 编程语言的语法结构通常通过上下文无关文法描述, 编译器和解释器使用这些语法规则来解析源代码。

- ** 自然语言处理: ** 上下文无关文法被用来描述自然语言中的句法结构, 用于句法分析和语法树的构建。

- ** 数据格式描述: ** 在数据交换和存储中, 上下文无关文法被用来描述数据格式, 例如 XML、JSON 等。

总之, 上下文无关文法是一种非常重要的形式化工具, 用于描述和分析各种类型的语言结构, 从编程语言到自然语言, 都可以使用上下文无关文法进行精确的描述。

2 CFG 相关设计

设计过程中主要参考了 miniSysY 的文法结构, 并在此基础上添加了对浮点数、字符型的定义及相关操作, 并对终结符的定义进行了简化, 具体如下, 将给出对应的 CFG 并逐行解释。

2.1 CFG 及相关注解

CompUnit -> [CompUnit] (Decl | FuncDef) // 编译单元, 可以包含声明或函数定义, 可以重复出现

Decl -> ConstDecl | VarDecl | FloatVarDecl | CharVarDecl // 声明, 可以是常量声明、整型变量声明、浮点型变量声明或字符型变量声明

ConstDecl -> 'const' BType ConstDef ';' ConstDef ';' // 常量声明, 关键字'const' 后跟基本类型和常量定义, 可以包含多个常量定义, 以逗号分隔, 末尾有分号

BType -> 'int' | 'float' | 'char' // 基本类型, 可以是整型、浮点型或字符型

ConstDef -> Ident '[' ConstExp ']' '=' ConstInitVal // 常量定义, 包括标识符和可选的数组维度, 等号后是常量初始化值

ConstInitVal -> ConstExp | '[' ConstInitVal ',' ConstInitVal ']' // 常量初始化值, 可以是常量表达式或花括号包裹的常量初始化列表

VarDecl -> BType VarDef ';' VarDef ';' // 变量声明, 关键字后跟变量定义, 可以包含多个变量定义, 以逗号分隔, 末尾有分号

FloatVarDecl -> 'float' FloatVarDef ';' FloatVarDef ';' // 浮点型变量声明, 关键字'float' 后跟浮点型变量定义, 可以包含多个变量定义, 以逗号分隔, 末尾有分号

FloatVarDef -> Ident '[' ConstExp ']' | Ident '[' ConstExp ']' '=' FloatInitVal // 浮点型变量定义, 包括标识符和可选的数组维度, 等号后是浮点型变量初始化值

FloatInitVal -> FloatExp | '[' FloatInitVal ',' FloatInitVal ']' // 浮点型变量初始化值, 可以是浮点数表达式或花括号包裹的浮点型变量初始化列表

CharVarDecl -> 'char' CharVarDef ';' CharVarDef ';' // 字符型变量声明, 关键字'char' 后跟字符型变量定义, 可以包含多个变量定义, 以逗号分隔, 末尾有分号

CharVarDef -> Ident '[' ConstExp ']' | Ident '[' ConstExp ']' '=' CharInitVal // 字符型变量定义, 包括标识符和可选的数组维度, 等号后是字符型变量初始化值

CharInitVal -> CharExp | '[' CharInitVal ',' CharInitVal ']' // 字符型变量初始化值, 可以是字符表达式或花括号包裹的字符型变量初始化列表

FuncDef -> FuncType Ident '(' [FuncFParams] ')' Block // 函数定义, 包括函数类型、标识符、参数列表和函数体

FuncType -> 'void' | 'int' | 'float' | 'char' // 函数类型, 可以是空类型、整型、浮点型或字符型

FuncFParams -> FuncFParam ',' FuncFParam // 函数形式参数列表, 可以包含一个或多个函数形式参数

FuncFParam -> BType Ident '[' ']' '[' Exp ']' // 函数形式参数, 包括基本类型和标识符, 可以有可选的数组维度

Block -> " BlockItem " // 代码块, 包括花括号内的多个语句或声明

BlockItem -> Decl | Stmt // 代码块中的元素, 可以是声明或语句

Stmt -> LVal '=' Exp ';' | [Exp] ';' | Block | 'if' '(' Cond ')' Stmt ['else' Stmt] | 'while' '(' Cond ')' Stmt | 'break' ';' | 'continue' ';' | 'return' [Exp] ';' // 语句, 包括赋值语句、空语句、代码块、条件语句、循环语句、中断语句、继续语句和返回语句

Exp -> AddExp // 表达式, 包括加法表达式

Cond -> LOrExp // 条件表达式, 包括逻辑或表达式

LVal -> Ident '[' Exp ']' // 左值, 包括标识符和可选的数组索引

PrimaryExp -> '(' Exp ')' | LVal | Number | FloatNumber | CharLiteral // 基本表达式, 可以是括号内的表达式、左值、整数、浮点数或字符字面量

UnaryExp -> PrimaryExp | Ident '(' [FuncRParams] ')' | UnaryOp UnaryExp // 一元表达式, 可以是基本表达式、函数调用或一元操作符应用的结果

UnaryOp -> '+' | '-' | '!' // 一元操作符, 可以是正号、负号或逻辑非

FuncRParams -> Exp ',' Exp // 函数实际参数列表, 可以包含一个或多个表达式

MulExp -> UnaryExp | MulExp ('*' | '/' | '//' 乘法表达式, 包括一元表达式和乘法运算

AddExp -> MulExp | AddExp ('+' | '-') MulExp // 加法表达式, 包括乘法表达式和加法运算

RelExp -> AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp // 关系表达式, 包括加法表达式和比较运算

EqExp -> RelExp | EqExp ('==' | '!=') RelExp // 相等性表达式, 包括关系表达式和相等性比较

LAndExp -> EqExp | LAndExp " EqExp // 逻辑与表达式, 包括相等性表达式和逻辑与运算

LOrExp -> LAndExp | LOrExp '||' LAndExp // 逻辑或表达式, 包括逻辑与表达式和逻辑或运算

ConstExp -> AddExp // 常量表达式, 包括加法表达式

FloatExp -> AddExp " Digit Digit // 浮点数表达式, 包括整数部分和小数部分

CharExp -> CharLiteral // 字符常量表达式, 包括字符字面量

Digit -> '0' | '1' | ... | '9' // 数字, 可以是 0 到 9 的任意数字

CharLiteral -> " (任意字符但不包括') " // 字符字面量, 单引号括起来的任意字符

FloatNumber -> Digit Digit " Digit Digit // 浮点数, 包括整数部分、小数点和小数部分

3 ARM 汇编

3.1 斐波那契数列

example.c 文件

```
1 #include <stdio.h>
```

```

2
3 int main() {
4     int a, b, i, t, n;
5     a = 0;
6     b = 1;
7     i = 1;
8
9     scanf("%d", &n);
10
11    printf("%d\n", a);
12    printf("%d\n", b);
13
14    while (i < n) {
15        t = b;
16        b = a + b;
17        printf("%d\n", b);
18        a = t;
19        i = i + 1;
20    }
21
22    return 0;
23 }

```

对于本程序，主要运用到了 scanf, printf, 以及 while 循环函数，下面给出带详细注释的 arm 代码

example.S 文件

```

1 .arch armv7-a
2
3 @ .comm section save global variable without initialization
4 .comm n, 4 @ global variables
5 .comm a, 4
6 .comm b, 4
7 .comm i, 4
8 .comm t, 4
9
10 .section .data
11 a:
12     .int 0
13 b:
14     .int 1
15 t:
16     .int 1
17
18 @ .rodata section save constant
19 .section .rodata
20 .align 2
21 __str1:
22 .ascii "%d"
23 .align 2

```

```

24 _str2:
25 .ascii "%d\n"
26
27 .text
28 .global main
29 .type main, %function
30 main:
31     push {fp, lr}
32     add fp, sp, #4
33
34
35     ldr r0, __bridge+20 @ *r0="%d"
36     ldr r1, __bridge    @ r1=&n
37     bl scanf            @ scanf("%d", &n)
38
39     ldr r0, __bridge    @ r0=&n
40     ldr r4, [r0]        @ r4=n
41
42     ldr r0, __bridge+4 @r0=&a
43     ldr r1,[r0]         @r1=a
44     ldr r0, __bridge+24 @*r0="%d\n"
45     bl printf          @printf("%d\n", a)
46
47     ldr r0, __bridge+8 @r0=&b
48     ldr r1,[r0]         @r1=b
49     ldr r0, __bridge+24 @*r0="%d\n"
50     bl printf          @printf("%d\n", b)
51
52 ll:
53     ldr r0, __bridge+4 @ r0=&a
54     ldr r1, [r0]       @ r1=a
55     ldr r0, __bridge+8 @ r0=&b
56     ldr r2, [r0]       @ r2=t=b(我省略了t的使用)
57     add r3, r1, r2     @ r3=r1+r2=a+b
58     ldr r0, __bridge+8 @ r0=&b
59     str r3, [r0]       @ b=a+b
60     ldr r0, __bridge+4 @ r0=&a
61     str r2, [r0]       @ a=t
62
63     ldr r0, __bridge+8 @r0=&b
64     ldr r1,[r0]        @r1=b
65     ldr r0, __bridge+24 @*r0="%d\n"
66     bl printf          @printf("%d\n", b)
67
68     ldr r0, __bridge+12 @ r0=&i
69     ldr r1, [r0]       @ r1=i
70     add r3, r1, #1     @ r3=i+1
71     ldr r0, __bridge+12 @ r0=&i
72     str r3, [r0]       @ i=r3=i+1

```

```

73      cmp r3,r4      @while(i<n) 循环
74      blt l1
75
76
77 l2:
78     pop {fp, pc} @ return 0
79
80 __bridge:
81     .word n
82     .word a
83     .word b
84     .word i
85     .word t
86     .word __str1
87     .word __str2

```

对于本段 arm 的汇编代码，主要讲解几部分。

(1) 变量部分的声明

```

1
2
3@ .comm section save global variable without initialization
4.comm n, 4 @ global variables
5.comm a, 4
6.comm b, 4
7.comm i, 4
8.comm t, 4
9
10.section .data
11a:
12     .int 0
13b:
14     .int 1
15t:
16     .int 1
17
18@ .rodata section save constant
19.section .rodata
20.align 2
21__str1:
22.ascii "%d"
23.align 2
24__str2:
25.ascii "%d\n"
26

```

定义数据区：.comm n, 4 定义了一个全局变量 n，大小为 4 字节。后续变量大同小异。

data 段：给数据初始化，对应 c 代码中 a = 0; b = 1; i = 1; 部分。

定义常量字符串：“%d” 等用于在后续的 scanf 和 printf 函数中使用。


```

3  _bridge:
4      .word n
5      .word a
5      .word b
7      .word i
3      .word t
3      .word _str1
3      .word _str2

```

此处定义 `_bridge` 的符号表项分别表示上述声明的各种变量，如 `a,b,_str1` 等。

(2) `scanf` 和 `printf` 的使用

```

ldr r0, _bridge+20 @ *r0="%d"
ldr r1, _bridge    @ r1=&n
bl scanf           @ scanf("%d", &n)

ldr r0, _bridge    @ r0=&n
ldr r4, [r0]       @ r4=n

ldr r0, _bridge+4  @r0=&a
ldr r1,[r0]        @r1=a
ldr r0, _bridge+24 @*r0="%d\n"
bl printf          @printf("%d\n", a)

```

此处通过 `r0,r1` 的设置参数，达到对于 `scanf` 和 `printf` 调用时参数的传入。例如先将 `"%d"` 传入 `r1`，将 `&n` 传入 `r0`，达到了 `scanf("%d", &n)` 的使用

(3) 循环计算输出斐波那契数列

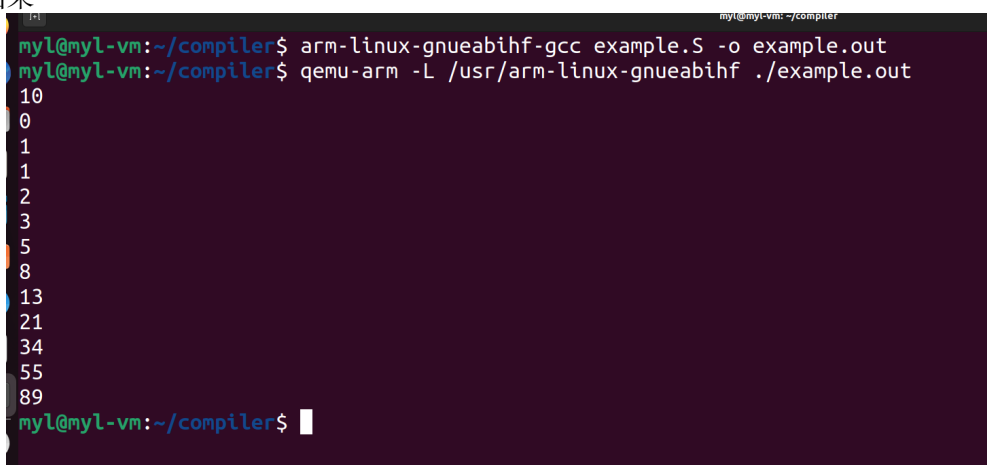
```

l1:
;    ldr r0, _bridge+4      @ r0=&a
;    ldr r1, [r0]           @ r1=a
;    ldr r0, _bridge+8      @ r0=&b
;    ldr r2, [r0]           @ r2=t=b(我省略了t的使用)
;    add r3, r1, r2         @ r3=r1+r2=a+b
;    ldr r0, _bridge+8      @ r0=&b
;    str r3, [r0]           @ b=a+b
;    ldr r0, _bridge+4      @ r0=&a
;    str r2, [r0]           @ a=t
;
;    ldr r0, _bridge+8      @r0=&b
;    ldr r1,[r0]            @r1=b
;    ldr r0, _bridge+24     @*r0="%d\n"
;    bl printf              @printf("%d\n", b)
;
;    ldr r0, _bridge+12     @ r0=&i
;    ldr r1, [r0]           @ r1=i
;    add r3, r1, #1         @ r3=i+1
;    ldr r0, _bridge+12     @ r0=&i
;    str r3, [r0]           @ i=r3=i+1
;
;    cmp r3,r4              @while(i<n) 循环
;    blt l1
;
l2:
;    pop {fp, pc} @ return 0
;

```

此处通过对于 l1 的代码段定义了循环过程，将 a 和 b 的值传入 r1, r2, 并进行加法计算，printf 打印输出。然后通过 str 将数值存储。整体取数的思路就是先获得引用，ldr r0, _bridge+4 @r1=a 如 r0=&b, 后对于引用进行操作，如对数值进行计算或存取 ldr r1,[r0] 或 str r1,[r0]。

再通过 cmp 比较和 blt 实现 while(i<n) 的分支跳转。如果 i<n 则进行 l2, 即退出程序。(4) 测试结果



```

myl@myl-vm: ~/compiler
myl@myl-vm:~/compiler$ arm-linux-gnueabi-gcc example.S -o example.out
myl@myl-vm:~/compiler$ qemu-arm -L /usr/arm-linux-gnueabi ./example.out
10
0
1
1
2
3
5
8
13
21
34
55
89
myl@myl-vm:~/compiler$

```

通过 arm-linux-gnueabi-gcc example.S -o example.out 和 qemu-arm -L /usr/arm-linux-gnueabi ./example.out 来编译测试代码，此处为输出前十位斐波那契数列。(由于源程序中有输出 a 和 b, 所

以此处输出了十二个数字)。

3.2 阶乘

此处不进行整段代码的粘贴，只进行部分重要代码的讲解。

(1) 下面为 c 代码，一个简单的实现阶乘的程序。

```
1 #include <stdio.h>
2
3 int main() {
4     int i,n,f;
5     i = 2;
6     f = 1;
7
8     scanf("%d", &n);
9
10
11     while (i <= n) {
12         f = f*i;
13         i = i + 1;
14     }
15     printf("%d\n", f);
16     return 0;
17 }
```

(2) 此处为定义 __bridge 的符号表项分别表示上述声明的各种变量，如 i,n,_str1 等。

```
6  
7 _bridge:  
8     .word i  
9     .word n  
0     .word f  
1     .word _str1  
2     .word _str2
```

(3) 此处为核心计算阶乘的部分, 通过对 i, f 的读取, 之后通过 MUL r2, r1, r2 完成乘法, add r2, r1, #1 完成 i 的自增, 通过 cmp r2,r3 和 ble l1 完成 while(i<=n) 的循环以及分支跳转, l2 部分则为打印最终结果以及程序退出部分。

```

0
9 l1:
0    ldr r0, _bridge      @ r0=&i
1    ldr r1, [r0]         @ r1=i
2    ldr r0, _bridge+8    @ r0=&f
3    ldr r2, [r0]         @ r2=f
4    MUL r2, r1, r2       @ r2=r1*r2
5
6    ldr r0, _bridge+8    @ r0=&f
7    str r2, [r0]         @ f=f*i
8
9
0    ldr r0, _bridge      @ r0=&i
1    ldr r1, [r0]         @ r1=i
2    add r2, r1, #1       @ r2=i+1
3    ldr r0, _bridge      @ r0=&i
4    str r2, [r0]         @ i=r2=i+1
5
6    cmp r2,r3             @while(i<=n) 循环
7    ble l1
8
9 l2:
0    ldr r0, _bridge+8    @r0=&f
1    ldr r1,[r0]          @r1=f
2    ldr r0, _bridge+16   @*r0="%d\n"
3    bl printf            @printf("%d\n", f)
4    pop {fp, pc} @ return 0
5
~

```

(4) 测试结果

```

myl@myl-vm:~/compiler$ arm-linux-gnueabi-gcc example3.S -o example3.out
myl@myl-vm:~/compiler$ qemu-arm -L /usr/arm-linux-gnueabihf ./example3.out
10
3628800
myl@myl-vm:~/compiler$ █

```

通过 `arm-linux-gnueabi-gcc example.S -o example3.out` 和 `qemu-arm -L /usr/arm-linux-gnueabihf ./example3.out` 来编译测试代码，并完成了阶乘操作，对于 10 的阶乘 3628800 完成了正确的输出。

4 小组分工

4.1 CFG 设计部分

米奕霖负责的是整型、浮点型常量、变量的定义及 NUMBER 等终结符的定义。
朱景博负责的是函数定义、加减乘除等表达式的定义、比较及逻辑表达式定义。

4.2 ARM 编程部分

米奕霖负责的是斐波那契数列部分的 arm 汇编代码编写即测试。

朱景博负责的是阶乘部分的 arm 汇编代码编写即测试。

5 思考与总结

在实验过程中，我们深入研究了 SysY 语言的语法和语义，为编译器设计提供了形式化的基础。通过使用上下文无关文法来描述 SysY 语言的语法结构，我们能够准确地捕捉源代码的结构，并使编译器能够正确地理解和处理它们。在 arm 编程过程中，我们更好的对于 arm 汇编语言进行了理解与运用。总之，这个实验为我们提供了宝贵的编译器设计和实现经验，我们深入了解了编译器的内部工作原理，并成功地将这些知识应用于实际项目中。这个经验将对我们未来的计算机科学研究和编程工作产生积极的影响。