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

预备工作二

小组成员：李潇逸 2111454 伍泓铮 2111885

年级：2021 级

专业：信息安全、法学

指导教师：王刚

2023 年 10 月 10 日

摘要

本编译原理实验报告旨在设计一个支持 SysY 语言特性的编译器，并生成对应的 ARM 汇编程序。报告包含实验目的、实验内容和汇编程序的定义。实验步骤包括定义编译器、编写上下文无关文法和代码测试。SysY 语言特性包括数据类型、运算符、控制流语句等。编译器程序的设计包括词法分析、语法分析、语义分析和代码生成等步骤。

关键字：编译器；SysY 语言；汇编程序

目录

一、 实验目的

第一部分，确定本小组实现的编译器支持哪些 SysY 语言特性，给出其形式化定义，并用上下文无关文法描述它的子集。

第二部分，设计几个 SysY 程序，编写等价的 ARM 汇编程序，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。这些程序应该尽可能全面地包含支持的语言特性。

question：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是怎样的?

answer：同预备工作 1，按照词法分析、语法分析、语义分析、中间代码生成、代码优化、目标代码生成、汇编和链接的步骤即可。

二、 实验内容

(一) 定义编译器

1. SysY 语言及其特性

SysY 是一种类 C 语言的静态类型语言，通常用于教学和学术研究。下面是 SysY 语言的详细定义及其特性：

1. 数据类型：
- 整数类型（int）：用于表示整数值。
- 布尔类型（bool）：用于表示真（true）或假（false）值。
2. 变量声明：
- 可以使用关键字 ‘int‘或 ‘bool‘声明变量，并指定其类型。
- 变量名必须以字母或下划线开头，可以包含字母、数字和下划线。
3. 表达式：
- 支持基本的算术运算符：加法 ‘+‘、减法 ‘-‘、乘法 ‘*‘、除法 ‘/‘。
- 支持比较运算符：等于 ‘==‘、不等于 ‘!=‘、大于 ‘>‘、小于 ‘<‘、大于等于 ‘>=‘、小于等于 ‘<=‘。
- 支持逻辑运算符：与、或、非。
4. 控制流语句：
- 条件语句（if-else）：使用关键字 ‘if‘和 ‘else‘，根据条件执行不同的代码块。
- 循环语句（while）：使用关键字 ‘while‘，当条件满足时重复执行代码块。
5. 函数定义和调用：
- 可以定义函数，包括函数名、参数列表和返回类型。
- 函数参数可以是任意合法的 SysY 数据类型。
- 函数可以有多个参数，并且可以返回一个值（整数或布尔类型）。
6. 数组：
- 支持一维数组的声明和使用。
- 数组的大小必须是一个正整数常量。
- 数组元素的访问使用索引，索引从 0 开始。

2. 本编译器支持的 SysY 语言特性

变量声明、函数声明、赋值语句、控制流语句（if、while、for）、返回语句

3. 上下文无关文法

上下文无关文法是一种用于描述程序设计语言语法的表示方式。一般来说，一个上下文无关文法（context-free grammar）由四个元素组成：

（1）一个终结符号集合 VT，它们有时也称为“词法单元”。终结符号是该文法所定义的语言的基本符号的集合。

（2）一个非终结符号集合 VN，它们有时也称为“语法变量”。每个非终结符号表示一个终结符号串的集合。

（3）一个产生式集合 P，其中每个产生式包括一个称为产生式头或左部的非终结符号，一个箭头，和一个称为产生式体或右部的由终结符号及非终结符号组成的序列。产生式主要用来表示某个语法构造的某种书写形式。如果产生式头非终结符号代表一个语法构造，那么该产生式体就代表了该构造的一种书写方式。

（4）指定一个非终结符号为开始符号 S。

因此，上下文无关文法可以通过（ VT , VN , P, S ）这个四元式定义。在描述文法时，我们将数位、符号和黑体字符串看作终结符号，将斜体字符串看作非终结符号，以同一个非终结符号为头部的多个产生式的右部可以放在一起表示，不同的右部之间用符号 | 分隔。

4. CFG 描述 SysY 语言特性

```
<program> ::= <declaration-list>

<declaration-list> ::= <declaration> <declaration-list> | <declaration>

<declaration> ::= <var-declaration> | <fun-declaration>

<var-declaration> ::= <type-specifier> <id-list> ';'

<type-specifier> ::= 'int' | 'void'

<id-list> ::= <identifier> ',' <id-list> | <identifier>

<fun-declaration> ::= <type-specifier> <identifier> '(' <params> ')' <
    compound-stmt>

<params> ::= <param-list> | 'void'

<param-list> ::= <param> ',' <param-list> | <param>

<param> ::= <type-specifier> <identifier>

<compound-stmt> ::= '{' <local-declarations> <statement-list> '}'

<local-declarations> ::= <var-declaration> <local-declarations> |

<statement-list> ::= <statement> <statement-list> |

<statement> ::= <expression-stmt> | <compound-stmt> | <selection-stmt> | <
    iteration-stmt> | <return-stmt>
```

```
28
29 <expression-stmt> ::= <expression> ';'
30
31 <selection-stmt> ::= 'if' '(' <expression> ')' <statement> 'else' <statement>
32                     | 'if' '(' <expression> ')' <statement>
33
34 <iteration-stmt> ::= 'while' '(' <expression> ')' <statement>
35                     | 'for' '(' <expression-stmt> <expression-stmt> <
                           expression> ')' <statement>
36
37 <return-stmt> ::= 'return' <expression> ';'
38                  | 'return' ';'
39
40
41 <expression> ::= <var> '=' <expression>
42                | <simple-expression>
43
44 <var> ::= <identifier> | <identifier> '[' <expression> ']'
45
46 <simple-expression> ::= <additive-expression> <relop> <additive-expression>
47                       | <additive-expression>
48
49 <relop> ::= '<' | '<=' | '>' | '>=' | '==' | '!='
50
51 <additive-expression> ::= <additive-expression> '+' <term>
52                         | <additive-expression> '-' <term>
53                         | <term>
54
55 <term> ::= <term> '*' <factor>
56          | <term> '/' <factor>
57          | <factor>
58
59 <factor> ::= '(' <expression> ')'
60            | <var>
61            | <call>
62            | <integer>
63
64 <call> ::= <identifier> '(' <args> ')'
65
66 <args> ::= <arg-list> |
67
68 <arg-list> ::= <expression> ',' <arg-list> | <expression>
69
70 <identifier> ::= <letter> <letter-or-digit>*
71 <integer> ::= <digit>+
72
73 <letter> ::= 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z'
74 <digit> ::= '0' | '1' | ... | '9'
```

```
<letter-or-digit> ::= <letter> | <digit>
```

(二) 汇编程序

1. 样例程序

c 语言:

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

void main() {
    int num = 5;
    int result = factorial(num);
    print(result);
}
```

arm 汇编:

```
.arch armv5t

.data
    num:     .word 5
    result: .word 0
    newline: .asciz "\n"

.text
    // main 标签的声明
    .global main

    // main 标签的定义
    main:
        push {lr}          // 保存返回地址
        ldr r0, =num       // 加载 num 的地址到 r0
        ldr r0, [r0]       // 加载 num 的值到 r0
        bl factorial       // 调用 factorial 函数
        ldr r2, =result    // 加载 result 的地址到 r2
        str r0, [r2]       // 将结果保存在 result 中
        ldr r0, [r2]       // 加载 result 的值到 r0
        bl print           // 调用 print 函数
        pop {pc}           // 返回到调用 main 的地址

    // 其他标签的定义
    factorial:
        push {lr}          // 保存返回地址
```

```
27          cmp r0, #0          // 判断 n 是否为 0
28          beq base_case       // 相等则跳转到 base_case
29          sub sp, sp, #4      // 为 n - 1 分配栈空间
30          str r0, [sp]        // 将参数 n 保存在栈上
31          sub r0, r0, #1      // r0 = n - 1
32          bl factorial        // 递归调用 factorial(n - 1)
33          ldr r1, [sp]        // 加载递归调用的结果到寄存器 r1
34          add sp, sp, #4      // 释放栈空间
35          mul r0, r1, r0      // r0 = r1 * r0
36          pop {pc}            // 返回到调用 factorial 的地址
37
38      base_case:
39          mov r0, #1          // n = 0 时, 返回 1
40          pop {pc}            // 返回到调用 factorial 的地址
41
42      print:
43          push {lr}           // 保存返回地址
44          mov r1, r0          // 将要打印的值保存在 r1
45          ldr r0, =newline    // 加载换行符的地址到 r0
46          bl printf           // 调用 printf 函数
47          pop {pc}            // 返回到调用 print 的地址
```

2. 代码测试

写完汇编程序（比如 example.S）后，使用下述指令即可测试它。

```
1 arm-linux-gnueabihf-gcc example.S -o example
2 qemu-arm ./example
```

结果如下图：

图 1: 测试

当然，这只是其中一个示例，更多的汇编程序的运行也是成功的。—

三、 实验总结

在完成编译原理实验后，我对编译器的工作原理和实现有了更深入的了解。通过实验，我学习了如何设计一个支持 SysY 语言特性的编译器，并生成对应的 ARM 汇编程序。在实验中，我了解了编译器的基本流程，包括词法分析、语法分析、语义分析和代码生成等步骤。同时，我也学习了如何编写上下文无关文法和进行代码测试。

在实验中，我遇到了一些困难和挑战。首先，我需要理解 SysY 语言的特性和语法规则，这需要花费一定的时间和精力。其次，我需要学习如何使用编译器工具和 ARM 汇编语言，这也需要一定的学习成本。最后，我需要编写测试代码来验证编译器的正确性，这需要一定的编程能力和耐心。

在实验中，我也收获了一些经验和技能。首先，我学会了如何使用编译器工具和 ARM 汇编语言，这对我的编程能力和职业发展都有很大的帮助。其次，我学会了如何编写上下文无关文法和进行代码测试，这对我的编程能力和软件开发能力都有很大的提升。最后，我也学会了如何团队合作和分工合作，这对我的团队协作能力和沟通能力都有很大的提升。

然而，我也意识到在实验中存在一些不足之处。首先，由于时间和资源的限制，我无法对编译器的性能和优化进行深入的研究和实践。这使得我无法完全理解和掌握编译器的优化技术和算法。其次，由于实验的范围和要求有限，我无法涉及到更多的编程语言和编译器的特性和技术。这使得我对编译器的整体理解还有待进一步的完善和深入。

为了进一步提高自己在编译原理方面的知识和能力，我计划在以后的学习和实践中继续深入研究编译器的优化技术和算法。我还计划学习更多的编程语言和编译器的特性，以扩大自己的知识面和技能。此外，我还希望能够参与更多的实际项目，通过实践来提高自己的编译器设计和实现能力。

总的来说，编译原理实验是一次非常有意义和有价值的实践活动。通过实验，我不仅学习了编译器的工作原理和实现，还提高了自己的编程能力和软件开发能力。同时，我也学会了如何解决问题和面对挑战。在实验过程中，我遇到了一些困难，但通过查阅资料、与同学讨论和请教助教的帮助，我成功地克服了这些困难。这让我意识到在面对问题时，要善于寻求帮助和与他人合作，这样才能更好地解决问题。

实验分工：

李潇逸:SysY 语言特性的选定与描述、汇编程序的撰写与测试

伍泓铮:SysY 语言特性的选定、汇编程序的撰写与测试

参考文献