



南開大學
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 12 日

摘要

本文主要介绍了本组所计划实现的 SysY 语言特性，以及预计对 SysY 语言内容，结构，功能进行的拓展，本文为需要实现的语言设计了 CFG 上下文无关文法，并用实际中的三个 SysY 程序和对应的人工翻译的 ARM 汇编代码进行举例，简单说明了本组实现的编程语言的部分翻译方法。

关键字： Compiler; ARM; SysY

This report introduces the language traits of SysY that we want to implement, as well as the extensions to the contents, structures and the functionalities of SysY. We design its corresponding context-free grammar for the language, and we also write three programs in assembly to briefly illustrate the approaches to translating the SysY programming language that we design.

Keywords: Compiler; ARM; SysY

目录

一、 编译器概述	1
(一) 编译器语言设定	1
(二) 语言结构和编译器功能概述	1
二、 编译器功能详细说明	1
(一) 变量类型	1
(二) 函数	2
(三) 语句	2
(四) 表达式	2
三、 语言的 Backus 语言范式及文法	3
1. 语言的终结符特征	3
四、 SysY 程序及其 ARM 汇编代码编写	5
(一) 递归计算阶乘的函数 Factorial	5
(二) 循环计算斐波那契数列的某项值	8
(三) 基础矩阵乘法	10
A 分工情况	13
B 文法定义	13
C GitLab 工程链接	14

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 $\&, |, \wedge, \sim$
- 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:

- $integer_const \rightarrow decimal_const \mid octal_const \mid hexadecimal_const$
- $decimal_const \rightarrow nonzero_digit \mid decimal_const \ digit \mid digit'.'decimal_const$
- $oct_const \rightarrow 0 \mid oct_const oct_digit$
- $hex_const \rightarrow hex_prefix \mid hex_const hex_digit$ 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。必须要说明的是，我们的 gcc-arm 指令已经链接到了 arm-linux-gnueabi-hf-gcc 指令上，所以下文所有的编译指令都使用 gcc-arm 进行代替。

(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

```
1 #include <stdio.h>
2 extern int factorial(int number); // 链接时寻找该函数即可。
3 int main() {
4     int input;
5     scanf("%d", &input);
6     int res = factorial(input);
7     printf("%d", res);
8     return 0;
9 }
```

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

```
1 int factorial(int number) {
2     if (number == 1 || number == 0) {
3         return 1;
4     } else {
5         return number * factorial(number - 1);
6     }
7 }
```

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

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



```

5  ldr r3, [fp, #-8]
6  cmp r3, #0
7  bne branch2
8
9  @ branch1 对应的是 number == 1 || number == 0
10 @ -> return 1 (r3)
11 @ factorial 不允许随便跳转到 branch1 :)
12 branch1:
13
14 @ branch2 对应 else 的情况
15 branch2:
16
17 @ end 对应函数返回
18 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

```

1  @ if number == 1 || number == 0
2  branch1:
3      mov r3, #1
4      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

```

1  branch2:
2      ldr r3, [fp, #-8]
3      sub r3, r3, #1
4
5      @ 改变一下 arg0 并递归调用
6      mov r0, r3
7      bl factorial
8
9      @ 计算上一个 factorial 返回的值和当前值的结果，然后保存
10     mov r8, r0
11     ldr r3, [fp, #-8] @ 上个函数已经修改了 :)
12     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

```

1  @ 备份上一个函数的栈帧情况，并保存返回地址到栈上
2  push { fp, lr }
3
4  @ 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
5  add fp, sp, #4
6  @ 开辟两个变量的空间作返回值。
7  sub sp, sp, #8
8
9  @ 保存参数number，后续可能会修改，需要从栈上去取
10 str r0, [fp, #-8]
```

最后完整代码如下：

完整代码

```

1  @ 定义目标架构为 ARM 平台
2  .arch armv7-a
3  .arm
4
5  .text
6  @ 定义字段对齐长度为1
7  .align 1
8
9  @ 定义一个全局名称（函数名）
10 .global factorial
11 @ factorial是一个函数类型，而非变量
12 .type factorial %function
13 @ 文件名为factorial
14 .file "factorial.sy"
15
16 .syntax unified
17
18 .fpu vfpv3-d16
19
20 factorial:
21     @ 函数原型是 int factorial(int number)
22     @ 递归函数入口 factorial
23     @ 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。
24     @ -----
25     @ arg0          <----- fp - 8
26     @ -----
27     @ ret地址
28     @ -----
29     @ XXXX          <--- fp
30     @ -----
31
32     @ 备份上一个函数的栈帧情况，并保存返回地址到栈上
33     push { fp, lr }
34
35     @ 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
```

```

36     add fp, sp, #4
37     @ 开辟两个变量的空间作返回值。
38     sub sp, sp, #8
39
40     @ 保存参数number, 后续可能会修改, 需要从栈上去取
41     str r0, [fp, #-8]
42
43     @ if else 语句
44     ldr r3, [fp, #-8]
45     cmp r3, #1
46     beq branch1
47     ldr r3, [fp, #-8]
48     cmp r3, #0
49     bne branch2
50
51     @ branch1 对应的是 number == 1 || number == 0
52     @ -> return 1 (r3)
53     @ factorial 不允许随便跳转到 branch1 :)
54     branch1:
55         mov r3, #1
56         b end
57
58     @ branch2 对应
59     branch2:
60         ldr r3, [fp, #-8]
61         sub r3, r3, #1
62
63         @ 改变一下 arg0 并递归调用
64         mov r0, r3
65         bl factorial
66
67         @ 计算上一个 factorial 返回的值和当前值的成绩, 然后保存
68         mov r8, r0
69         ldr r3, [fp, #-8] @ 上个函数已经修改了 :)
70         mul r3, r8, r3 @ r3 将在 end 处使用到
71
72     @ 函数返回
73     end:
74         mov r0, r3
75         sub sp, fp, #4
76         pop { fp, lr }
77         bx lr
78     .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

```

运行结果如图1:

(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。

```
1  #include <stdio.h>
2  int main() {
3      int a = 1;
4      int b = 1;
5      int c = 1;
6      int input;
7      scanf("%d", &input);
8      input = input - 2; // 用于处理斐波那契数列的前2位
9      while(input > 0) {
10         a = b;
11         b = c;
12         c = a + b;
13         input = input - 1;
14     }
15     printf("result:%d", input);
16 }
```

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。

```
1  .arch armv7-a
2  .arm
3
4  .global main
5  .type main, %function
6  .text
7  main:
8      @ 保存 lr 和 fp
9      push {fp, lr}
10
11     @ 调用 scanf 获取用户输入
12     ldr r0, =format
13     sub sp, sp, #4
14     mov r1, sp
15     bl  scanf
16     ldr r3, [sp, #0]
17     add sp, sp, #4
18
19     @ 准备局部变量
20     sub r3, r3, #2
21     mov r0, #1
22     mov r1, #1
23     mov r2, #1
24
25  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
```

最终我们得到如图2所示结果。

```

(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

(三) 基础矩阵乘法

我们通过基础的矩阵乘法算法，对语言的 for 循环，乘法运算进行演示。由于汇编语言的翻译都由 gcc-arm 完成，在此对编译的具体过程不再赘述。

编写 SysY 语言程序如下。

```

1  #include <stdio.h>
2  int a[2][2] = [1,2,3,4];
3  int b[2][2] = [2,3,4,5];
4  int main(){
5      int result;
6      for(int i = 0; i < 2; i++) {

```

```

7         for(int j = 0; j < 2; j++) {
8             result = 0;
9             for(int k = 0; k < 2; k++)
10                result += a[i][k] * b[k][j];
11            printf("%d ", result);
12        }
13        printf("\n");
14    }
15}

```

在对数组进行访问时，我们需要注意地址的移位，因为处理器是按照字进行访存的。编写的汇编代码如下。

```

1    .arch armv7-a
2    .arm
3
4    .global main
5    .type main, %function
6
7    .text
8
9    main:
10        push {r4, r5, r6, r7, r8, r9, r10, fp, lr}
11
12        @ i
13        mov r4, #0
14    LOOP_1:
15        cmp r4, #2
16        bge END_1
17
18        @ j
19        mov r5, #0
20    LOOP_2:
21        cmp r5, #2
22        bge END_2
23
24        @ result
25        mov r7, #0
26
27        @ k
28        mov r6, #0
29    LOOP_3:
30        cmp r6, #2
31        bge END_3
32
33        @ address
34        mov r10, #2
35        mul r8, r4, r10
36        add r8, r8, r6
37        mul r9, r6, r10
38        add r9, r9, r5
39
40        @ get data
41        ldr r10,=array_a
42        ldr r8, [r10, r8, lsl#2]
43        ldr r10,=array_b

```

```

44     ldr r9, [r10, r9, lsl #2]
45
46     # calculate
47     mul r8, r8, r9
48     add r7, r7, r8
49
50     add r6, r6, #1
51     b    LOOP_3
52 END_3:
53
54     ldr r0, =column
55     mov r1, r7
56     bl  printf
57
58     add r5, r5, #1
59     b    LOOP_2
60 END_2:
61
62     ldr r0, =row
63     bl  printf
64
65     add r4, r4, #1
66     b    LOOP_1
67 END_1:
68
69     pop {r4, r5, r6, r7, r8, r9, r10, fp, lr}
70     bx lr
71
72 .data
73 column:
74     .asciz "%d "
75
76 row:
77     .asciz "\n"
78
79 array_a:
80     .long 1, 2, 3, 4
81
82 array_b:
83     .long 2, 3, 4, 5

```

最终，我们得到如图3所示的运行结果。

```

(base) darren@darren-MS-7C82:~/compiler$ gcc-arm -static mat.S -o mat
(base) darren@darren-MS-7C82:~/compiler$ qemu-arm ./mat
10 13
22 29
(base) darren@darren-MS-7C82:~/compiler$ 

```

图 3: 基础矩阵乘法运行结果

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 代码和矩阵乘法代码的设计；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	$\text{CompUnit} \rightarrow [\text{CompUnit}](\text{Decl} \mid \text{FuncDef} \mid \text{StructDef})$
声明	$\text{Decl} \rightarrow \text{ConstDecl} \mid \text{VarDecl} \mid \text{StructDecl}$
常量声明	$\text{ConstDecl} \rightarrow \text{'const'} \text{ BType ConstDef}\{', \text{'ConstDef}'\};'$
基本类型	$\text{BType} \rightarrow (\text{'int'} \mid \text{'char'} \mid \text{'float'} \mid \text{'double'})[*]$
常数定义	$\text{ConstDef} \rightarrow \text{ItemType}\{[' \text{ConstExp} ']\} = \text{'ConstInitVal}'$
常量初值	$\text{ConstInitVal} \rightarrow \text{ConstExp} \mid \{ '[\text{ConstInitVal}\{', \text{'ConstInitVal}'\}] \}'$
变量声明	$\text{VarDecl} \rightarrow \text{BType VarDef}\{', \text{'VarDef}'\};'$
变量定义	$\text{VarDef} \rightarrow \text{Ident}\{[' \text{ConstExp} ']\} \mid \text{Ident}\{[' \{ \text{ConstExp} \} ']\} = \text{'InitVal}'$
变量初值	$\text{InitVal} \rightarrow \text{Exp} \mid \{ '[\text{InitVal}\{', \text{'InitVal}'\}] \}'$
函数定义	$\text{FuncDef} \rightarrow \text{FuncType Ident}('[\text{FuncFParams}])' \text{Block}$
函数类型	$\text{FuncType} \rightarrow \text{'void'} \mid \text{'int'} \mid \text{'char'} \mid \text{'float'} \mid \text{'double'}$
函数形参表	$\text{FuncFParams} \rightarrow \text{FuncFParam}\{', \text{'FuncFParam}'\}$
函数形参	$\text{FuncFParam} \rightarrow \text{BType ItemType}\{[' ']\{[' \text{Exp} ']\}\}$
结构体声明	$\text{StructDecl} \rightarrow \text{'struct'} \text{ItemType};'$
结构体定义	$\text{StructDef} \rightarrow \text{'struct'} \text{ItemType}\{ '[\text{CompUnit}];' \}$
语句块	$\text{Block} \rightarrow \{ \{ \text{BlockItem} \} \}'$
语句块项	$\text{BlockItem} \rightarrow \text{Decl} \mid \text{Stmt}$
语句	$\begin{aligned} \text{Stmt} \rightarrow & \text{LVal} = \text{'Exp'}; \mid [\text{Exp}]; \mid \text{Block} \\ & \mid \text{'if'}('(\text{Cond})' \text{Stmt} [\text{'else'} \text{Stmt}] \\ & \mid \text{'while'}('(\text{Cond})' \text{Stmt} \\ & \mid \text{'break'}; \mid \text{'continue'}; \mid \\ & \mid \text{'return'}[\text{Exp}]; \mid \\ & \mid \text{'for'}('([\text{Exp}]; \text{'Cond'}; \text{'Exp'})' \text{Stmt} \end{aligned}$
表达式	$\text{Exp} \rightarrow \text{BitExp}$
条件表达式	$\text{Cond} \rightarrow \text{LOrExp}$
左值表达式	$\text{LVal} \rightarrow \text{ItemType}\{[' \text{Exp} ']\}$
基本表达式	$\text{PrimaryExp} \rightarrow (' \text{Exp} ') \mid \text{LVal} \mid \text{Number}$
数值	$\text{Number} \rightarrow \text{IntConst} \mid \text{CharConst} \mid \text{FloatConst} \mid \text{DoubleConst}$

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

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.