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

第二次作业

定义你的编译器 & 汇编编程

段国豪 1911407　刘卓航 1911443

年级：2019 级

专业：信息安全、法学

指导教师：王刚

2021 年 10 月 14 日

摘要

本次实验完成编译器的设计工作，同时通过编写 SysY 语言代码以及 arm 汇编代码，进一步了解编译器的特性。首先，介绍编译器语言特性设计的理论基础-上下无关文法，同时对未来编译器支持的语言特性进行设计；之后，实现了一些汇编代码，通过熟练地掌握相关语法，从而在未来设计相关功能时可以轻车熟路，提高效率。

关键字：上下文无关文法，SysY **语言，**ARM **汇编**

目录

一、 概述

本次实验分工：
终结符、非终结符部分由两人共同完成设计
段国豪：编译单元、函数、语句的 CFG 设计，汇编代码中代码样例 1、2 的编写
刘卓航：声明、类型、定义、初值、数组的 CFG 设计，汇编代码部分代码样例 3 的编写

这里笔者需要先陈述一些实验理论基础，来使后文更为精简。

- 上下文无关文法（缩写为 CFG），在计算机科学中，若一个形式文法 G = (V_T,V_N,P,S) 的产生式规则都取如下的形式：A→ α ，则称之为上下文无关文法。其中 A ∈ V_T ，$\alpha \in V_N \cup V_T$ 。

  上下文无关文法取名为“上下文无关”的原因就是因为字符 A 总可以被字符串 α 自由替换，而无需考虑字符 A 出现的上下文。选取上下无关文法，一方面是因为它们拥有足够强的表达力来表示大多数程序设计语言的语法；实际上，几乎所有程序设计语言都是通过上下文无关文法来定义的。另一方面，上下文无关文法又足够简单，使得我们可以构造有效的分析算法来检验一个给定字符串是否是由某个上下文无关文法产生的。

  上下无关文法形式定义：

  - 上下文无关文法 G 是 4-元组：G = (V_T,V_N,P,S)
    这里的
  - V_N 是“非终结”符号或变量的有限集合。它们表示在句子中不同类型的短语或子句。
  - V_T 是“终结符”的有限集合，无交集于 V_N，它们构成了句子的实际内容。
  - S 是开始变量，用来表示整个句子（或程序）。它必须是 V_N 的元素。
  - P 是从 V_N 到 $(V_N \cup V_T)^*$ 的关系，使得$\exists$ w ∈ $(V_N \cup V_T)^*$:(S,w) ∈ R. 此外，R 是有限集合。R 的成员叫做文法的“规则”或“产生式”。星号表示闭包运算。

- BNF（巴科斯-瑙尔范式）常来表达上下文无关文法。

  BNF 规定是推导规则 (产生式) 的集合，写为：< 符号 > ::= < 使用符号的表达式 >

  这里的 < 符号 > 是非终结符，而表达式由一个符号序列，或用指示选择的竖杠'|' 分隔的多个符号序列构成，每个符号序列整体都是左端的符号的一种可能的替代。从未在左端出现的符号即为终结符。

二、 语言特性设计

(一) 终结符号 V_T

该语言的终结符号有 Ident-标识符和 InstConst-数值常量。我们规定标识符（identifier）开头必须是字母或者下划线，整体由字母、数字和下划线构成。表现为：

identifier→ identifier_nondigit
|identifier identifier_nondigit
|identifier identifier_digit

identifier_nondigit → '_' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm'
| 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'
| 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M'
| 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V'| 'W' | 'X' | 'Y' | 'Z

identifier_digit →'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

对于数值常量（IntConst），常用的有二进制、八进制、十进制和十六进制，其中默认为十进制，二进制以'0b' 开头，八进制以’ 0’ 开头，十六进制以’ 0x’ 开头：

integer_const → decimal_const
| bin_const
| octal_const
| hexadecimal_const

decimal_const → nonzero_digit| decimal_const digit

octal_const→ 0 octal_digit

bin_const→ bin_prefix bin_digit

hexadecimal_const → hexadecimal_prefix hexadecimal_digit
| hexadecimal_const hexadecimal_digit

bin_prefix → 0b' | 0B'

hexadecimal_prefix → 0x' | 0X'

nonzero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

bin_digit → 0 | 1

octal_digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

hexadecimal_digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F

(二) 非终结符集合 V_N

编译单元 CompUnit

声明 Decl 常量声明 ConstDecl 变量声明 VarDecl

基本类型 BType 函数类型 FuncType

常量定义 ConstDef 变量定义 VarDef 函数定义 FuncDef

常量初值 ConstInitVal 变量初值 InitVal

函数形参表 FuncFParams 函数形参 FuncFParam 函数实参表 FuncRParams

语句块 Block 语句块项 BlockItem 语句 Stmt

表达式 Exp 基本表达式 PrimaryExp 条件表达式 Cond 左值表达式 LVal

数值 Number

一元表达式 UnaryExp 单目运算符 UnaryOp

乘除模表达式 MulExp 加减表达式 AddExp 常量表达式 ConstExp

关系表达式 RelExp 相等性表达式 EqExp
逻辑与表达式 LAndExp 逻辑或表达式 LOrExp
注释 Ann

(三) 开始符号 S

编译单元 CompUnit

(四) 产生式集合 P

以下按照语言特性分别给出该语言各个部分的产生式。

1. **编译单元**

编译单元 CompUnit ::= CompUnit Decl | CompUnit FuncDef | Decl | FuncDef

2. **声明**

声明 Decl ::= ConstDecl ';' | VarDecl ';'
常量声明 ConstDecl ::= 'const' BType ConstDef | ConstDecl ',' ConstDef
变量声明 VarDecl ::= BType VarDef | VarDecl ',' VarDef

3. **类型、定义、初值 (不包括函数)**

基本类型 BType ::= 'int'
常数定义 ConstDef ::= Ident { [ConstExp] } '=' ConstInitVal
常量初值 ConstInitVal ::= ConstExp | {[ConstInitVal{','ConstInitVal}]}
变量定义 VarDef ::= Ident [ConstExp] | Ident [ConstExp] '=' InitVal
变量初值 InitVal ::= Exp | [InitVal','InitVal]
数值列表 InitValList ::= ConstExp | InitValList, ConstExp

4. **函数**

函数定义 FuncDef ::= FuncType Ident '(' ')'Block
| FuncType Ident '(' FuncFParams ')' Block
函数类型 FuncType ::= 'void'|'int'
函数形参表 FuncFParams ::= FuncFParam { ',' FuncFParams }
函数形参 FuncFParam ::= BType Ident ['['']{ '['Exp']' }]

5. **数组**

数组定义 ArrDef ::= ArrType Id '[' ']'|IdType Id '[' Number ']'
返回类型 ArrType ::= 'int'

6. **语句**

语句块 Block ::= '' BlockItem ''
语句块项 BlockItem ::= Decl | Stmt

语句 Stmt ::= ; | LVal '=' Exp ';'
| Exp ';'
| Block
| 'if' '(Cond ')' Stmt
| 'if' '(Cond ')' Stmt 'else' Stmt
| 'while' '(' Cond ')' Stmt
| 'break' ';'
| 'continue' ';'
| 'return' ';'
| 'return' Exp ';'

7. **表达式**

表达式 Exp ::= AddExp
条件表达式 Cond ::= LOrExp
左值表达式 LVal ::= Ident | LVal '[' Exp ']'
基本表达式 PrimaryExp ::= '(' Exp ')' | LVal | Number
数值 Number ::= IntConst
一元表达式 UnaryExp ::= PrimaryExp
| Ident '(' ')'
| Ident '(' FuncRParams ')'
| UnaryOp UnaryExp
单目运算符 UnaryOp ::= '+' | '−' | '!'
函数实参表 FuncRParams ::= Exp | FuncRParams ',' Exp
乘除模表达式 MulExp ::= UnaryExp
| MulExp '*' UnaryExp
| MulExp '/' UnaryExp
| MulExp '%' UnaryExp
加减表达式 AddExp ::= MulExp | AddExp '+' MulExp | AddExp '-' MulExp
关系表达式 RelExp ::= AddExp
| RelExp '<' AddExp
| RelExp '>' AddExp
| RelExp '<=' AddExp
| RelExp '>=' AddExp
相等性表达式 EqExp ::= RelExp | EqExp '==' RelExp | EqExp '!=' RelExp
逻辑与表达式 LAndExp ::= EqExp | LAndExp '&&' EqExp
逻辑或表达式 LOrExp ::= LAndExp | LOrExp '||' LAndExp
常量表达式 ConstExp ::= AddExp

8. **注释**

注释 Ann ::= '/*' str '*/' |'/*"*/'|"//"|"//"str

```
dgh@ubuntu:~/Desktop$ arm-linux-gnueabihf-gcc -g main.S -o main.out
dgh@ubuntu:~/Desktop$ qemu-arm -L /usr/arm-linux-gnueabihf ./main.out
hello.world
```

图 1: ”hello.world”

三、 汇编代码

为了更好地熟悉 arm 语言, 笔者编程了一段简单的代码, 功能就是输出字符串 “hello.world”, 汇编代码展示如下。

```
.global main
.section .text

main:
        mov r7 , #0x4
        mov r0 , #1
        ldr r1 , =message
        mov r2 , #13
//这里r2后面的立即数是根据”hello.world\n”来决定, ”hello.world\n”长度为12, 通过r0偏移后即可找到字符串
        swi 0

        mov r7 , #0x1
        mov r0 , #65
        swi 0

.section .data
        message:
        .ascii ”hello.world\n”
```

指令 1:

```
    arm-linux-gnueabihf-gcc main.S -c -o main.o
    arm-linux-gnueabihf-gcc main.o -o main
    qemu-arm -L /usr/arm-linux-gnueabihf ./main.out
```

生成程序运行结果如图1所示。

对 arm 语言有了基本了解以后, 笔者编写了一个简单的小程序, 功能是对输入的两个数字进行求和。代码逻辑是很简单的, 但这里笔者写了近百行汇编代码。主要难点在于数字的输入与输出处理。

这里笔者简单说明一下汇编代码的输入输出逻辑。在输入方面, 笔者先预留了 8 字节的空间来存储数字, 接着通过 svc 方法读取输入流并将数字存储于寄存器。在输出方面, 笔者先对数字进行求和, 再将数字逐位转换为 ASCII 字符进行输出。

在汇编代码这里, 由于笔者在整理资料的过程中不知道实验指导手册中的 arm 工具不支持除法指令, 因此笔者 arm-linux-gnueabihf-gcc 工具改为 arm-linux-gnueabi-gcc, 一方面是方便进行输出, 一方面是为了简化代码结构。

源代码 2:

```
    int main()
```

```
{
    int m=getint();
    int n=getint();
    m=m+n;
    putint(m);
    return 0;
}
```

汇编代码 2:

```
    .section .text
    .global _start
    _input:
        push {lr}
        push {r4-r11}
        push {r1}
        push {r0}
        mov r7,#0x3
        mov r0,#0x0
        pop {r1}
        pop {r2}
        svc 0x0
        pop {r4-r11}
        pop {pc}

    //将寄存器状态初始化，压栈进行后续操作
    _init:
        push {lr}
        push {r4-r11}
        mov r2,#0x0
        mov r5,#0x0
        mov r6,#1
        mov r7,#10
        //倍数1和10，获取数字各位

    //功能：测量数字转化成的字符串长度
    _str_length:
        ldrb r8,[r0]
        cmp r8,#0xa
        beq _count
        add r0,r0,#1
        add r2,r2,#1
        b _str_length

    _count:
        sub r0, r0, #1
        ldrb r8, [r0]
        sub r8, r8, #0x30
        //r8内部是数字字符，我们要将其转换成数字进行运算
```

```
40          mul r4, r8, r6
41          mov r8, r4
42          mul r4, r6, r7
43          mov r6, r4
44          add r5, r5, r8
45          sub r2, r2, #1
46          cmp r2 , #0x0
47          //如果所有数字都检索完成，就可以离开循环，否则继续检索下一位数字
48          beq _leave
49          b _count
50
51      //压栈后进行出栈
52      _leave:
53          mov r0,r5
54          pop {r4-r11}
55          pop {pc}
56
57      int_to_str:
58          push {lr}
59          push {r4-r11}
60          mov r2,#0x0
61          mov r3,#1000
62          mov r7,#10
63
64      _loop:
65          mov r4,#0x0
66          udiv r4,r0,r3
67          //转换成ASCII码0-9：48-57，刚好差0x30
68          add r4, r4,#0x30
69
70          ldr r5,=sum
71          add r5, r5, r2
72          strb r4,[r5]
73          add r2,r2,#1
74
75          sub r4,r4,#0x30
76          mul r6,r4,r3
77          sub r0, r0,r6
78
79          udiv r6,r3,r7
80          mov r3,r6
81          cmp r3,#0
82          //根据商情况决定是否进行下次循环，商为0，说明数字全部处理完毕，就可以
                离开循环
83          beq _leave_int
84          b _loop
85
86      _leave_int:
```

```
87          mov r4,#0xa
88          ldr r5,=sum
89          add r5,r5,r2
90          add r5,r5,#1
91          strb r4,[r5]
92          pop {r4-r11}
93          pop {pc}
94
95      _output:
96          push {lr}
97          push {r4-r11}
98          mov r7,#0x4
99          mov r0,#0x1
100         ldr r1,=sum
101         mov r2,#0x8
102         svc 0x0
103         pop {r4-r11}
104         pop {pc}
105
106     _start:
107         //存储输入的两个数字
108         ldr r0,=first
109         ldr r1,=#0x6
110         bl _input
111         ldr r0,=second
112         ldr r1,=#0x6
113         bl _input
114         //将输入流存入寄存器，方便函数调用
115         ldr r0,=first
116         bl _init
117         mov r4,r0
118         ldr r0, =second
119         bl _init
120         mov r5, r0
121
122         add r0,r4,r5
123         //转换成字符串
124         bl int_to_str
125
126         //将结果进行输出
127         bl _output
128         //恢复初始状态
129         mov r0,#0x0
130         mov r7,#0x1
131         svc 0x0
132     .section .data
133     first:
134         .skip 8
```

```
dgh@ubuntu:~/Desktop$ arm-linux-gnueabi-as  main2.S -o main2.o
dgh@ubuntu:~/Desktop$ arm-linux-gnueabi-gcc main2.o -o main2 -static -nostdlib
dgh@ubuntu:~/Desktop$ ./main2
9
9
0018
dgh@ubuntu:~/Desktop$ ./main2
3
123
0126
```

图 2: result2

```
dgh@ubuntu:~/Desktop$ ./main2
6
6
0012
dgh@ubuntu:~/Desktop$ ./main2
125
5
0130
dgh@ubuntu:~/Desktop$ ./main2
666
999
1665
dgh@ubuntu:~/Desktop$
dgh@ubuntu:~/Desktop$ ./main2
1999
11
2010
```

图 3: result3

```
135     second:
136         .skip 8
137     sum:
138         .skip 8
```

使用指令 2 便可以将代码转变为可执行程序:

```
1 arm-linux-gnueabi-as  .S -o .o
2 arm-linux-gnueabi-gcc .o -o <program name> -static -nostdlib
```

汇编代码程序运行结果如图2和图3所示。源代码 3

```
1     void sort(int a[]){
2         int i=0,j=0;
3         int min=i;
4         while(i<10){
5                 j=i+1;
6                 while(j<10){   //find min
7                         if(a[j]<a[min]){
8                                 min=j;
9                         }
10                        j++;
```

```
                }
                int temp=a[min];
                a[min]=a[i];
                a[i]=temp;
                i++;
                min=i;
        }
}


int main(){
        int a[10];
        getarray(a);

        sort(a);

        putarray(10,a);

        printf("\n");
}
```

汇编代码 3:

```
    .arch armv7-a
        .arm

        .data
        .global a
        .align 4
        .size a,40
a:
        .word 0
        .word 0
    .word 0
    .word 0
    .word 0
    .word 0
    .word 0
    .word 0
    .word 0
    .word 0



        .global main
        .type main, %function

main:
```

```
27          @call getarray 获得输入
28          ldr r0,=a
29          bl getarray
30
31          @call sort  排序
32          ldr r0,=a
33          bl sort
34
35          @call putarray  输出结果
36          ldr r1,=a
37          mov r0,#10
38          bl putarray
39
40          bx lr
41
42
43  @排序, 参考上面的sysy代码进行写作, 基本是逐句翻译
44          .global sort
45          .type sort, %function
46  sort:
47  @i=0
48          mov r1,#0
49  @j=0
50          mov r2,#0
51  @min=i
52          mov r3,r1
53
54  outerLoop:
55  @ <10 <==> >9
56          cmp r1,#36
57          bgt endloop
58          add r2,r1,#4
59
60  InnerLoop:
61          cmp r2,#36
62          bgt endInnerLoop
63  @put a[j] in r9
64          add r4,r0,r2
65          ldr r9,[r4]
66
67  @put a[min] in r8
68          add r5,r0,r3
69          ldr r8,[r5]
70
71          cmp r9,r8
72          bge if_end
73          mov r3,r2
74  if_end:
```

```
75          add r2,r2,#4
76          b InnerLoop
77
78  endInnerLoop:
79  @put a[i] in r10
80          add r4,r0,r1
81          ldr r10,[r4]
82
83  @put a[min] in r8
84          add r5,r0,r3
85      ldr r8,[r5]
86
87  @put temp in r9, and then, swap
88          mov r9,r8
89          mov r8,r10
90          mov r10,r9
91
92  @store a[i] and a[min]
93          str r10,[r0,r1]
94          str r8,[r0,r3]
95
96          add r1,r1,#4
97          mov r3,r1
98          b outerLoop
99
100 endloop:
101         bx lr
```

通过以下指令进行汇编、运行

```
1  //汇编（链接sysy运行时库）
2  arm-linux-gnueabihf-gcc -mcpu=cortex-a72 -g mysort.s -o mysort.out
       sysyruntimelibrary/libsysy.a
3  //运行
4  qemu-arm -L /usr/arm-linux-gnueabihf ./mysort.out
```

```
liuhangzhuo@ubuntu:~/Desktop/compiler/lab2/sort$ qemu-arm -L /usr/arm-linux-gnueabihf ./mysort.out
10
0
9
8
6
7
5
4
1
2
3
10: 0 1 2 3 4 5 6 7 8 9
qemu: uncaught target signal 11 (Segmentation fault) - core dumped
Segmentation fault (core dumped)
```

首先输入数组个数（10），之后输入数组数据，最后显示出了正确的结果，但是，由于不明原因，最后显示出现了 segment fault，这可能是库函数的原因（因为将调用 sort 函数的代码注

释掉，仅运行 getarray 和 putarray 时（即将输入数据直接输出），仍然会产生同样的 segment fault 输出），也可能是数组存储位置的原因。

四、 gitlab 链接

https://gitlab.eduxiji.net/nku2021-daybreaker/lab2.git

五、 总结

本实验中，完成了最后要实现的语言特性的 CFG 设计，进行了 arm 汇编代码的写作练习。通过这次实验，对于 sysy 语言的特性、arm 汇编写作、上下文无关文法有了更好的掌握。