南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

编译系统原理工程作业

预备工作 2——定义你的编译器 & 汇编编程

辛紫遇 王晓凡

年级 ： 2019 级

专业 ： 计算机科学与技术 信息安全

指导教师 ： 王刚

2021 年 10 月 12 日

摘要

本文使用上下文无关文法描述了我们需要实现的编译器所支持的 SysY 语言特性，用该语言设计了一系列能够尽可能多体现语言特性程序，编写了等价的 ARM 汇编程序并测试。

关键字：SysY，CFG，ARM 汇编

目录

一、 引言

1983 年，ANSI 为 C 语言创立了一套标准，即 ANSI C。这套标准经历了 C89，C90，C99 和 C11 四个阶段，几乎被所有广泛使用的编译器支持。ANSI C 包括关键字，基础数据类型，结构类型，数组，指针，字符串，运算符号，控制语句，函数和输入输出等特性。在本课程中，我们将仿照特性这些实现它的一个子集--SysY 语言。

二、 SysY 语言的 CFG 描述

一个完善有效的语言应当被使用上下文无关文法描述。我们将参照 SysY 语言定义，自底向上地组织起整个语言。由于字体难以排版，本文使用单引号引起所有终结符。

(一) 标识符

标识符应该由字母或下划线引起，后面跟随一系列字母、数字或下划线。

```
      id -> nondigit
          | id nondigit
          | id digit
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'
          |'_'
   digit -> '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
```

(二) 数值常量

数值常量可以为十进制数、0 引起的八进制数、0x 或 0X 引起的十六进制数。

```
    int-const -> dec-const | oct-const | hex-const
    dec-const -> nonzero-digit | dec-const digit
    oct-const -> 0 | oct-const oct-digit
    hex-const -> hex-prefix hex-digit | hex-const hex-digit
   hex-prefix -> '0x' | '0X'
nonzero-digit -> '1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
    oct-digit -> '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'
    hex-digit -> '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
                |'a'|'b'|'c'|'d'|'e'|'f'
                |'A'|'B'|'C'|'D'|'E'|'F'
```

(三) 数据类型

变量只有整数类型，函数返回值可以有整数或空类型。

```
   BType -> 'int'
FuncType -> 'void' | 'int'
```

（四） 运算符

运算符分为三个优先级，从低到高分别为加减运算符、乘除取模运算符、单目运算符。

```
  AddOp -> '+' | '-'
  MulOp -> '*' | '/' | '%'
UnaryOp -> '+' | '-' | '!'
   EqOp -> '==' | '!='
  RelOp -> '<' | '>' | '<=' | '>='
```

（五） 算数表达式

算数表达式分为一般算数表达式和常量表达式，优先级与运算符一致。函数调用被认为是一个单目表达式。还需要定义括号表达式和合法左值表达式。

```
        Exp -> AddExp
   ConstExp -> AddExp （仅当所有id均为常量时）
     AddExp -> MulExp | AddExp AddOp MulExp
     MulExp -> UnaryExp | MulExp MulOp UnaryExp
   UnaryExp -> PrimaryExp | UnaryOp UnaryExp
             | id '()' | id '(' FuncRParams ')'
FuncRParams -> Exp | FuncRParams ',' Exp
 PrimaryExp -> '(' Exp ')' | LVal | int-const
       LVal -> id | id '[' Exp ']'
  ConstLVal -> id | id '[' ConstExp ']'
```

（六） 条件表达式

条件表达式的优先级从低到高为：或表达式、与表达式、相等不等表达式、关系表达式。

```
   Cond -> LOrExp
 LOrExp -> LAndExp | LOrExp '||' LAndExp
LAndExp -> EqExp | LAndExp '&&' EqExp
  EqExp -> RelExp | EqExp EqOp RelExp
 RelExp -> AddExp | RelExp RelOp AddExp
```

（七） 语句

语句分为赋值语句、分号语句、表达式语句、if 语句、if-else 语句、while 语句、break、continue、return 语句。语句块也被视为一个语句。多个声明和语句被花括号包括即成为语句块。声明又分为常量声明和变量声明。

```
     Block -> '{' BlockItems '}'
BlockItems -> BlockItem | BlockItems BlockItem
 BlockItem -> Decl | Stmt
      Decl -> ConstDecl | VarDecl
      Stmt -> LVal '=' Exp ';'
            | ';' | Exp ';' | Block
            | 'if' '( Cond ')' Stmt
```

```
            | 'if' '( Cond ')' Stmt 'else' Stmt
            | 'while' '(' Cond ')' Stmt
            | 'break' ';' | 'continue' ';'
            | 'return' ';' | 'return' Exp ';'
```

（八） 常量声明语句

声明语句需要考虑是否为常量声明。一个声明语句可以声明多个变量，每个声明都可赋初值，数组初值可以由花括号引起。

```
    ConstDecl -> 'const' BType ConstDefs ';'
    ConstDefs -> ConstDef | ConstDefs ',' ConstDef
     ConstDef -> ConstLVal '=' ConstInitVal
 ConstInitVal -> ConstExp | '{' '}' | '{' ConstInitVals '}'
ConstInitVals -> ConstInitVal | ConstInitVals ',' ConstInitVal
```

（九） 变量声明语句

```
 VarDecl -> BType VarDefs ';'
 VarDefs -> VarDef | VarDefs ',' VarDef
  VarDef -> ConstLVal | ConstLVal '=' InitVal
 InitVal -> Exp | '{' '}' | '{' InitVals '}'
InitVals -> InitVal | InitVals ',' InitVal
```

（十） 函数定义

函数定义由类型、函数名、参数表和函数体组成，参数可以有 0 个或多个。

```
    FuncDef -> FuncType id '(' ')' Block
             | FuncType id '(' FuncFParams ')' Block
FuncFParams -> FuncFParam | FuncFParams ',' FuncFParam
 FuncFParam -> BType id | BType id '[' ']'
             | FuncFParam '[' Exp ']'
```

（十一） 编译单元

在最顶层，所有的声明和函数定义被集成为编译单元。

```
CompUnit -> Unit | CompUnit Unit
    Unit -> Decl | FuncDef
```

三、 样例程序

我们根据上一节定义的语言设计了一段代码，并力求其能全面地包含尽可能多的语言特性：

SysY 语言样例程序

```
#include "sylib.h"

int i, j=101;

void doNothing(){
    int k = 1;
    0;;
    return;
}

int func(int x, int y){
    if(x != y) return 5;
    else {
        doNothing();
        return 6;
    }
}

int main(){
    int a[2][2] = {{0, 1}, {}}; // 0 1 0 0
    const int b = -9+5*2; // 1
    while(!(0 && b <= 5)){ // true
        a[1][b] = func(7, 8); // 0 1 0 5
        if(a[0][0] || b==0) continue; // false
        break;
    }
    putarray(4, a); // 0 1 0 5
    putint(b); // 1
    return 0;
}
```

将这段代码编写成等价的 ARM 汇编代码：

ARM 汇编样例程序

```
    .arch       armv7-a
    .arm
    .file       "test.c"
    @ 以下是代码段
    .text

    @ i是全局变量，但没有初始化，所以我们放在.bss段
    .globl      i
    .bss

    @ 保持内存对齐
    .align      2

```

```
14      @ i是我们int实例出来的一个对象，因此是object类型，又因为int是4字节，因此i
            的size为4
15      .type        i, %object
16      .size        i, 4
17
18      @ 上面只是通知汇编器我们定义的i是个什么变量，我们还需要给i以内存空间
19  i:
20      .space       4
21
22      @ j是初始化的全局变量，我们放在.data段，同时，因为j的大小是4字节(一个字)
            ，我们于是用.word给j赋值；其余的跟i一样
23      .globl       j
24      .data
25      .align       2
26      .type        j, %object
27      .size        j, 4
28  j:
29      .word        101
30
31      @ 以下是代码段
32      .text
33      @ 声明下面的代码段是一个函数
34      .align       1
35      .globl       doNothing
36      .type        doNothing, %function
37  doNothing:
38      push         {lr,fp} @ 这里我们压栈保存返回地址和栈基地址
39      add fp, sp, #0       @ 提升fp指向sp
40      sub sp, sp, #4       @ 扩栈，留下4字节大小的栈空间(因为只要这么多)
41
42      mov          r4, #1                  @ r4存放我们的局部变量k，因为r4是局部
            变量寄存器
43      str          r4, [sp, #4]    @ 这里我们把变量k放到内存中存储，注意是r7(fp)
            +4的位置，
44
45      nop                          @ 注意，这里很有趣，我们源文件这里是"0;;"，但
            是毫无意义，因此被翻译为空指令
46
47      add          sp, #4                  @ 回收之前的栈空间
48      pop          {fp}                        @ 弹栈，将之前保存的值返还回
            去
49      pop          {lr}
50
51      bx  lr                       @ lr寄存器记录着返回地址，现在我们返回去
52      .size        doNothing, .-doNothing
53
54      .align       1
55      .globl       func
```

```
56      .type       func, %function
57  func:
58      push        {lr, fp}
59      add fp, sp, #0
60      @ 注意这里我们不需要扩栈了，因为没有新建变量
61
62      @ 注意，在arm汇编中是从左至右一次放入r0,r1,r2,r3中的，例如这里调用func(x,
            y)，r0存储x，r1存储y
63      cmp         r0, r1
64      beq         .LABLE_of_func
65      mov         r0, #5                                          @ 返回值放在
            r0
66      b           .LABLE_return_of_func           @ 这两句相当于return 5;
67  .LABLE_of_func:
68      bl          doNothing(PLT)                          @ 因为这是汇编代码，
            还没分配具体的内存，因此不知道函数所在的内存地址；同时，目前操作系统
            规定代码段不能修改，因此均采用PLT表作为跳转，
69                                      @ 这样我们编写汇编代码时就不需要关心
                                            函数的具体位置或偏移，只需要标注
                                            一下让汇编器自动往对应的PLT项跳转
                                            即可
70
71      mov         r0, #6
72  .LABLE_return_of_func:
73      pop         {fp}
74      pop         {lr}
75      bx   lr                                                     @ 这
            次我们没有使用栈，就不用恢复栈了，把保存的值恢复出来即可
76      .size       func, .-func
77
78      .align      1
79      .globl      main
80      .type       main, %function
81  main:
82      push        {lr, fp}
83      add fp, sp, #0
84      sub sp, sp, #20   @ 准备20的空间，因为我们有a[2][2]共16字节+ const int b
            共4字节
85
86      @ 局部变量a[2][2]的空间位于sp+4到sp+16的位置，从低地址向高地址增长
87      mov r0, #0
88      str r0, [sp, #4]            @ a[0][0] = 0
89      str r0, [sp, #12]           @ a[1][0] = 0
90      str r0, [sp, #16]           @ a[1][1] = 0
91      mov r0, #1
92      str r0, [sp, #8]            @ a[0][1] = 1
93
94      @ 局部变量 const int b 存储于栈顶，我们可以看到先声明的变量放在高地址，后
```

```
        声明的变量放在低地址
95      @ gcc给出的汇编是直接复制公式结果了，我们这里为体现过程要去计算
96      mov r0, #5
97      mov r1, #2
98      mul r3, r1, r0              @ 计算5*2
99      add r0, r3, #-9             @ 计算-9+5*2
100     str r0, [sp]                @ const int b = -9+5*2
101
102     @ 我们进入了while循环
103 .LABLE_WHILE_LOOP:
104     @ 这一步代码的意思是：我们设置r0为b是否小于等于5的flag，若小于等于5，则r0
        为1，否则为0
105     mov r0, #1
106     ldr r1, [sp]
107     cmp r1, #5
108     bls .LABLE_not_set_flag
109     mov r0, #0
110 .LABLE_not_set_flag:
111
112     and r0, r0, #0              @ 等价于0&&b<=5
113     mvn r0, r0
114     and r0, r0, #1      @ 这两步等价于!(0&&b<=5)
115     cmp r0, #1
116     bne .LABLE_END_LOOP @ 这里注意一下，在源程序中我们可以直接算出表达式的结
        果是1，因此while条件总是执行的，但这里我们为了体现计算条件表达式这一
        过程，我们设计如上“冗余”的代码
117
118     mov r1, #8
119     mov r0, #7
120     bl  func(PLT)
121     mov r1, #2
122     mov r2, #1
123     mul r1, r2, r1              @ r1 = 2*1
124     ldr r2, [sp]                @ r2 = b
125     add r1, r1, r2      @ r1 = 2*1 + b，这里我们的目的是获得a[1][b]的偏移量，
        因为a一行两个元素，故1*2，又因为在b列，故1*2+b
126     lsl r1, r1, #2      @ r1 = 4*(2*1+b)，r1左移两位即r1乘4，之所以要乘四是因
        为a中每个元素都是int，占4个字节
127
128     add r1, #4                  @ 因为a的地址在sp+4的位置，所以r1要加4
129     str r0, [sp, r1]    @ a[1][b] = r0 = func(7,8)
130     ldr r0, [sp, #4]    @ r0 = a[0][0]
131
132     ldr r1, [sp]                        @ r1 = b
133     cmp r1, #0
134     mrs r1, cpsr
135     and r1, r1, #0x40000000     @ 这步操作是为了获取b==0的布尔值结果，去掉其
        它结果，cmp指令的比较结果会写到cpsr寄存器的zero flag上，若相等则为1，
```

```
        不等则为0
136     lsr r1, #30                         @ 因为zero flag在第31位，所以要逻辑右
        移30位，得到结果：r1 = (b == 0)
137
138     orr r0, r0, r1                      @ 计算 a[0][0] || (b == 0)
139     cmp r0, #0
140     bgt .LABLE_WHILE_LOOP       @ 如果(a[0][0] || (b == 0))为真，则continue
141     nop                                         @ 否则，直接break；因为我们这
        里是直接退出循环并不做任何操作，所以我们直接一个nop表示break;语句
142
143 .LABLE_END_LOOP:
144     add r1, sp, #4              @ r1 = a的地址
145     mov r0, #4
146     bl  putarray(PLT)    @ 调用putarray()
147
148     ldr r0, [sp]                @ r0 = b
149     bl  putint(PLT)             @ 调用putint()
150
151     mov r0, #0                  @ return 0; r0存储函数返回值
152     add sp, sp, #20             @ 回收栈空间
153
154     pop         {fp}
155     pop         {lr}
156     bx   lr                             @ 返回
157
158     .size       main, .-main
159     .ident      "GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0"
160     .section    .note.GNU-stack,"",%progbits
```

运行这段汇编代码，结果如下：

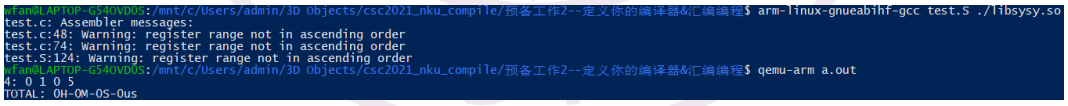

图 1: 汇编代码运行结果

四、 总结

(一) 分工情况

辛紫遇负责 CFG 设计、SysY 语言程序设计、文档编写、汇编代码微调，王晓凡负责汇编代码编写与测试、文档修改。

(二) Gitlab 链接

https://gitlab.eduxiji.net/Wang_XiaoFan/csc2021_nku_compile