

# Runtime Organization and Code Generation Exercises

## CS 4610 — Spring 2017

This Review Set asks you to prepare written answers to questions on code generation. Each of the questions has a short answer. You may discuss this Review Set with other students and work on the problems together.

### 1 Definitions and Background

1. Define the following terms and give examples where appropriate.

(a) Compiler:

(b) Stack Machine:

(c) Register:

(d) Instruction Set Architecture (ISA):

(e) RISC Machine:

(f) CISC Machine:

(g) Code Generation:

(h) Activation Record (Stack Frame):

(i) Liskov Substitution Principle:

(j) Dispatch Table (vtable):

(k) Address Space:

2. What invariants must we maintain in our stack machine with an accumulator architectural model?

## 2 Runtime Organization

1. Consider the following piece of code:

```
1 fact(n : Int) : Int {  
2   if n > 0 then n*fact(n-1) else 1 fi  
3 };
```

Draw the tree of activation records for a call to `fact(4)`. Include at least the stack pointers, frame pointers, return values, and parameters.

### 3 Code Generation

For these questions, consider the simple language grammar and code generation routines we discussed in class:

$$\begin{aligned} P &\rightarrow D; P \mid D \\ D &\rightarrow \text{def id}(ARGS) = E; \\ ARGS &\rightarrow \text{id}, ARGS \mid \text{id} \\ E &\rightarrow \text{int} \mid \text{id} \mid \text{if}(E_1 = E_2) \{E_3\} \text{else} \{E_4\} \\ &\quad \mid E_1 + E_2 \mid E_1 - E_2 \mid \text{id}(E_1, \dots, E_n) \end{aligned}$$

1. Generate code for the following code:

```
1 def fib(x) = if ( x = 1 ) { 0 } else {
2           if (x = 2 ) { 0 } else {
3             fib( x - 1 ) + fib( x - 2 )
4           }
5         };
```

2. The code you generated in the previous question is not particularly space-efficient with regard to the calling sequence. While still pushing arguments onto the stack, what could be done to reduce the overall number of instructions being generated?

3. Storing intermediate values (e.g. the value of  $E_1$  in an addition) is costly; In addition to the load/store instructions, we also need to update the value of the stack pointer (this is hidden by the pop and push instructions). One way to optimize this code is to allocate space in the activation record for the intermediate values (we can access these locations as a fixed offset from the frame pointer). We can reuse these temporary locations from previous code generation routines. To support this, we will need to track how many temporary variables are needed for each code generation routine; we define this as the  $NT(e)$  function. For example:

$$NT(e_1 + e_2) = \max(NT(e_1), 1 + NT(e_2))$$

because we need at least as many temporaries as code generation for  $e_1$  requires, and at least as many temporaries as code generation for  $e_2$  needs (plus one extra to store the value of  $e_1$ ).

(a) Write out the  $NT$  equations for the remaining expressions in the simple language.

(b) Use your equations to determine  $NT(\text{fib})$  (from question 1).

4. Suppose we wish to support assignment to variables in our simple language:

$$E \rightarrow \text{id} \leftarrow E$$

Write a code generation routine for this expression.

5. Draw a suitable object and vtable layout for the following code (Don't forget inherited fields and methods!).

```
1 class Point {
2     vx : Int <- 0;
3     vy : Int <- 0;
4     move( x : Int, y : Int) : SELF_TYPE {
5         { vx <- x + vx; vy <- y + vy; self; }
6     };
7 };
8 class ColorPoint inherits Point {
9     r : Int <- 0;
10    g : Int <- 0;
11    b : Int <- 0;
12    changeColor( new_r : Int, new_g : Int, new_b : Int) : SELF_TYPE {
13        { r <- new_r; g <- new_g; b <- new_b; self; }
14    };
15 };
16 class Main inherits IO {
17     main() : Object {
18         let x : ColorPoint <- new ColorPoint in
19             { x <- x.changeColor( 25, 25, 25 ); out_string("Hello, world!\n"); }
20     };
21 };
```