Runtime Organization and Code Generation Exercises
CS 364 — Spring 2022

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) Activation Record (Stack Frame):

   (g) Liskov Substitution Principle:

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:

```
fact(n) {
  if (n > 0) {
    n * fact(n-1);
  } else {
    1;
  };
};
```

   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:

$$
\begin{array}{rcl}
P & \rightarrow & D; P \mid D \\
D & \rightarrow & \texttt{def id}(ARGS) = E; \\
ARGS & \rightarrow & \texttt{id}, ARGS \mid \texttt{id} \\
E & \rightarrow & \texttt{int} \mid \texttt{id} \mid \texttt{if}(E_1 = E_2)\,\{E_3\}\,\texttt{else}\,\{E_4\} \\
 & & \mid E_1 + E_2 \mid E_1 - E_2 \mid \texttt{id}(E_1, \ldots, E_n)
\end{array}
$$

1. Generate code for the following program. You may assume the code for the `mod` function already exists and is callable.

```
1 def gcd(x, y) = if ( y = 0 ) { x } else {
2                   gcd(y, mod(x,y))
3               };
```

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 additional 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 $NT(\texttt{gcd})$ (from question 1).

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

$$E \quad \rightarrow \quad \texttt{id} \leftarrow E$$

   Write a code generation routine for this expression.