Types & Programming Languages
Exercises 1

These exercises are based on the material in Lectures 1 and 2.

1. For each of the following expressions, show its sequence of reductions as far as possible, and give a full derivation of each reduction step.

   (a) (if 1==1 then 2 else 3)+(1+2)

   (b) if 1+2 then 3 else 4

2. Extend the language with a multiplication operator: modify the abstract syntax and give suitable reduction rules.

3. Similarly extend the language with a logical or operator, giving both straightforward and "short-cut" reduction rules in the same way that we did for logical and.

4. This exercise explores the relationship between the reduction relation and realistic evaluation of expressions on a stack. For each expression e we define $\mathit{compile}(e)$ to be a sequence of instructions for a stack machine. First of all we ignore conditional expressions, and define

$$\begin{array}{rcl} \mathit{compile}(v) & = & \mathsf{push}(v) \\ \mathit{compile}(e+e') & = & \mathit{compile}(e)\ \mathit{compile}(e')\ \mathsf{add} \\ \mathit{compile}(e==e') & = & \mathit{compile}(e)\ \mathit{compile}(e')\ \mathsf{eqtest} \\ \mathit{compile}(e\&e') & = & \mathit{compile}(e)\ \mathit{compile}(e')\ \mathsf{and} \end{array}$$

   The instruction push places a value onto the stack, the instruction add replaces the top two values on the stack with their sum, and the instructions eqtest and and work similarly. The stack may contain both integer and boolean values.

   For example, $\mathit{compile}(1+(2+3)) = \mathsf{push}(1)\ \mathsf{push}(2)\ \mathsf{push}(3)\ \mathsf{add}\ \mathsf{add}$. Executing this sequence of instructions would result in a stack containing the value 6.

   $\mathit{compile}(1+(2==3)) = \mathsf{push}(1)\ \mathsf{push}(2)\ \mathsf{push}(3)\ \mathsf{eqtest}\ \mathsf{add}$. Executing this sequence of instructions would generate a run-time type error or get stuck.

   (a) What can you say about the relationship between the result of reducing an expression e, and the result of executing $\mathit{compile}(e)$? What can you say about the relationship between individual reduction steps and the execution of individual stack instructions?

   (b) Suppose we extend *compile* to conditional expressions by defining

$$\mathit{compile}(\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'') \quad = \quad \mathit{compile}(e')\ \mathit{compile}(e'')\ \mathit{compile}(e)\ \mathsf{cond}$$

   where the instruction cond replaces the top three values on the stack with either the second or third value from the top, depending on whether the top value is true or false.

   What is the difference between reduction and stack evaluation of conditional expressions? If we want reduction to match this definition of stack evaluation, what should the reduction rules for conditional expressions be? Why don't we want to modify the reduction rules, and how should we modify stack execution instead?