CSCI 3155: Principles of Programming Languages Midterm practice questions Spring 2018

Name: _____

---

1. Consider the following function in Scala:

```
1   def f(n: Int): Boolean = {
2     require(n >= 0)
3     def g(n: Int): Boolean = n match {
4       case 0 => false
5       case n => f(n - 1)
6     }
7     n match {
8       case 0 => true
9       case _ => g(n - 1)
10    }
11  }
```

   (a) **Scoping**. For each variable use site in the above code, draw an arrow from the use site to the binding site? **Hint**: There a total of 7 use sites.

> **Solution:**
>
> - Use of `n` at 3, bound at 1.
> - Use of `n` at 5, bound at 5.
> - Use of `f` at 9, bound at 1.
> - Use of `n` at 9, bound at 9.
> - Use of `n` at 13, bound at 1.
> - Use of `g` at 17, bound at 5.
> - Use of `n` at 17, bound at 1.

   (b) **Evaluation**. What does `f(3)` evaluate to? Please show the important steps of evaluation using the "e steps e' in 0 or more steps" relation from the course notes: $e \longrightarrow^* e'$. For this question, we will consider function call expressions (e.g., `f(3)` and `g(2)`) and the final value important. For example, the first step is `f(3)` $\longrightarrow^*$ `g(2)`. You are welcome to chain the step relation (e.g., writing $e \longrightarrow^* e' \longrightarrow^* e'' \longrightarrow^* \cdots \longrightarrow^* v$ where the e's are function call expressions and v is the final value).

> **Solution:** `f(3)` $\longrightarrow^*$ `g(2)` $\longrightarrow^*$ `f(1)` $\longrightarrow^*$ `g(0)` $\longrightarrow^*$ **false**

   (c) Explain briefly in one sentence what function `f` computes?

> **Solution:** Function `f` returns **true** if and only if its argument is an even natural number.

2. **Recursion**. Implement a recursive function that computes $\texttt{x}^{\texttt{n}}$ (i.e., raises `x` to the `n`th power). It does not need to be particularly efficient.

```
def pow(x: Int, n: Int): Int =
```

Solution:

```
def pow(x: Int, n: Int): Int = n match {
  case 0 => 1
  case _ => x * pow(x, n - 1)
}
```

This implementation is $O(\texttt{n})$, which is not the most efficient one.

3. **Recursive Data Structures**. Consider the following recursive data type `SearchTree` from Lab 1:

```
sealed abstract class SearchTree
case object Empty extends SearchTree
case class Node(l: SearchTree, d: Int, r: SearchTree) extends SearchTree
```

Implement a function `sum` recursively that sums up all of the data values in all nodes of tree `t`. For extra credit, re-implement `sum` using an accumulator parameter (note, your implementation is not likely to be tail recursive, which is ok)—come to office hours to claim your extra credit.

```
def sum(t: SearchTree): Int =
```

Solution: The following solution is probably the most straightforward one:

```
def sum(t: SearchTree): Int = t match {
  case Empty => 0
  case Node(l, d, r) => sum(l) + d + sum(r)
}
```

Here, we implement `sum` using a helper function with an accumulator parameter. Note that this implementation is not tail recursive because there is work to be done after return from the `loop(l, acc)` recursive call.

```
def sum(t: SearchTree): Int = {
  def loop(t: SearchTree, acc: Int): Int = t match {
    case Empty => acc
    case Node(l, d, r) => loop(r, d + loop(l, acc))
  }
```

```
    loop(t, 0)
  }
```

In case you are curious, it is possible to implement `sum` tail recursively using what is called a *continuation*:

```
  def sum(t: SearchTree): Int = {
    def loop(t: SearchTree, k: Int => Int): Int = t match {
      case Empty => k(0)
      case Node(l, d, r) => loop(l, a => d + loop(r, k))
    }
    loop(t, a => a)
  }
```

The continuation parameter `k` is like an accumulator the accumulates the operations that need to be done to compute the sum. We have not (yet) discussed continuations, so you are not expected to have been able to come up with this solution. It is, however, informative to trace through an evaluation using this solution to see how it works.

4. **Grammars: Derivations and Ambiguity**. Consider the following grammar with numbers n, an if-then-else expression, and an addition expression.

$$e ::= n \mid \textbf{if } (e)\ e\ \textbf{else } e \mid e + n$$

(a) [5 points] Is the sentence

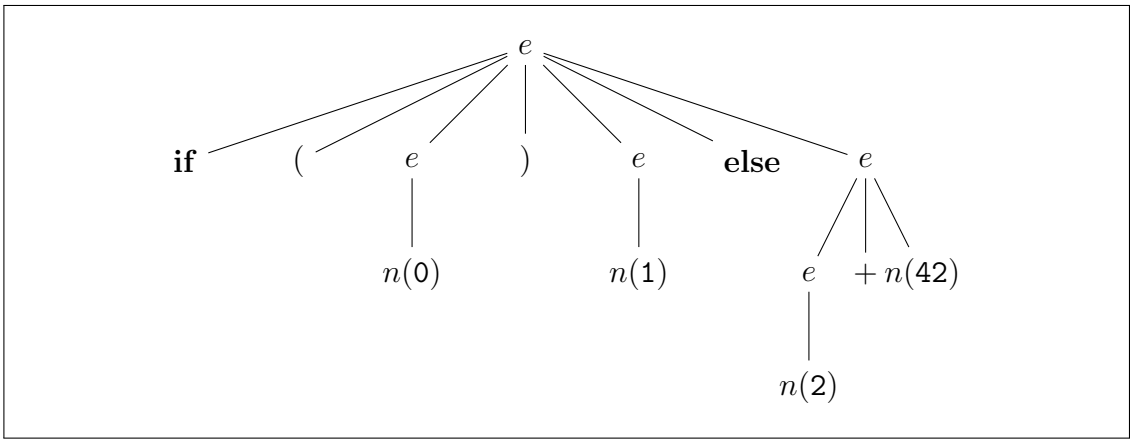
$$\textbf{if } (0)\ 1\ \textbf{else } 2 + 42$$

in the language for e described by the above grammar? If so, give a *derivation* with the grammar for the sentence. If not, briefly explain why in no more than 1–2 sentences.

Solution: Yes, this sentence is in the language given above. Here's a derivation that witnesses this statement:

$$\begin{aligned} e &\implies \textbf{if } (e)\ e\ \textbf{else } e \\ &\implies \textbf{if } (n(0))\ e\ \textbf{else } e \\ &\implies \textbf{if } (n(0))\ n(1)\ \textbf{else } e \\ &\implies \textbf{if } (n(0))\ n(1)\ \textbf{else } e + n(42) \\ &\implies \textbf{if } (n(0))\ n(1)\ \textbf{else } n(2) + n(42) \end{aligned}$$

This question asked for a *derivation* as given above. However, a sentence can also be shown to be in the language described by a grammar with a *parse tree*. Here's a parse tree for this sentence:

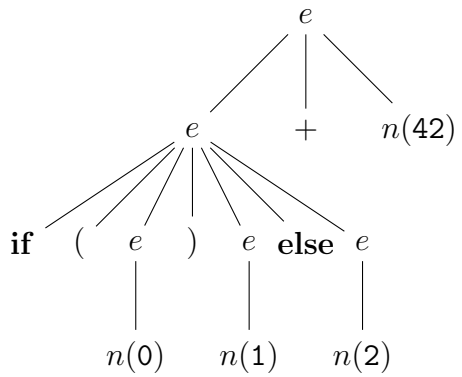


(b) [5 points] Is the above grammar ambiguous? If so, demonstrate the ambiguity by drawing two parse trees for the same sentence. If not, briefly explain why in no more than 1–2 sentences.

Solution: Yes, this grammar is ambiguous. In particular, the sentence given above (i.e., **if** (0) 1 **else** 2+42) can be used to demonstrate the ambiguity. This sentence can be read as

if (0) 1 **else** (2 + 42) or (**if** (0) 1 **else** 2) + 42 .

The left reading corresponds to the parse tree given in (a), while the right reading corresponds to the following parse tree:



5. [5 points] **Grammars: Eliminating Ambiguity**. Consider the following grammar:

$$e ::= n \mid e + e \mid e * e$$

This grammar is ambiguous (as we recall from class). Rewrite this grammar into another grammar that accepts the same set of strings but is unambiguous. Resolve the ambiguity of the grammar by making $+$ and $*$ *left associative* and so that $*$ has *higher precedence* than $+$.

Solution:

$$e ::= t \mid e + t$$
$$t ::= n \mid t * n$$

6. [5 points] **Evaluation (Synthesis).** Consider a very small subset of JAVASCRIPTY with only numbers, number variables, and **const** declarations:

expressions	$e ::= n \mid x \mid \textbf{const}\ x = e_{\text{bind}}; e_{\text{body}}$
numbers	n
variables	x

Consider the following abstract syntax classes, which uses a different representation for **const** declarations and variables compared to Lab 2:

```
sealed abstract class Expr
case class N(n: Double) extends Expr
case class ConstDecl(ebind: Expr, ebody: Double => Expr) extends Expr
```

Instead of using a value environment for variables, we represent body expression e_{body} of a **const** declaration as Scala function from number values (`Double`) to expressions (`Expr`)—that is, `Double => Expr`. Observe that we have no `Expr` form for variables. As an example, the concrete syntax

```
const a = 1; a
```
(1)

is represented by the following abstract syntax tree:

```
ConstDecl( N(1) , { (a: Double) => N(a) }) .
```
(2)

Recall that the code

```
(a: Double) => N(a)
```

is a Scala function literal expression. Observe that abstract syntax tree (2) represents all concrete expressions like (1) however variable `a` is named.

Complete the following implementation of `eval` that evaluates an `Expr` to a `Double` value.

```
def eval(e: Expr): Double = e match {
  case N(n) => n
  case ConstDecl(ebind, ebody) =>
```

```
}
```

Solution:

```
def eval(e: Expr): Double = e match {
  case N(n) => n
  case ConstDecl(ebind, ebody) => eval(ebody(eval(ebind)))
}
```

7. **Small-Step Substitution Semantics: Arithmetic Expressions with Binding**. Consider an arithmetic expression language with binding:

$$e \quad ::= \quad n \mid x \mid e_1 + e_2 \mid e_1 * e_2 \mid \textbf{const}\ x = e_1; e_2$$

where n and x correspond to numbers (e.g., 0, 1, 42) and variable identifiers (e.g., `x`, `y`), respectively. The values of this language are simply numbers n. The operators $+$ and $*$ are the usual arithmetic operators. The **const** $x = e_1; e_2$ expression binds a variable x to the value e_1, which is then in scope in expression e_2.

A one-step evaluation judgment for this language has the following form: $e \longrightarrow e'$. Informally, this judgment says "Closed expression e takes one step of evaluation to expression e'." This judgment gives a small-step operational semantics for this language, which we define in this question.

(a) [3 points] Suppose that rules to evaluate expressions of the form $e_1 + e_2$ are as follows:

$$\frac{n' = n_1 + n_2}{n_1 + n_2 \longrightarrow n'} \qquad \frac{e_1 \longrightarrow e_1'}{e_1 + e_2 \longrightarrow e_1' + e_2} \qquad \frac{e_2 \longrightarrow e_2'}{e_1 + e_2 \longrightarrow e_1 + e_2'}$$

where the $+$ in the first rule is bolded to emphasize that it is the mathematical plus.

What is stated about the order of evaluation for $+$ in the above rules? Explain in 1–2 sentences.

Solution: These rule say that the evaluation of $e_1 + e_2$ is non-deterministic. The evaluator can interleave steps of evaluating e_1 or e_2.

(b) [5 points] Give rules for evaluating $e_1 * e_2$ such that the evaluation short-circuits if e_1 is 0.

Solution:

$$\textsc{DoTimes}\ \frac{n' = n_1 \mathbin{*} n_2 \qquad n_1 \neq 0}{n_1 * n_2 \longrightarrow n'} \qquad \textsc{DoTimesZero}\ \frac{}{0 * e_2 \longrightarrow 0} \qquad \textsc{StepTimesLeft}\ \frac{e_1 \longrightarrow e_1'}{e_1 * e_2 \longrightarrow e_1' * e_2}$$

$$\textsc{StepTimesRight}\ \frac{e_2 \longrightarrow e_2' \qquad n_1 \neq 0}{n_1 * e_2 \longrightarrow n_1 * e_2'}$$

The highlighted premise ensures that the DoTimesZero rule is the only one that applies when the left expression evaluates to 0.

(c) [4 points] Give rules for evaluating **const** $x = e_1$; e_2 expressions where first e_1 must be evaluated to a value and then e_2 is evaluated. Recall that x is in scope in e_2 and refers to the value of e_1. You may use the notation $e[e'/x]$ for the capture-avoiding substitution in e of e' for x.

Solution:

$$\frac{}{\textbf{const}\ x = n_1;\ e_2 \longrightarrow e_2[n_1/x]} \qquad \frac{e_1 \longrightarrow e_1'}{\textbf{const}\ x = e_1;\ e_2 \longrightarrow \textbf{const}\ x = e_1';\ e_2}$$

8. **Big-Step Environment Semantics: Dynamic Scoping**. Consider the following JavaScripty expression:

$$\textbf{const}\ \mathtt{x} = 1;\ \textbf{const}\ \mathtt{g} = (\mathtt{y}) \Rightarrow \mathtt{x};\ \big((\mathtt{x}) \Rightarrow \mathtt{g}(2)\big)(3)$$

Recall that $p(x) \Rightarrow e$ is a function literal.

(a) [4 points] What is the value of this expression under static scoping? Under dynamic scoping? Hint: this example expression, should look very familiar.

Solution: Static scoping: 1. Dynamic scoping: 3.

(b) [4 points] Recall that a derivation of a judgment is an application of inference rules arranged in a tree. Using the big-step operational semantics with dynamic scoping as in Lab 3 (see **??**), let us consider a derivation that specifies the value of the above expression. After the evaluation of the two top-level **const** bindings, there is one sub-derivation of the following form:

$$\frac{\mathscr{D}_1 \quad \mathscr{D}_2 \quad \mathscr{D}_3}{E_1 \vdash \big((\mathtt{x}) \Rightarrow \mathtt{g}(2)\big)(3) \Downarrow \square}\ \textsc{Call}$$

where $E_1 \stackrel{\text{def}}{=} \cdot[\mathtt{x} \mapsto \mathtt{1}][g \mapsto (\mathtt{y}) \Rightarrow \mathtt{x}]$ and with three sub-derivations $\mathscr{D}_1$, $\mathscr{D}_2$, and $\mathscr{D}_3$. Derivations $\mathscr{D}_1$ and $\mathscr{D}_2$ are given as follows:

$$\mathscr{D}_1 \stackrel{\text{def}}{=} \frac{}{E_1 \vdash (\mathtt{x}) \Rightarrow \mathtt{g(2)} \Downarrow (\mathtt{x}) \Rightarrow \mathtt{g(2)}}\ \textsc{Val} \qquad \mathscr{D}_2 \stackrel{\text{def}}{=} \frac{}{E_1 \vdash \mathtt{3} \Downarrow \mathtt{3}}\ \textsc{Val}$$

Complete the derivation above by filling in the box above for the value of the expression and give the sub-derivation $\mathscr{D}_3$ below. Hint: sub-derivation $\mathscr{D}_3$ consists of applications of 4 inference rules.

Solution: The box gets filled in with the value 3. The sub-derivation $\mathscr{D}_3$ is as follows:

$$\frac{\dfrac{v_1 = (\mathtt{y}) \Rightarrow \mathtt{x}}{E_1[\mathtt{x} \mapsto \mathtt{3}] \vdash \mathtt{g} \Downarrow v_1}\ \textsc{Var} \quad \dfrac{}{E_1[\mathtt{x} \mapsto \mathtt{3}] \vdash \mathtt{2} \Downarrow \mathtt{2}}\ \textsc{Val} \quad \dfrac{}{E_1[\mathtt{x} \mapsto \mathtt{3}][\mathtt{y} \mapsto \mathtt{2}] \vdash \mathtt{x} \Downarrow \mathtt{3}}\ \textsc{Var}}{E_1[\mathtt{x} \mapsto \mathtt{3}] \vdash \mathtt{g(2)} \Downarrow \mathtt{3}}\ \textsc{Call}$$