Elements of Programming Languages
Tutorial 1: Abstract syntax trees, evaluation and typechecking
Week 3 (October 2–6, 2017)

Starred exercises (⋆) are more challenging. Please try all unstarred exercises before the tutorial meeting.

1. **Pattern matching.** For this problem, you should use the Scala definition of \( L_{\text{Arith}} \) abstract syntax trees presented in the lectures:

   ```scala
   abstract class Expr
   case class Num(n: Integer) extends Expr
   case class Plus(e1: Expr, e2: Expr) extends Expr
   case class Times(e1: Expr, e2: Expr) extends Expr
   ```

   (a) Write a Scala function `evens[A]: List[A] => List[A]` that traverses a list and returns all of the elements in even-numbered positions. For example, `evens(List('a','b','c','d','e','f')) = List('a','c','e')`. The solution should use pattern-matching rather than indexing into the list.

   (b) Write a Scala function `allplus: Expr => Boolean` that traverses a \( L_{\text{Arith}} \) term and returns `true` if all of the operations in it are additions, `false` otherwise. (For this problem, you may want to use the Scala Boolean AND operation `&&`.)

   (c) Write Scala function `consts: Expr => List[Int]` that traverses a \( L_{\text{Arith}} \) expression and constructs a list containing all of the numerical constants in the expression. (For this problem, you may want to use the Scala list-append operation `++`.)

   (d) Write Scala function `revtimes: Expr => Expr` that traverses a \( L_{\text{Arith}} \) expression and reverses the order of all multiplication operations (i.e. \( e_1 \times e_2 \) becomes \( e_2 \times e_1 \)).

   (e) (⋆) Write a Scala function `printExpr: Expr => String` that traverses an expression and converts it into a (fully parenthesised) string. For example:

   ```scala
c> printExpr( Times(Plus(Num(1), Num(2)), Times(Num(3), Num(4))))
res0: String = ((1 + 2) * (3 * 4))
```

2. **Evaluation derivations.** Recall the evaluation rules covered in lectures:

\[
e \downarrow v
\]
Write out derivation trees for the following expressions:

(a) $6 \times 9$

(b) $3 \times 3 + 4 \times 4 = 5 \times 5$

(c) $(\ast) (1 + 1 = 2 \text{ then } 2 + 3 \text{ else } 2 + 3)$

(d) $(\ast) (1 + 1 = 2 \text{ then } 3 \text{ else } 4) + 5$

3. Typechecking derivations. Recall the typechecking rules covered in lectures:

\[
\vdash e : \tau
\]

\[
\begin{align*}
\vdash n \in \mathbb{N} & \quad \vdash e_1 : \text{int} \quad \vdash e_2 : \text{int} \\
\vdash b \in \mathbb{B} & \quad \vdash e_1 : \tau \quad \vdash e_2 : \tau \\
\vdash b : \text{bool} & \quad \vdash e_1 == e_2 : \text{bool} \quad \vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau
\end{align*}
\]

Write out typing derivations for the following judgments:

(a) $\vdash 6 \times 9 : \text{int}$

(b) $(\ast) \vdash (\text{if } 1 + 1 == 2 \text{ then } 3 \text{ else } 4) + 5 : \text{int}$

4. $(\ast)$ Nondeterminism. Suppose we add the following construct $e_1 \square e_2$ to $L_{\text{Arith}}$:

\[
e ::= e_1 + e_2 \mid e_1 \times e_2 \mid n \in \mathbb{N} \mid \text{true} \mid \text{false} \mid e_1 == e_2 \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \mid e_1 \square e_2
\]

Informally, the semantics of $e_1 \square e_2$ is that we evaluate either $e_1$ or $e_2$ nondeterministically. To model this we extend the evaluation rules as follows:

\[
e \downarrow v
\]

(a) What property of $L_{\text{Arith}}$ (among those discussed in Lecture 2) is violated after we add $\square$?

(b) Write a sensible rule for typechecking $e_1 \square e_2$.

(c) For each of the following expressions $e$, list all of the possible values $v$ such that $e \downarrow v$ is derivable:

i. $(1 \square 2) \times (3 \square 4)$

ii. if (true $\square$ false) then 1 else 2

(d) Define an expression $e$ and a value $v$ such that there are two different derivations of the judgment $e \downarrow v$. (What does it mean for the derivations to be different?)