Elements of Programming Languages
Tutorial 1: Abstract syntax trees, evaluation and typechecking
Week 3 (October 5–9, 2015)

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
scala> 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) if $1 + 1 == 2$ then $2 + 3$ else $2 * 3$
(d) if $1 + 1 == 2$ then $3$ else $4 + 5$

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

\[
\begin{align*}
\vdash e &: \tau \\
\vdash n &: \int & \Rightarrow \vdash e_1 &: \int & \vdash e_2 &: \int & \vdash e_1 &: \int & \vdash e_2 &: \int \\
\vdash b &: \bool & \Rightarrow \vdash e_1 &: \tau & \vdash e_2 &: \tau & \vdash e &: \bool & \vdash e_1 &: \tau & \vdash e_2 &: \tau \\
\vdash b &: \bool & \Rightarrow \vdash e_1 &: \tau & \vdash e_2 &: \tau & \vdash e &: \bool & \vdash e_1 &: \tau & \vdash e_2 &: \tau \\
\end{align*}
\]

Write out typing derivations for the following judgments:
(a) $\vdash 6 \times 9 : \int$
(b) $\vdash \text{if } 1 + 1 == 2 \text{ then } 3 \text{ else } 4 + 5 : \int$

4. (⋆) **Nondeterminism.** Suppose we add the following construct $e_1 \square e_2$ to $\mathcal{L}_\text{Arith}$:

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

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:

\[
\begin{align*}
\[ e \downarrow v \]
\end{align*}
\]

(a) What property of $\mathcal{L}_\text{Arith}$ (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?)