Exercises marked * are more advanced. Please try all unstarred exercises before the tutorial meeting.

1. **Typing derivations**
   Construct typing derivations for the following expressions, or argue why they are not well-formed:
   
   (a) $\Lambda A.\lambda x: A. x + 1$
   (b) $\star (\Lambda A.\lambda x: A \times A. \text{if} \; \text{fst} \; x \; == \; \text{snd} \; x \; \text{then} \; \text{fst} \; x \; \text{else} \; \text{snd} \; x)$ (and how does its well-formedness depend on the typing rule for equality?)

2. **Evaluation derivations**
   Construct evaluation derivations for the following expressions, or explain why they do not evaluate:
   
   (a) $(\Lambda A.\lambda x: A. x + 1)[\text{int}] \; 42$
   (b) $(\Lambda A.\lambda x: A. x + 1)[\text{bool}] \; \text{true}$

3. ** Lists and polymorphism**
   Recall the proposed rules for lists from the previous tutorial.

   $e ::= \cdots \mid \text{nil} \mid e_1 :: e_2 \mid \text{case} \; \text{list} \; e \; \text{of} \{ \text{nil} \Rightarrow e_1 ; \; x :: y \Rightarrow e_2 \}$

   $v ::= \cdots \mid \text{nil} \mid v_1 :: v_2$

   $\tau ::= \cdots \mid \text{list} [\tau]$

   Define $\text{L}_{\text{list}}$ to be $\text{L}_{\text{poly}}$ extended with the above constructs.
   
   (a) Write a polymorphic function $\text{map}$ that has this type:

   $\forall A.\forall B. (A \to B) \to (\text{list}[A] \to \text{list}[B])$

   so that $\text{map}(f)(l)$ is the function that traverses a list of $A$’s and, for each element $x$ in $l$, applies the function $f$ to it.

   (b) Write out a typing derivation tree for the expression

   $\text{map}[\text{int}][\text{int}][\lambda x. x + 1](2 :: \text{nil})$

   assuming that $\text{map}$ has the type given above.

   (c) Are lists and their associated operations definable in $\text{L}_{\text{poly}}$ already? Why or why not?
4. Modules and Interfaces in Scala

Consider the following Scala object definition.

```scala
object A {
  type T = Int
  val c: T = 1
  val d: T = 2
  def f(x: T, y: T): T = x + y
}
object B {
  type T = String
  val c: T = "abcd"
  val d: T = "1234"
  def f(x: T, y: T) = x + y
}
```

(a) Write expressions showing how to access each of the elements of A and B.

(b) Suppose we execute the import statements

```
import A._
import B._
```

after finishing the declaration of A. What does unqualified identifier d refer to after that? What if we import in the opposite order?

(c) (+) Construct a Scala trait ABlike defining bindings for all of the components of A and B, and so that we can assert that both A and B extend ABlike.

(d) (+) Define a function g taking an argument x: ABlike that applies f to c and d. Apply it to both instances of ABlike above. What is its return type?

(e) (+) Create an anonymous instance of ABlike with T = Boolean and call the function g on it.

5. (+) Ad hoc polymorphism

Traits can also accommodate overloading and reuse of the same name for operations on different types. An operation such as size can be defined as part of a trait as follows:

```scala
trait HasSize { def size(): Int }
```

(a) Modify the definition of List[A] above so that it extends HasSize, and define an appropriate size method for it.

(b) Modify the definition of Tree[A] so that it extends HasSize and define its size operation.

(c) Write a function sameSize that takes two values of type HasSize and checks whether they have the same size.

(d) Call this function on a List[Int] and a Tree[String] to verify that the correct implementations of size are called for different types.