159.272 Programming Paradigms

Tutorial 3
Higher-Order Functions and Algebraic Types

In this tutorial we work through some exercises using both higher-order functions and user-defined algebraic types. Use the file **tutorial3.scala** as a template to demonstrate your results. This file provides you with correct type annotations for the functions you are asked to define and the required algebraic data types (case classes) for questions 2 and 3.

1. Use appropriate constructs to define the following higher-order functions:

   a. Define a function **applyEach**, that, given a list of functions and a value, applies each function to the given value and returns a list of results.
   Eg: **applyEach(List((x=>x*x),(x=>x+1),(x=>List.fill(3)(x))),5)** should return **List(25,6,List(5,5,5))**.

   ```
   def applyEach(fnList:List[(Int=>Any)], x:Int): List[Any] =
   ```

   b. Consider the function **twice** that takes a function and an argument and applies the function twice to the argument.
   For example: **twice((x:Int) => x+5, 1)** should return **11**.

   ```
   def twice[T](fn:(T=>T), x:T): T =
     {
       fn(fn(x))
     }
   ```

   Generalise **twice** to **iter**, a function that takes a function, an integer **n**, and an argument, and applies the function n times to the argument.

   ```
   def iter[T](fn:(T=>T), n:Int, x:T): T =
   ```

   c. Now consider the function **liftTwice** that takes a function returns a **composition** of two copies of the function, so the return value of **liftTwice** is a new function that applies the original function twice to an argument.
   For example: **liftTwice((x:Int) => x+5) (1)** should return **11**.

   ```
   def liftTwice[T](fn:(T=>T)): (T => T) =
     {
       fn compose fn
     }
   ```

   Generalise **liftTwice** to **liftIter**, a function that takes a function and an integer **n**, and returns a **composition** of **n** copies of the function. What should be returned when **n** = 0?

   ```
   def liftIter[T](fn:(T=>T), n:Int): (T => T) =
   ```

2. In lectures we defined an algebraic data type for arithmetic expressions.

   ```
   sealed abstract class Expr
   case class Var(name:String) extends Expr
   case class Number(num:Double) extends Expr
   case class UnOp(operator:String, arg:Expr) extends Expr
   case class BinOP(operator:String, left:Expr, right:Expr) extends Expr
   ```

a. Define a function **exprSize** that computes the size of an Expr value, that is, the total number of variables, numbers and operators in the expression.

```
def exprSize(e: Expr):Int =
```

b. Define a function **exprToString** that returns a nicely formatted string version of an Expr value. For example **exprToString(UnOp("-", Var("x")))** could return **-(x)**.

```
def exprSize(e: Expr):String =
```

3. In the game of Bridge, four players in two competing partnerships are each dealt a hand of 13 cards. Each player must evaluate his hand in order to decide on the correct opening bid to make in the auction for the contract. A bid is either **Pass** or consists of a **level**, which is a number between 1 and 7, and a **trump suit**, which is any of the four suits or No Trumps.

   The first step when determining the correct bid is to add up the High Card Points (HCP) in the hand, 4 for an Ace, 3 for a King, 2 for a Queen, 1 for a Jack.

   The following is a (somewhat simplified) set of rules that should be followed to decide on an opening bid under the Standard American bidding system:

   - 7-11 HCP and 7 or more cards in one suit: bid 3 of that suit
   - otherwise, with fewer than 12 HCP: Pass
   - 12-15 points and 5+ Spades or 5+ Hearts: bid 1 Spade or 1 Heart, if both are possible, bid 1 Spade
   - otherwise, 12-15 HCP and 4+ Diamonds: bid 1 Diamond
   - otherwise, 12-15 HCP and none of above: bid 1 Club
   - 16-18 HCP and a balanced hand (at least three cards in each of three suits, at least two cards in the fourth suit): bid 1 NoTrump
   - otherwise, with an unbalanced hand, same as for 12-15 HCP
   - 20-22 HCP and a balanced hand: bid 2 NoTrump
   - otherwise, 19-21 HCP: bid 2 Club, 22+ HCP: bid 2 Diamond

   Using the algebraic data types (case classes) provided in **tutorial3.scala**, define a function

   ```
   def openingBid(myHand:Hand):Bid =
   ```

   that takes a hand of 13 cards as input and outputs a correct opening bid according to the rules given here. You can define any auxiliary functions that you deem necessary.