Homework 7

Due date: Tuesday, November 9, 2015

The primary purpose of this assignment is to practice the use of higher-order functions and collections. Concretely, we will extend JakartaScript with multi-parameter functions. We will use Scala's list data structure to represent parameter lists in the AST representation of function expressions. Before you extend your interpreter, you will practice the use of higher-order functions and working with the Scala collection API.

Like last time, you will work on this assignment in pairs. However, note that each student needs to submit a write-up and are individually responsible for completing the assignment. You are welcome to talk about these exercises in larger groups. However, we ask that you write up your answers in pairs. Also, be sure to acknowledge those with which you discussed, including your partner and those outside of your pair.

Try to make your code as concise and clear as possible. Challenge yourself to find the most crisp, concise way of expressing the intended computation. This may mean using ways of expressing computation currently unfamilar to you.

Finally, make sure that your file compiles and runs (using Scala 2.11.7). A program that does not compile will *not* be graded.

Submission instructions. Upload to NYU classes exactly one file named as follows:

- `hw07-YourNYULogin.scala` with your answers to the coding exercises.

Replace `YourNYULogin` with your NYU login ID (e.g., I would submit `hw07-tw47.scala`). To help with managing the submissions, we ask that you rename your uploaded files in this manner.

Getting started. Download the code pack `hw07.zip` from the assignment section on the NYU classes page.

Problem 1 Collection and Higher-Order Functions (25 Points)

To implement our interpreter for JakartaScript with multi-parameter functions, we will need to make use of collections from Scala's library. One of the most fundamental operations that one needs to perform with a collection is to iterate over the elements of the collection. Like many other languages with function expressions (e.g., Python, ML, Haskell), the Scala library provides various iteration operations via *higher-order functions*. Higher-order functions are functions that take functions as parameters. The function parameters are often called *callbacks*, and for collections, they typically specify what the library client wants to do for each element.

In this problem, we practice both writing such higher-order functions in a library and using them as a client.

(a) Implement a function

```
def compressRec[A](l: List[A]): List[A]
```

that eliminates consecutive duplicates of list elements. If a list contains repeated elements they should be replaced with a single copy of the element. The order of the elements should not be changed.

Example:

```
scala> compressRec(List(1, 2, 2, 3, 3, 3))
res0: List[Int] = List(1, 2, 3)
```

This test has been provided for you in the template. For this exercise, implement the function by direct recursion (e.g., pattern match on `l` and call `compressRec` recursively). Do not call any methods from the `List` class in the standard library. This exercise is from Ninety-Nine Scala Problems:

`http://aperiodic.net/phil/scala/s-99/`.

Some sample solutions are given there, which you are welcome to view. However, it is strongly encouraged that you first attempt this exercise before looking there. The purpose of the exercise is to get some practice for the later part of this homework. Note that the solutions there do not satisfy the requirements here (as they use library functions). If at some point you feel like you need more practice with collections, the above page is a good resource.

(b) Reimplement the compress function from the previous part as `compressFold` using the `foldRight` method from the `List` class in the standard library. The call to `foldRight` has been provided for you. Do not call `compressFold` recursively or any other `List` methods.

(c) Implement a higher-order recursive function

```
  def mapFirst[A](f: A => Option[A])(l: List[A]): List[A]
```

that finds the first element in `l` where `f` applied to it returns a `Some(a)` for some value `a`. It should replace that element with `a` and leave `l` the same everywhere else. Example:

```
scala> mapFirst((i: Int) => if (i < 0) Some(-i) else None)(List(1,2,-3,4,-5))
res0: List[Int] = List(1, 2, 3, 4, -5)
```

(d) Consider again the binary search tree data structure from Homework 2:

```
sealed abstract class Tree {
  def insert(n: Int): Tree = this match {
    case Empty => Node(Empty, n, Empty)
    case Node(l, d, r) =>
    if (n < d) Node(l insert n, d, r) else Node(l, d, r insert n)
  }
  def foldLeft[A](z: A)(f: (A, Int) => A): A = {
    def loop(acc: A, t: Tree): A = t match {
      case Empty => ???
      case Node(l, d, r) => ???
    }
    loop(z, this)
    }
  }
  case object Empty extends Tree
  case class Node(l: Tree, d: Int, r: Tree) extends Tree
```

Here, we have implemented the binary search tree operation `insert` as a method of `Tree`. For this exercise, complete the higher-order method `foldLeft`. This method performs an in-order traversal of the input tree **`this`** calling the callback `f` to accumulate a result. Suppose the in-order traversal of the input tree yields the following sequence of data values: $d_1, d_2, \ldots, d_n$. Then, `foldLeft` yields

$$\mathtt{f}(\ldots(\mathtt{f}(\mathtt{f}(\mathtt{z}, d_1), d_2))\ldots), d_n) \ .$$

We have provided a test client `sum` that computes the sum of all of the data values in the tree using your `foldLeft` method.

(e) Implement a function

```
    def strictlyOrdered(t: Tree): Boolean
```

as a client of your `foldLeft` method that checks whether the data values of `t` as an in-order traversal are in strictly ascending order (i.e., $d_1 < d_2 < \cdots < d_n$).

Example:

```
scala> strictlyOrdered(treeFromList(List(1,1,2)))
res0: Boolean = false
```

Problem 2 JakartaScript with Multi-Parameter Functions (15 Points)

In this exercise, we extend JakartaScript with multi-parameter functions (see Figure 1). In Figure 2, we show the updated AST representation. We update `Function` and `Call` for multiple parameters and arguments, respectively.

From now on, we will use big-step semantics to formalize the intended behavior of our JakartaScript interpreter as it matches more naturally the interpreter implementation. We will implicitly assume that the subexpressions of binary operators, call expressions, etc. are evaluated in left-to-right order, even if this evaluation order is not explicitly captured by the rules of our big-step semantics. Furthermore, from now on, we will focus on static binding semantics.

The semantics of our new version of JakartaScript is specified by the rules in figures 3, 4, and 5. The rules in figures 3 and 5 have already been implemented for you in the template since they are very similar to the previous homework assignments. Your task is to implement the remaining rules in Figure 4, which specify the behavior of function calls with multiple arguments.

The modifications that we need to make to the rules for single argument function calls that we considered so far are for the most part straightforward. The only complication arises from the fact that JavaScript allows function calls to be evaluated, even if the number of arguments provided in the call does not match the number of parameters of the called

$n \in \mathit{Num}$	numbers (double)
$s \in \mathit{Str}$	strings
$x \in \mathit{Var}$	variables
$b \in \mathit{Bool} ::= \textbf{true} \mid \textbf{false}$	Booleans
$v \in \mathit{Val} ::= \textbf{undefined} \mid n \mid b \mid s \mid \textbf{function}\ p\,(x_1,\ldots,x_n)\,e \mid$ typeerror	values
$e \in \mathit{Expr} ::= x \mid v \mid \mathit{uop}\ e \mid e_1\,\mathit{bop}\,e_2 \mid e_1\ ?\ e_2\ \textbf{:}\ e_3 \mid \textbf{const}\ x = e_1; e_2 \mid \textbf{console.log}(e) \mid e\,(e_1,\ldots,e_n)$	expressions
$\mathit{uop} \in \mathit{Uop} ::= - \mid\ !$	unary operators
$\mathit{bop} \in \mathit{Bop} ::= + \mid - \mid * \mid / \mid \texttt{===} \mid \texttt{!==} \mid < \mid > \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{\&\&} \mid \texttt{\|\|} \mid \texttt{,}$	binary operators
$p ::= x \mid \epsilon$	function names

Figure 1: Abstract syntax

```
sealed abstract class Expr extends Positional
abstract class Val extends Expr
...
case class Function(p: Option[String], xs: List[String], e: Expr) extends Val
```
`Function(`p`, List(`$x_1,\ldots,x_n$`), `e`)` **function** $p\,(x_1,\ldots,x_n)\,e$

```
case class Call(e: Expr, es: List[Expr]) extends Expr
```
`Call(`e`, List(`$e_1,\ldots,e_n$`))` $e\,(e_1,\ldots,e_n)$

Figure 2: Representing in Scala the abstract syntax of JakartaScript. After each **case class** or **case object**, we show the correspondence between the representation and the concrete syntax.

function. Specifically, if the number of arguments in a call expression is greater than the number of parameters of the called function, then the auxiliary arguments are simply ignored when the call is evaluated. For example, consider the following program:

```
const f = function(x) { return x; };
f(1, 2 + 3)
```

This program defines a function `f` that takes a single parameter `x`. However, the call `f(1, 2 + 3)` then calls `f` with two arguments. The first argument `1` is passed to the parameter `x` of `f` to evaluate the body of `f` and compute the result of the call. The value obtained from the second argument `2 + 3` is simply ignored. Note however that all arguments in a function call expression are evaluated, regardless of whether they are actually passed to a function parameter or whether they are ignored. Thus, in the above program, the auxiliary argument `2 + 3` is still evaluated, but its result is discarded. The general case where the number of arguments in the call is greater or equal to the number of function parameters is described by the rules EvalCall$_{\geq}$ and EvalCallRec$_{\geq}$.

If the number of arguments in a call expression is smaller than the expected number of parameters of the called function, then the missing arguments are simply set to the value **undefined**. For example, consider the following program:

```
const f = function(x, y) { return y; };
f(1)
```

The function `f` takes two parameters `x` and `y` and simply returns the value of `y`. The call `f(1)` only provides an argument for the parameter `x`. When this call is evaluated, the parameter `y` is set to **`undefined`**. Hence, the whole program evaluates to **`undefined`**. The general case where the number of arguments in a call expression is smaller than the number of function parameters is described by the rules $\textsc{EvalCall}_<$ and $\textsc{EvalCallRec}_<$.

Complete the functions `subst` and `eval` provided in the code package. We suggest the following step-by-step order:

(a) First, bring over the implementation of `subst` from Homework 6. Modify the cases for `Call` and `Function` expressions to account for multiple parameters/arguments. **Hint:** You will find the methods `map` and `contains` or `exists` in Scala's `List` class useful.

(b) Then, work on the missing `Call` cases in `eval`. **Hint:** Helpful library methods here include `map`, `foldRight`, `zipped`, and `padTo`. The calls to some of theses are already provided for you.

Problem 3 Pretty Printing Calendars (voluntary, 10 Bonus Points)

If you would like to get more practice using the Scala collections API, we suggest to solve this voluntary problem.

In this problem we are interested in printing a calendar using Scala. More specifically, we want to print an overview of a given month that shows which date falls on which day of the week. For example, in 2015, the First of April is a Wednesday. The month of April 2015 should be printed as follows:

```
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30
```

Before you start solving this problem, make yourself familiar with the methods provided by the `List` class in the Scala standard API. All parts of this problem have very short solutions if you use the appropriate functions provided by class `List`. As a hint, the code template lists the functions that we used in the sample solution and the lines of code needed to implement each function. You may want to use different functions than the ones we suggested, but try to make your solution as concise as possible. The sample solution does not use any loops and does not define any new recursive functions. You should also try to do without these.

$$\frac{}{v \Downarrow v}\ \textsc{EvalVal} \qquad \frac{e_1 \Downarrow v_1 \quad \textbf{true} = \mathit{toBool}(v_1) \quad e_2 \Downarrow v_2}{e_1 \;\&\&\; e_2 \Downarrow v_2}\ \textsc{EvalAndTrue}$$

$$\frac{e_1 \Downarrow v_1 \quad \textbf{false} = \mathit{toBool}(v_1) \quad e_2 \Downarrow v_2}{e_1 \;||\; e_2 \Downarrow v_2}\ \textsc{EvalOrFalse}$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 \,\texttt{,}\, e_2 \Downarrow v_2}\ \textsc{EvalSeq} \qquad \frac{e_1 \Downarrow v_1 \quad \textbf{false} = \mathit{toBool}(v_1)}{e_1 \;\&\&\; e_2 \Downarrow v_1}\ \textsc{EvalAndFalse}$$

$$\frac{e \Downarrow v \quad v \text{ printed}}{\textbf{console.log}\,(e) \Downarrow \textbf{undefined}}\ \textsc{EvalPrint} \qquad \frac{e_1 \Downarrow v_1 \quad \textbf{true} = \mathit{toBool}(v_1)}{e_1 \;||\; e_2 \Downarrow v_1}\ \textsc{EvalOrTrue}$$

$$\frac{e \Downarrow v \quad v' = -\,\mathit{toNum}(v)}{-\,e \Downarrow v'}\ \textsc{EvalUMinus} \qquad \frac{e \Downarrow v \quad v' = \,!\,\mathit{toBool}(v)}{!\,e \Downarrow v'}\ \textsc{EvalNot}$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v = \mathit{toNum}(v_1) + \mathit{toNum}(v_2) \quad v_1, v_2 \notin \mathit{Str}}{e_1 + e_2 \Downarrow v}\ \textsc{EvalPlusNum}$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v = v_1 + \mathit{toString}(v_2) \quad v_1 \in \mathit{Str}}{e_1 + e_2 \Downarrow v}\ \textsc{EvalPlusStr}_1$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v = \mathit{toString}(v_1) + v_2 \quad v_2 \in \mathit{Str}}{e_1 + e_2 \Downarrow v}\ \textsc{EvalPlusStr}_1$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v = \mathit{toNum}(v_1)\, \mathit{bop}\, \mathit{toNum}(v_2) \quad \mathit{bop} \in \{*, /, -\}}{e_1\, \mathit{bop}\, e_2 \Downarrow v}\ \textsc{EvalArith}$$

$$\frac{e_d \Downarrow v_d \quad e_b[v_d/x] \Downarrow v_b}{\textbf{const}\ x = e_d\,\texttt{;}\, e_b \Downarrow v_b}\ \textsc{EvalConst}$$

$$\frac{e_1 \Downarrow v_1 \;\; e_2 \Downarrow v_2 \;\; v = \mathit{toNum}(v_1)\ \mathit{bop}\ \mathit{toNum}(v_2) \;\; v_1 \notin \mathit{Str} \;\; \mathit{bop} \in \{>, >=, <, <=\}}{e_1\ \mathit{bop}\ e_2 \Downarrow v}\ \textsc{EvalInequalNum}_1$$

$$\frac{e_1 \Downarrow v_1 \;\; e_2 \Downarrow v_2 \;\; v = \mathit{toNum}(v_1)\ \mathit{bop}\ \mathit{toNum}(v_2) \;\; v_2 \notin \mathit{Str} \;\; \mathit{bop} \in \{>, >=, <, <=\}}{e_1\ \mathit{bop}\ e_2 \Downarrow v}\ \textsc{EvalInequalNum}_2$$

$$\frac{e_1 \Downarrow s_1 \quad e_2 \Downarrow s_2 \quad v = s_1\ \mathit{bop}\ s_2 \quad \mathit{bop} \in \{>, >=, <, <=\}}{e_1\ \mathit{bop}\ e_2 \Downarrow v}\ \textsc{EvalInequalStr}$$

$$\frac{\begin{array}{c} e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad b = (v_1\ \mathit{bop}\ v_2) \\ v_1 \neq \textbf{function}\, p_1\,(\overline{x})\, e_3 \quad v_1 \neq \textbf{function}\, p_2\,(\overline{x})\, e_4 \quad \mathit{bop} \in \{===, !==\} \end{array}}{e_1\ \mathit{bop}\ e_2 \Downarrow b}\ \textsc{EvalEqual}$$

$$\frac{e_1 \Downarrow v_1 \quad \mathit{toBool}(v_1) = \textbf{true} \quad e_2 \Downarrow v_2}{e_1\ ?\ e_2\ \texttt{:}\ e_3 \Downarrow v_2}\ \textsc{EvalIfThen}$$

$$\frac{e_1 \Downarrow v_1 \quad \mathit{toBool}(v_1) = \textbf{false} \quad e_3 \Downarrow v_3}{e_1\ ?\ e_2\ \texttt{:}\ e_3 \Downarrow v_3}\ \textsc{EvalIfElse}$$

Figure 3: Big-step operational semantics of JakartaScript: non-call rules

$$\frac{e_0 \Downarrow \textbf{function}\,(x_1,\ldots,x_m)\,e \quad n < m \quad e_1 \Downarrow v_1 \quad \ldots \quad e_n \Downarrow v_n \qquad e' = e[v_1/x_1]...[v_n/x_n][\textbf{undefined}/x_{n+1}]...[\textbf{undefined}/x_m] \quad e' \Downarrow v}{e_0\,(e_1,\ldots,e_n) \Downarrow v}\ \textsc{EvalCall}_<$$

$$\frac{e_0 \Downarrow \textbf{function}\,(x_1,\ldots,x_m)\,e \quad n \geq m \quad e_1 \Downarrow v_1 \quad \ldots \quad e_n \Downarrow v_n \qquad e' = e[v_1/x_1]...[v_m/x_m] \quad e' \Downarrow v}{e_0\,(e_1,\ldots,e_n) \Downarrow v}\ \textsc{EvalCall}_\geq$$

$$\frac{e_0 \Downarrow v_0 \quad v_0 = \textbf{function}\ x_0\,(x_1,\ldots,x_m)\,e \quad n < m \quad e_1 \Downarrow v_1 \quad \ldots \quad e_n \Downarrow v_n \qquad e' = e[v_0/x_0]...[v_n/x_n][\textbf{undefined}/x_{n+1}]...[\textbf{undefined}/x_m] \quad e' \Downarrow v}{e_0\,(e_1,\ldots,e_n) \Downarrow v}\ \textsc{EvalCallRec}_<$$

$$\frac{e_0 \Downarrow v_0 \quad v_0 = \textbf{function}\ x_0\,(x_1,\ldots,x_m)\,e \quad n \geq m \quad e_1 \Downarrow v_1 \quad \ldots \quad e_n \Downarrow v_n \qquad e' = e[v_0/x_0]...[v_m/x_m] \quad e' \Downarrow v}{e_0\,(e_1,\ldots,e_n) \Downarrow v}\ \textsc{EvalCallRec}_\geq$$

Figure 4: Big-step operational semantics of JakartaScript: call rules

$$\frac{e_1 \Downarrow \textbf{function}\ p\,(\overline{x})\,e \quad bop \in \{\texttt{===}, \texttt{!==}\}}{e_1\ bop\ e_2 \Downarrow \mathsf{typeerror}}\ \textsc{EvalTypeErrorEqual}_1$$

$$\frac{e_2 \Downarrow \textbf{function}\ p\,(\overline{x})\,e \quad bop \in \{\texttt{===}, \texttt{!==}\}}{e_1\ bop\ e_2 \Downarrow \mathsf{typeerror}}\ \textsc{EvalTypeErrorEqual}_2$$

$$\frac{e_0 \Downarrow v_0 \quad v_0 \neq \textbf{function}\ p\,(\overline{x})\,e}{e_0\,(e_1,\ldots,e_n) \Downarrow \mathsf{typeerror}}\ \textsc{EvalTypeErrorCall}$$

$$\frac{e_1 \Downarrow \mathsf{typeerror}}{e_1\ \texttt{?}\ e_2\ \texttt{:}\ e_3 \Downarrow \mathsf{typeerror}}\ \textsc{EvalPropIf} \qquad \frac{e_1 \Downarrow \mathsf{typeerror}}{e_1\ bop\ e_2 \Downarrow \mathsf{typeerror}}\ \textsc{EvalPropBop}_1$$

$$\frac{e \Downarrow \mathsf{typeerror}}{uop\ e \Downarrow \mathsf{typeerror}}\ \textsc{EvalPropUop} \qquad \frac{e_2 \Downarrow \mathsf{typeerror} \quad bop \notin \{\texttt{\&\&}, \texttt{||}\}}{e_1\ bop\ e_2 \Downarrow \mathsf{typeerror}}\ \textsc{EvalPropBop}_2$$

$$\frac{e_1 \Downarrow v_1 \quad v_1 \neq \mathsf{typeerror} \quad \textbf{true} = toBool(v_1) \quad e_2 \Downarrow \mathsf{typeerror}}{e_1\ \texttt{\&\&}\ e_2 \Downarrow \mathsf{typeerror}}\ \textsc{EvalPropAnd}$$

$$\frac{e_1 \Downarrow v_1 \quad v_1 \neq \mathsf{typeerror} \quad \textbf{false} = toBool(v_1) \quad e_2 \Downarrow \mathsf{typeerror}}{e_1\ \texttt{||}\ e_2 \Downarrow \mathsf{typeerror}}\ \textsc{EvalPropOr}$$

$$\frac{e \Downarrow \mathsf{typeerror}}{\textbf{console.log}\,(e) \Downarrow \mathsf{typeerror}}\ \textsc{EvalPropPrint} \qquad \frac{e_1 \Downarrow \mathsf{typeerror}}{e_1\,(e_2) \Downarrow \mathsf{typeerror}}\ \textsc{EvalPropCall}_1$$

$$\frac{e_d \Downarrow \mathsf{typeerror}}{\textbf{const}\ x = e_d\,\textbf{;}\ e_b \Downarrow \mathsf{typeerror}}\ \textsc{EvalPropConst} \qquad \frac{e_2 \Downarrow \mathsf{typeerror}}{e_1\,(e_2) \Downarrow \mathsf{typeerror}}\ \textsc{EvalPropCall}_2$$

Figure 5: Big-step operational semantics of JakartaScript: dynamic type error rules

Warm-Up

Define a function `unlines` that turns a list of lists of characters into a list of characters inserting a `\n` character between each two lists. The following example illustrates the function `unlines`:

```
unlines(List(List('f','e','i','s','t','y'),List('f','a','w','n')))
```

should yield

```
List('f','e','i','s','t','y','\n','f','a','w','n')
```

Leap years, the First of January and all that

To be able to print a monthly overview, we first have to determine on which weekday falls the first day of the given month. We provide you with the following function definitions to simplify this task:

```
/** The weekday of January 1st in year y, represented
  * as an Int. 0 is Sunday, 1 is Monday etc. */
def firstOfJan(y: Int): Int = {
  val x = y - 1
  (365*x + x/4 - x/100 + x/400 + 1) % 7
}

def isLeapYear(y: Int) =
  if (y % 100 == 0) (y % 400 == 0) else (y % 4 == 0)

def mlengths(y: Int): List[Int] = {
  val feb = if (isLeapYear(y)) 29 else 28
  List(31, feb, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
}
```

With the help of these functions, define a function `firstDay` that calculates the weekday of the first day of a given month:

```
def firstDay(month: Int, year: Int): Int = ???
```

How to picture that?

Picturing data with a non-trivial layout such as a calendar can be tricky. Therefore, we want to use a compositional approach where larger, more complex pictures are composed of smaller, simpler pictures.

In our design, pictures are represented as instances of the `Picture` case class:

```
case class Picture(height: Int, width: Int, pxx: List[List[Char]]) {
  def showIt: String = unlines(pxx).mkString("")
}
```

As we can see, a picture has a height and width, and contents `pxx` which is character data represented as a list of rows, where each row is a list of characters. The `showIt` method turns the picture into a list of characters using the `unlines` function defined in the first part. The following function `pixel` creates a simple picture of height and width 1 that contains a given character:

```
def pixel(c: Char) = Picture(1, 1, List(List(c)))
```

From pictures as simple as that, we want to compose larger ones using composition operators.

(a) Define a method `above` for class `Picture` that returns a new picture where the argument picture is put below **`this`**:

```
case class Picture(...) {
  def above(q: Picture): Picture = ???
}
```

For instance, the following code

```
println((pixel('a') above pixel('b')).showIt)
```

should print

```
a
b
```

(b) Define a method `beside` for class `Picture` that returns a new picture where the argument picture is put on the right side of this:

```
case class Picture(...) {
  def beside(q: Picture): Picture = ???
}
```

(c) Define functions `stack` and `spread` that arrange a list of pictures above and beside each other, respectively, producing a single resulting picture. For `stack`, the picture at the head of the argument list should be the topmost picture in the result. Similarly for `spread`, the head of the list should be the leftmost picture in the result.

```
def stack(pics: List[Picture]): Picture = ???
def spread(pics: List[Picture]): Picture = ???
```

(d) Define a function `tile` that arranges a list of rows of pictures in a rectangular way using the `stack` and `spread` functions:

```
def tile(pxx: List[List[Picture]]): Picture = ???
```

(e) Define a function that takes a width `w` and a list of characters, and produces a picture of height 1 and width `w` where the given characters are justified on the right border:

```
def rightJustify(w: Int)(chars: List[Char]): Picture = ???
```

(f) Define a function `group` that splits a list into sublists. The function takes an integer as argument that indicates the split indices (e.g. split every 7 elements). We intend to use this function to split a list representing a whole month into a list of weeks. Note that this function is parameterized which means that it can be used with lists of any element type.

```
def group[T](n: Int, xs: List[T]): List[List[T]] = ???
```

(g) Define a function `dayPics` that takes the number of the first day and the number of days of a month and produces a list of 42 pictures. In this list, the first `d` pictures are empty (i.e., the character data is a list of spaces) if the number of the first day is `d` (`d==0`: Sunday, `d==1`: Monday, etc.). The trailing pictures that correspond to days of the next month are empty, too. Using this function, a picture of a calendar can be produced by grouping and tiling the result of `dayPics`.

```
def dayPics(d: Int, s: Int): List[Picture] = ???
```

Hint: A Scala string can be converted to a list of characters by calling its `toList` method. This might come in handy when converting days to lists of characters.

(h) Using the functions defined in the previous steps, define a function `calendar` that produces a picture of a calender that corresponds to the given year and month.

```
def calendar(year: Int, month: Int): Picture = ???
```