Functional Programming

Final Exam

Friday, December 22 2017

Your points are *precious*, don't let them go to waste!

Your Time All points are not equal. Note that we do not think that all exercises have the same difficulty, even if they have the same number of points.

Your Attention The exam problems are precisely and carefully formulated: some details can be subtle. Pay attention, because if you do not understand a problem, you can not obtain full points.

Stay Functional You are strictly forbidden to use return statements, mutable state (vars) and mutable collections in your solutions.

Some Help The last pages of this exam contains an appendix which is useful for formulating your solutions. You can detach these pages and keep them aside.

Using APIs Unless otherwise noted, you are always allowed to use methods of the Scala API that you know, even if they are not listed in the appendix. However, for Lisp, you may not use non-operator functions that are not listed in the appendix, unless you define them.

Exercise	**Points**	**Points Achieved**
1	10	
2	10	
3	10	
Total	30	

Exercise 1: Constraint Solving (10 points)

You have been hired by a real estate developer to help plan the development of a beach on a tropical island somewhere in the Pacific[1].

The real estate developer tells you that, to attract tourists, he plans to build a series of buildings on the beach. Buildings are to be arranged on a single line along the beach, so that every building is facing the sea! Every single slot of land must be occupied by exactly one building.

Your goal is to come up with a set of constraints to help him build the perfect beach resort. You must use the following types and values in your solution.

```
sealed abstract class BuildingSort
case object ChangingRoom extends BuildingSort
case object LifeGuardTower extends BuildingSort
case object Restaurant extends BuildingSort
case object Shop extends BuildingSort
case object Toilets extends BuildingSort

val allBuildingSorts: List[BuildingSort] =
  List(ChangingRoom, LifeGuardTower, Restaurant, Shop, Toilets)
val allSlots: List[Int] =
  (0 until n).toList  // Where n is the number of slots.
```

The following propositional variables encode the fact that a slot hosts a building of the given sort.

```
val vars: Map[(Int, BuildingSort), Formula] =
  allSlots.flatMap(s => allBuildingSorts.flatMap(b => (s, b) -> propVar())).toMap
```

[1] Unfortunately, this is a remote job...

Exercise 1.1

Write down the formula that specifies that there is a changing room at the first or at the last slot.

```
val correctChangingRoomLocation: Formula =
```

Exercise 1.2

Write down the formula that specifies there is at least one life guard tower along the beach.

```
val atLeastOneLifeGuardTower: Formula =
```

Exercise 1.3

Write down the formula that specifies that no two shops may be adjacent to each other.

```
val noTwoAdjacentShops: Formula =
```

Exercise 1.4

Write down the formula that specifies that restaurants must have toilets on at least one of the adjacent slots.
Hint: Be extra careful at the first and last slots.

```
val restaurantsAjacentToilets: Formula =
```

Exercise 1.5

Write down the formula that specifies that each slot is occupied by *one and exactly one* building.

```
val exactlyOneBuildingPerSlot: Formula =
```

Exercise 2: Lisp (10 points)

After having written your Lisp interpreter, you decide that you want to implement some transformations over your Lisp programs. The first transformation you consider is partial derivation of mathematical formulas. Namely, you want terms of the shape `(∂ x expr)` to be transformed to some new term `expr2` that corresponds to the derived formula of `expr`. `expr2` does NOT need to be simplified.

`expr2` $= \frac{\partial}{\partial x}$`expr`

For example, the term `(∂ x (+ y (+ 2 x)))` should be rewritten to `(+ 0 (+ 0 1))`.

You can assume the expr only contains numbers, symbols, sums `(+ e1 e2)` and multiplications `(* e1 e2)`.

Rules for partial derivation

$\frac{\partial}{\partial x} x = 1$

$\frac{\partial}{\partial x} n = 0$ **for** $n \in \mathbb{Z}$

$\frac{\partial}{\partial x} y = 0$

$\frac{\partial}{\partial x} \left(f(x) + g(x)\right) = \frac{\partial}{\partial x} f(x) + \frac{\partial}{\partial x} g(x)$

$\frac{\partial}{\partial x} \left(f(x) \cdot g(x)\right) = \left(\frac{\partial}{\partial x} f(x) \cdot g(x)\right) + \left(f(x) \cdot \frac{\partial}{\partial x} g(x)\right)$

Exercise 2.1: Scala implementation

In order to achieve the above task, complete the following function:

```
def derive(x: Symbol, expr: Any): Any = ???
```

Note that in the example above, you would call

```
derive('x, List('+, 'y, List('+, 2, 'x)))
```

and the expected result is

```
List('+, 0, List('+, 0, 1))
```

Remember that Lisp terms are of type `Any`. Composite terms have type `List[Any]`, variable names are instances of `Symbol` (you can use `==` between symbols) and numbers are of type `Int`. You may use any method in the Scala standard library (see appendix for most typical ones).

Hint: Remember that you can use a default case in a pattern match.

```
def derive(x: Symbol, expr: Any): Any =
```

Exercise 2.2: Translation into Lisp

You now want to implement the same functionality within the Lisp language itself. This means **your input now consists of a Lisp list**. Complete the following function definition (using the syntactic sugar proposed in the assignment):

```
(def (derive x expr) ??? rest)
```

The example above would this time correspond to

```
(derive 'x (list '+ 'y (list '+ 2 'x)))
```

and your function should output

```
(list '+ 0 (list '+ 0 1))
```

The available Lisp API can be found in the appendix. You may use the predicates `isCons?` and `isNil?` to determine whether a given term respectively corresponds to an instance of `cons` or `nil`. You are **strongly encouraged** to use `list` to create lists and `nth` to get the n^{th} element of a list (starting with `0`). If you wish to use a function that was not defined in the appendix, provide its definition as well.

Hint: Remember Lisp (non-)syntax of prefix notation, for example, comparisons are written as (= a b).

Note: Indent your code properly when answering this question, syntax is very important here.

```
(def (derive x expr)
  (









  )
  rest)
```

Exercise 3: Streams (10 points)

We all know the common use case of replacing all occurrences of a substring `pattern` within a string `text` by another substring `replacement`. If we represent strings as `List[Char]`, we can write the signature of such a method as

```
def replaceAll(text: List[Char], pattern: List[Char],
    replacement: List[Char]): List[Char] = /* omitted */
```

(do *not* implement this method)

Even though this method seems very well defined on its own, there are several interpretations that we can give to partially overlapping patterns and replacements. For example, what should

```
replaceAll(List('a', 'a', 'a'), List('a', 'a'), List('b'))
```

return? It could return `List('b', 'a')` or `List('a', 'b')`.

In this exercise, we use the interpretation that patterns are tested *left-to-right*, so that the left-most match wins. Under that interpretation, `List('b', 'a')` must be returned.

Moreover, we choose a slightly unusual interpretation that, after replacing a substring, we consider the replacement itself, together with the rest of the input string, as candidate for further replacement. This means that

```
replaceAll(List('a', 'a', 'b', 'a', 'b', 'a'), List('a', 'b', 'a'), List('b', 'a'))
```

results in `List('a', 'b', 'b', 'a')`.

Indeed, we first find the substring `aba` at index 1. We immediately send the left part of the input (until index 1, i.e., `a`) to the output, so it will not be reconsidered for replacement. We then replace `aba` by `ba`, resulting in a new remaining input `baba`. In that substring, we identify a new occurrence of `aba` (after the first `b`, which is emitted to the output) which must be replaced. Note that this subsequence was not part of the original input. This results in the new remaining input `ba`. At this point, no new replacement is possible. The final result is therefore `a` followed by `b` followed by `ba`, i.e., `abba`.

In this exercise, the `replacement` is always guaranteed to be strictly shorter than the `pattern` (i.e., `replacement.size < pattern.size`).

We can generalize this problem to streams, by admitting `input: Stream[Char]` and returning an output `Stream[Char]`:

```
def replaceAll(input: Stream[Char], pattern: List[Char],
    replacement: List[Char]): Stream[Char] = ???
```

Note that the `pattern` and `replacement` remain `List`s. The resulting stream will be infinite if and only if `input` is infinite.

Exercise 3.1

Implement the helper function

```
def testStartsWith(input: Stream[Char], pattern: List[Char]): Option[Stream[Char]]
```

which tests whether `input` starts with `pattern`, and if yes, returns the remaining of the `input`.

Examples:

```
testStartsWith(Stream('a', 'b', 'c'), List('a', 'b')) == Some(Stream('c'))
testStartsWith(Stream('a', 'b', 'c'), List('b', 'c')) == None
testStartsWith('a' #:: 'b' #:: infiniteStream, List('a')) == Some('b' #:: infiniteStream)
```

When `replacement.size >= pattern.size`, your implementation can behave in an arbitrary way (e.g., it could throw or infinitely loop). For all other inputs, `testStartsWith` must terminate in finite time, whether `input` is finite or infinite.

To get full points in this subquestion, your implementation of `testStartsWith` should complete in `O(n)` time, where `n` is `pattern.size` (assuming the input stream can produce each new element that you request in constant time).

```
def testStartsWith(input: Stream[Char], pattern: List[Char]): Option[Stream[Char]] =
```

Exercise 3.2

Implement the function `replaceAll` for streams defined above.

Your implementation must be able to handle both finite and infinite input streams. This means that `replaceAll(input, pat, repl).take(n).toList` must complete in finite time for all `input`, `pat`, `repl` and `n`, such that `replacement.size < pattern.size`, even if `input` is infinite.

There are no constraints on the complexity of `replaceAll`: even an inefficient implementation can get full points (as long as it satisfies the previous paragraph).

```
def replaceAll(input: Stream[Char], pattern: List[Char],
    replacement: List[Char]): Stream[Char] =
```

Exercise 3.3

We now further generalize the problem to a list of pairs `(pattern, replacement)`:

```
def replaceAllMany(input: Stream[Char],
    patternsAndReplacements: List[(List[Char], List[Char])]): Stream[Char] = ???
```

which, at each step, tries all the patterns in `patternsAndReplacements` and applies the first one that matches, then starts over with the resulting new input.

For all pair `(pat, repl)` in `patternsAndReplacements`, it is guaranteed that `repl.size < pat.size`.

Examples:

```
replaceAll(Stream('a', 'a', 'b', 'b'), List(
    (List('a', 'b'), List('c')),
    (List('a', 'a'), List('d'))
)) == Stream('d', 'b', 'b')
```

Note that the second pattern won because it applies *earlier in the input string.*

```
replaceAll(Stream('d', 'a', 'a', 'a', 'b', 'b'), List(
    (List('e', 'b'), List('c')),
    (List('a', 'a', 'a'), List('d', 'e')),
    (List('d', 'd'), List('z'))
)) == Stream('d', 'd', 'c', 'b')
```

Here, note that after applying the second pattern at index 1, we obtained the new input `debb`, which subsequently matched the first pattern. The `dd` in the final string was not replaced by the third pattern, because the first `d` was sent to the output right from the start, given that the first match was found at index 1.

Implement `replaceAllMany`. Similarly to `replaceAll`, your implementation must be able to handle both finite and infinite input streams. There are otherwise no constraints on its complexity.

Please answer on the following page.

```
def replaceAllMany(input: Stream[Char],
    patternsAndReplacements: List[(List[Char], List[Char])]): Stream[Char] =
```

Appendix

API

on Constraints

- `def propVar(): Formula`: Returns a new propositional variable with a fresh (unique) name.
- `case class And(lhs: Formula, rhs: Formula) extends Formula`: Conjunction of `lhs` and `rhs`.
- `def and(formulas: Formula*): Formula`: Returns the conjunction of all formulas.
- `def and(formulas: List[Formula]): Formula`: Returns the conjunction of all formulas in the list.
- `f1 && f2`: Is equivalent to `And(f1, f2)`.
- `case class Or(lhs: Formula, rhs: Formula) extends Formula`: Disjunction of `lhs` and `rhs`.
- `def or(formulas: Formula*): Formula`: Returns the disjunction of all formulas.
- `def or(formulas: List[Formula]): Formula`: Returns the disjunction of all formulas in the list.
- `f1 || f2`: Is equivalent to `Or(f1, f2)`.
- `case class Implies(lhs: Formula, rhs: Formula) extends Formula`: Implication by the `lhs` of the `rhs`.
- `case class Not(formula: Formula) extends Formula`: Negation of `formula`.
- `!f`: Is equivalent to `Not(f)`.
- `case object True extends Formula`: The trivially true formula.
- `case object False extends Fomula`: The trivially false formula.

on List (containing elements of type A):

- `xs ++ (ys: List[A]): List[A]`: appends the list `ys` to the right of `xs`, returning a `List[A]`.
- `xs.apply(n: Int): A`, or `xs(n: Int): A`: returns the `n`-th element of `xs`. Throws an exception if there is no element at that index.
- `xs.drop(n: Int): List[A]`: returns a `List[A]` that contains all elements of `xs` except the first `n` ones. If there are less than `n` elements in `xs`, returns the empty list.
- `xs.filter(p: A => Boolean): List[A]`: returns all elements from `xs` that satisfy the predicate `p` as a `List[A]`.
- `xs.flatMap[B](f: A => List[B]): List[B]`: applies `f` to every element of the list `xs`, and flattens the result into a `List[B]`.
- `xs.foldLeft[B](z: B)(op: (B, A) => B): B`: applies the binary operator `op` to a start value and all elements of the list, going left to right.
- `xs.map[B](f: A => B): List[B]`: applies `f` to every element of the list `xs` and returns a new list of type `List[B]`.
- `xs.nonEmpty: Boolean`: returns `true` if the list has at least one element, `false` otherwise.

- `xs.reverse: List[A]`: reverses the elements of the list `xs`.
- `xs.take(n: Int): List[A]`: returns a `List[A]` containing the first `n` elements of `xs`. If there are less than `n` elements in `xs`, returns these elements.
- `xs.toMap: Map[K, V]`: provided `A` is a pair `(K, V)`, converts this list to a `Map[K, V]`.
- `xs.zip(ys: List[B]): List[(A, B)]`: zips elements of `xs` and `ys` in a pairwise fashion. If one list is longer than the other one, remaining elements are discarded. Returns a `List[(A, B)]`.

You can use the same API for `Stream`, replacing `List` by `Stream`.

on Stream (containing elements of type A):

- `xs #:: (ys: => Stream[A]): Stream[A]`: Builds a new stream starting with the element `xs`, and whose future elements will be those of `ys`.

on Stream (the object):

- `Stream.Empty: Stream[Nothing]`: The empty stream.
- `Stream.from(i: Int): Stream[Int]`: Creates an infinite stream of integers starting at i.

on Map (containing keys of type K, values of type V):

- `mp.get(k: K): Option[V]`: For a given key `k`, returns `Some(v)` if a value exists in the map `mp`, `None` otherwise.

Lisp API

Here are a few Lisp functions you may find useful:

- `(isCons? term)`: returns `1` if the provided term is equal to (cons x y) for some `x` and `y`, and `0` otherwise.
- `(isNil? term)`: returns `1` if the provided term is equal to `nil`, and `0` otherwise.
- `(cons x y)`: constructs a list with head `x` and tail `y` (corresponds to `::` in Scala).
- `nil`: constructs an empty list (corresponds to `Nil` in Scala).
- `(list x1 x2 ... xn)`: creates the list `(cons x1 (cons x2 (...((cons xn nil)...)`.
- `(car x)`: head of the list `x`.
- `(cdr x)`: tail of the list `x`.
- `(nth i x)`: element in position `i` of the list `x` starting from 0.
- `(if cond then else)`: returns the term `then` if `cond` DOES NOT evaluate to `0`, and `else` otherwise.

You may also find the following conditional construct useful:

```
(cond (test1 expr1) ...
      (testN exprN)
      (else exprElse))
```

The `(test expr)` pairs are successfully considered until a `test` evaluates to something different from `0`, in which case the corresponding `expr` is returned, or until the `(else exprElse)` pair is reached in which case `exprElse` is returned.

Don't forget the syntax for

1. defining values:

   ```
   (val name body rest)
   ```

2. defining functions:

   ```
   (def (name arg1 ... argN) body rest)
   ```