Functional Programming

Final Exam Solution

Friday, December 22 2017

Exercise 1: Ex 1 ... (10 points)

```
val exactlyOneBuildingPerSlot: Formula = and({
  for {
    s <- allSlots
  } yield {

    val atLeastOne = or({
      for {
        b <- allBuildingSorts
      } yield vars((s, b))
    }: _*)

    val atMostOne = and({
      for {
        i <- 0 until allBuildingSorts.size
        j <- i + 1 to allBuildingSorts.size
      } yield Not(And(
        vars((s, allBuildingSorts(i))),
        vars((s, allBuildingSorts(j)))))
    }: _*)

    and(atLeastOne, atMostOne)
  }
}: _*)


val atLeastOneLifeGuardTower: Formula = or({
  for {
    s <- allSlots
  } yield vars((s, LifeGuardTower))
}: _*)


val correctChangingRoomLocation: Formula =
  or(vars((allSlots.head, ChangingRoom)), vars((allSlots.last, ChangingRoom)))


val noTwoAdjacentShops: Formula = and({

  for ((i, j) <- allSlots.zip(allSlots.tail)) yield {
    not(and(vars(i, Shop), vars(j, Shop)))
  }

}: _*)


val restaurantsAjacentToilets: Formula = and({
```

```
  def varOrFalse(i: Int, b: BuildingSort): Formula =
    if (i >= 0 && i < n) vars((i, b)) else False

  for (s <- allSlots) yield {
    implies(
      vars((s, Restaurant)),
      or(
        varOrFalse(s - 1, Toilets),
        varOrFalse(s + 1, Toilets)))
  }
}: _*)
```

Exercise 2: Ex 2 Lisp (10 points)

Exercise 2.1: Scala implementation (5 points)

```
def derive(x: Symbol, expr: Any): Any = {
   expr match {
     case List('+, e1, e2) => List('+, derive(x, e1), derive(x, e2))
     case List('*, e1, e2) => List('+, List('*, derive(x, e1), e2), List('*, e1, derive(x,
     case _ => if (x == expr) 1 else 0
   }
 }
```

Exercise 2.2: Translation into Lisp (5 points)

```
(def (derive x expr)
  (if (isCons? expr)
    (if (= '+ (nth 0 expr))
      (list '+ (derive x (nth 1 expr)) (derive x (nth 2 expr)))
      (list '+ (list '* (derive x (nth 1 expr)) (nth 2 expr))
               (list '* (nth 1 expr) (derive x (nth 2 expr)))))
    (if (= x expr) 1 0))
  rest)
```

Exercise 3: Ex 3 Streams (10 points)

```
def testStartsWith(input: Stream[Char], pattern: List[Char]): Option[Stream[Char]] = {
  pattern match {
    case Nil =>
      Some(input)
    case x :: xs =>
      input match {
        case y #:: ys if y == x =>
          testStartsWith(ys, xs)
        case _ =>
          None
      }
  }
}

def replaceAll(input: Stream[Char], pattern: List[Char],
    replacement: List): Stream[Char] = {
  testStartsWith(input, pattern) match {
    case None =>
      input match {
        case x #:: xs =>
          x #:: replaceAll(xs, pattern, replacement)
        case Stream.Empty =>
          Stream.Empty
      }

    case Some(rest) =>
      val newInput = replacement.toStream ++ rest
      replaceAll(newInput, pattern, replacement)
  }
}

def replaceAllMany(input: Stream[Char],
    patternsAndReplacements: List[(List[Char], List[Char])]): Stream[Char] = {
  val resultsOfTests = patternsAndReplacements.map {
    case (pat, repl) => testStartsWith(input, pat) -> repl
  }
  resultsOfTests.collectFirst {
    case (Some(rest), repl) =>
      val newInput = repl.toStream ++ rest
      replaceAllMany(newInput, patternsAndReplacements)
  }.getOrElse {
    input match {
      case x #:: xs =>
        x #:: replaceAll(xs, patternsAndReplacements)
      case Stream.Empty =>
        Stream.Empty
    }
  }
}
```