CHALMERS TEKNISKA HÖGSKOLA
Datavetenskap
Jan-Willem Roorda

Wednesday, August 18th, 2004
Functional Programming
INN040/TDA450

Exam Functional Programming

Wednesday, August 18th, 2004, 14.15-18.15.
Examiner: John Hughes, tel 070 756 3760.
Questions during the exam will be answered by Jan-Willem Roorda, tel 031-7721023.

Permitted aids:
English-Swedish or English-other language dictionary.

- Begin each question on a new sheet. Write your personal number on every sheet.
- You may lose marks for unnecessarily long, complicated, or unstructured solutions.
- Full marks are awarded for solutions which are elegant, efficient, and correct.
- You are free to use any Haskell standard functions, including those whose definitions are attached, unless the question specifically forbids you to do so.
- You may use the solution of an earlier part of a question to help solve a later part, even if you did not succeed in solving the earlier part.
- The exam consists of 3 questions, worth 12, 25, and 12 points. A total of 20 points is sufficient to pass for Chalmers students. GU students need a total of 23 points to pass.

1. (a) A *rotation* of a list is obtained by removing any number of elements from the beginning of the list, and appending them to the end of the list instead. For example, the rotations of `[1,2,3]` are `[1,2,3]`, `[2,3,1]` and `[3,1,2]`. Define a function

```
rotations :: [a] -> [[a]]
```

which returns a list of all the possible rotations of its argument. For example,

```
rotations [1,2,3] == [[1,2,3],[2,3,1],[3,1,2]]
```

(4 p)

(b) A *rotation of a book title* is a rotation of the words in the title which does not begin with a "trivial" word. Given the definition

```
trivialWords = ["the", "a", "of", "my"]
```

define a function

```
titleRotations :: String -> [String]
```

which returns a list of all the rotations of a book title. For example,

```
titleRotations "the craft of functional programming"
  == ["craft of functional programming the",
      "functional programming the craft of",
      "programming the craft of functional"]
```

(4 p)

(c) A *keyword in context (KWIC) index* is a sorted list of rotations of titles, which makes it easy to find a title containing a particular word. For example, a KWIC index of the titles `"the craft of functional programming"` and `"beware the jabberwock my son"` would contain

```
beware the jabberwock my son
craft of functional programming the
functional programming the craft of
jabberwock my son beware the
programming the craft of functional
son beware the jabberwock my
```

Define a function

```
kwic :: [String] -> [String]
```

which produces a KWIC index from a list of titles. (4 p)

2. In this question you will develop functions for manipulating *regions in the plane*, which might form part of a library for 2D graphics. We shall represent points by their $x - y$ coordinates

```
data Point = Pt Float Float
```

and regions by *functions* with the type

```
type Region = Point -> Bool
```

A region `r` represents a set of points in the plane; a point `p` is in this set if `r p` returns `True`.

(a) Define

```
inR :: Point -> Region -> Bool
```

such that `p `inR` r` is `True` if `p` is in the region `r`. (2 p)

(b) Define functions

```
box :: Point -> Point -> Region
circle :: Point -> Float -> Region
```

where

- `box p q` returns the region of points in the rectangle whose lower left hand corner is at `p`, and whose upper right hand corner is at `q`,
- `circle p r` returns the region of points in the circle of radius `r` with centre at `p`.

Notice that these functions return *functions* as results! (4 p)

(c) Define

```
unionR :: Region -> Region -> Region
intersectR :: Region -> Region -> Region
complementR :: Region -> Region
```

such that a point `p` is in `r `unionR` r'` if it is in `r` *or* `r'`, it is in `r `intersectR` r'` if it is in `r` *and* `r'`, and it is in `complementR r` if it is *not* in `r`. (6 p)

(d) Define

```
ring :: Point -> Float -> Float -> Region
```

such that `ring p r w` constructs a ring-shaped region centred on `p`, where the inside edge of the ring is a circle of radius `r`, and the width (or thickness) of the ring is `w`. (3 p)

(e) Region operations can be made more efficient by storing a *bounding box* with each region: the points outside the box are either all in, or all not in the region, which makes testing whether such a point is in the region fast. We represent boxes by their lower left and upper right corners, and define

```
data Box = Box Point Point
type BoxedRegion = (Box,Bool,Region)
```

A *boxed region* `(box,b,r)` contains a point `p` if `p` lies inside the box and is in region `r`, or lies outside the box and `b` is `True`.

i. Define

```
inBR :: Point -> BoxedRegion -> Bool
```

to test whether a point lies in a boxed region. (2 p)

ii. Define

```
boxBR :: Point -> Point -> BoxedRegion
circleBR :: Point -> Float -> BoxedRegion
```

to construct boxed regions representing the same sets of points as the functions `box` and `circle` from part 2b. (4 p)

iii. Define

```
unionBR :: BoxedRegion -> BoxedRegion -> BoxedRegion
```

to compute the union of two boxed regions. (4 p)

3. In this question, you will write functions to manipulate arithmetic expressions in *reverse polish* notation. In this notation, arithmetic operators are written *after* their operands, so $1 + 2$ for example is written as 1 2 +. An operand can itself be a reverse polish expression: for example $(1 + 2) \times 3$ is written as 1 2 + 3 $\times$, in which the first operand of $\times$ is the expression 1 2 +. Brackets are not needed because there is no ambiguity to resolve: different bracketings of the "same" expression lead to different reverse polish translations. For example, $1 + (2 \times 3)$ is written as 1 2 3 $\times$ + — compare with the previous case. Because of its unambiguity, reverse polish notation is often used in the input to pocket calculators.

   We will work with expressions containing just integers, addition, and multiplication. Such expressions can be represented in Haskell programs using the type

```
data Expr = Num Int | Add Expr Expr | Mul Expr Expr
```

   (a) How would the expression $(1 + 2) \times 3$ be represented as a value of this type? (2 p)

   (b) Define a function

```
rp :: Expr -> [String]
```

   which translates an expression to reverse polish notation. Represent reverse polish by a list of strings, each string being either a number, `"+"`, or `"*"`. For example,

```
Main> rp (Add (Mul (Num 2) (Num 10)) (Num 3))
["2","10","*","3","+"]
```

   (4 p)

   (c) Reverse polish expressions can be evaluated using a *stack* of numbers. The expression is evaluated from left to right, as follows:

   - Each number is added to the front of the stack.
   - + is evaluated by replacing the first two numbers on the stack by their sum.
   - $\times$ is evaluated by replacing the first two numbers on the stack by their product.

   For example, 1 2 3 $\times$ + would be evaluated as follows:

Symbol	Action	Stack
		Initially empty
1	Add 1 to the front of the stack	1
2	Add 2 to the front of the stack	2 1
3	Add 3 to the front of the stack	3 2 1
$\times$	Replace 3 2 by 6	6 1
+	Replace 6 1 by 7	7

   Define a function

```
evaluate :: [String] -> [Int] -> [Int]
```

which takes a reverse polish expression and an initial stack as arguments, and returns the stack resulting from the evaluation of the expression as its result. For example,

```
Main> evaluate ["1","2","3","*","+"] []
[7]
Main> evaluate ["3","*"] [2,1]
[6,1]
```

(6 p)