EBERHARD KARLS UNIVERSITÄT TÜBINGEN

Functional Programming
Wintersemester 2021/2022
Prof. Dr. T. Grust, D. Hirn

Assignment 4 (19.11.2021)

Handin until: Friday, 26.11.2021, 00:00

Exercise 1: Minesweeper **[10 Points]**

The objective of Minesweeper is to clear a rectangular field which contains hidden *mines*. For each field cell the player steps on, a hint is given about the number of mines in the direct neighborhood.
We want to implement a function to compute these numbers for a given field with visible mines:

```
    minesweep :: [[Char]] -> [[Int]]
```

Example:

```
field = [ "        "
        , "***     "
        , "* *     "
        , "***     "
        ]

Main> minesweep field
[ [ 2, 3, 2, 1, 0, 0, 0, 0 ]
, [ 3, 5, 3, 2, 0, 0, 0, 0 ]
, [ 5, 8, 5, 3, 0, 0, 0, 0 ]
, [ 3, 5, 3, 2, 0, 0, 0, 0 ]
]
```

Follow these steps to solve the problem:

1. The actual algorithm shall be formulated as a combination of helper functions, you have to implement in advance:
   (a) Write a function `num :: Char -> Int` which takes a field cell and returns 1, if it is mined (`'*'`), and `0`, if it is safe (`' '`).
   (b) Write a function `shiftL :: a -> [a] -> [a]`. `shiftL x xs` shifts all elements of a list `xs` to the left, the original head is dropped, the new rightmost element is `x`.
   Note: The length of list `xs` before and after shifting is identical.
   (c) Write a function `shiftR :: a -> [a] -> [a]` which shifts a given list of numbers to the right.
   (d) Write a function

   ```
   zipWith3' :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
   ```

   It takes a function `f :: (a -> b -> c -> d)`, as well as three lists of identical length and returns a list where each element is the combination (using `f`) of the input lists' elements at the same position.[1]
   Example:

   ```
   zipWith3' (\a b c -> a * b + c) [1,2] [3,4] [5,6] ≡ [8,14]
   ```

   (e) Use functions `shiftL`, `shiftR` and `zipWith3'` to implement a function

   ```
   addNeighbours :: [Int] -> [Int]
   ```

   For each element of a given list of integers, the function returns the sum of the element itself, together with its left and right neighbour. The first (last) element has no left (right) neighbour.
   Example:

   ```
   addNeighbours [1,1,1,0,1] ≡ [2,3,2,2,1]
   ```

[1] Obviously you shall implement this function on your own and must not use the `Prelude` function `zipWith3`.

(f) Write a function which transposes the rows and columns of a given matrix (list of lists of identical length):[2]

```haskell
transpose :: [[a]] -> [[a]]
```

Example:

```haskell
transpose [[1,2,3],[4,5,6]] ≡ [[1,4],[2,5],[3,6]]
```

2. Implement `minesweep` as a combination of `Prelude` function `map` and the helper functions `num`, `addNeighbours` and `transpose` defined in step 1.

Exercise 2: Regular Expressions **[10 Points]**

Finite state machines aren't the only method to implement regular expression matching. Here, we will build a regular expression matcher using the *derivatives of a regular expression*.
To implement this approach, we first need a representation for regular expressions on a given alphabet of symbols.

1. Define a data type `RegExp a` for regular expressions, which are one of:
   - the empty sequence of symbols ε,
   - a symbol of the alphabet (symbols have type a – typically $\mathbf{a} \equiv \mathbf{Char}$),
   - a concatenation $r_1 r_2$ of two regular expressions r_1 and r_2 (r_1 followed by r_2),
   - the *Kleene star* r^* of a regular expression r (r repeated zero or more times),
   - an alternation $r_1|r_2$ of two regular expressions r_1 and r_2 (r_1 or r_2),
   - a special regular expression $\varnothing$ which accepts no input at all.
2. Write an instance of type class `Show` for `RegExp a`. Use parentheses () to group parts of the regular expression, if necessary.

   Example: The output of `show` for regular expression $(x|(yz))^*$ looks like this: `"(x|(yz))*"`.

The *derivative of a regular expression* r with respect to symbol a is another regular expression r'. If input s is accepted by r, then r' accepts s with the starting symbol a removed. For example, consider the regular expression $r = \mathtt{ab}^*$. The derivative of r with respect to $\mathtt{a}$ is $\mathtt{b}^*$ and the derivative of $\mathtt{b}^*$ with respect to $\mathtt{b}$ is again $\mathtt{b}^*$. However, the derivative of r with respect to $\mathtt{b}$ is the regular expression $\varnothing$.
These derivatives can be used to implement regular expression matching.
First we need a function $\nu(r)$ to test whether a regular expression r is nullable. We say that r is nullable, if r accepts the empty sequence ε:

$$
\begin{aligned}
\nu(\varepsilon) &= \mathtt{True} \\
\nu(a) &= \mathtt{False} \\
\nu(r^*) &= \mathtt{True} \\
\nu(r_1 r_2) &= \nu(r_1) \wedge \nu(r_2) \\
\nu(r_1|r_2) &= \nu(r_1) \vee \nu(r_2) \\
\nu(\varnothing) &= \mathtt{False}
\end{aligned}
$$

3. Write a function `nullable :: RegExp a -> Bool` implementing ν.

Now we can define a function $\partial_a(r)$ to compute the derivative of a regular expression r with respect to a symbol a:

$$
\begin{aligned}
\partial_a(\varepsilon) &= \varnothing \\
\partial_a(b) &= \begin{cases} \varepsilon & \text{, } \mathit{if}\ a = b \\ \varnothing & \text{, } \mathit{if}\ a \neq b \end{cases} \\
\partial_a(r^*) &= \partial_a(r r^*) \\
\partial_a(r_1 r_2) &= \begin{cases} \partial_a(r_1) r_2 \,|\, \partial_a(r_2) & \text{, } \mathit{if}\ \nu(r_1) \\ \partial_a(r_1) r_2 & \text{, } \mathit{if}\ \neg\nu(r_1) \end{cases} \\
\partial_a(r_1|r_2) &= \partial_a(r_1) \,|\, \partial_a(r_2) \\
\partial_a(\varnothing) &= \varnothing
\end{aligned}
$$

[2] You must not use the `Data.List` function `transpose`, but implement the function on your own.

4. Write a function `derive :: Eq a => RegExp a -> a -> RegExp a` implementing ∂.

Supposed we have a regular expression r and a string of symbols $s = a_1 \ldots a_n$. To test whether r accepts s, we can make use of a successive application of ∂ to r with respect to the symbols of s. If and only if the final derivative with respect to a_n is nullable, i.e., matches the empty string ε, the regular expression r matches the whole string s:

$$r \text{ matches } s \quad \Leftrightarrow \quad \nu(\partial_{a_n}(\cdots \partial_{a_1}(r)))$$

5. Write a function `match :: Eq a => RegExp a -> [a] -> Bool` implementing the regular expression matcher. Remember to provide some tests, i.e. sample regular expressions and sample symbol sequences.

[Optional:] Extend your definitions of `RegExp a`, `nullable` and `derive` to also support the *Kleene plus*: A regular expression can also be r^+ (the regular expression r repeated one or more times).