EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Functional Programming
Summer term 2023
Prof. Torsten Grust, Denis Hirn
WSI — Database Systems Research Group

Assignment 4

Hand in this assignment until **Friday, 26. May 2023, 10:00** at the latest.

Ⓔxam-style Exercises

Exercises marked with Ⓔ are similar in style to those you will find in the exam. You can use these to hone your expectations and gauge your skills.

Running out of ideas?

Are you hitting a roadblock? Are some of the exercises unclear? Do you just need that one hint to get the ball rolling? Refer to the `#forum` channel on our Discord server and check the tag for this assignment—maybe you'll find just the help you need.

Task 1: Minesweeper (List Processing)

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:

(a) The actual algorithm shall be formulated as a combination of helper functions, you have to implement in advance:

i. Write a function `num :: Char -> Int` which takes a field cell and returns `1`, if it is mined (`'*'`), and `0`, if it is safe (`' '`).

ii. 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.

iii. Write a function `shiftR :: a -> [a] -> [a]` which shifts a given list of numbers to the right.

iv. 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]
```

v. 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]
```

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

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

Example:

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

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

Task 2: Regular Expressions (Algebraic Data Types)

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.

(a) Complete the definition of an algebraic data type (ADT) for regular expressions:

```
data RegExp a = RxNone     -- ε
              | RxEpsilon  -- e*
              | ...
```

The ADT needs constructors for the following cases:

- the empty sequence of symbols ε,
- a symbol of the alphabet (symbols have type a – typically `a ≡ 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.

(b) 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))*"`.

[1] Obviously you shall implement this function on your own and must not use the `Prelude` function `zipWith3`.
[2] You must not use the `Data.List` function `transpose`, but implement the function on your own.

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 $r = \texttt{ab}^*$. The derivative of r with respect to `a` is $\texttt{b}^*$ and the derivative of $\texttt{b}^*$ with respect to `b` is again $\texttt{b}^*$. However, the derivative of r with respect to `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:

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

(c) Write a function `nullable :: RegExp a -> Bool` implementing ν.

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

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

(d) Write a function `derive :: Eq a => RegExp a -> a -> RegExp a` implementing ∂.

Suppose we have a regular expression r and a string of symbols $s = c_1 \ldots c_n$. 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 c_n is nullable, i.e., matches the empty string ε, the regular expression r matches the whole string s:

$$r \text{ matches } s \quad \Leftrightarrow \quad \nu(\partial_{c_n}(\cdots \partial_{c_1}(r)))$$

(e) 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).