Principles of Programming Languages 2016W, Functional Programming

Assignment 1: Recursive Functions in Haskell

Assignment 1.1: Tokenizer (1 Point)

Write a function `tokenize`

```
tokenize :: String -> [String]
```

to split an input string into a list of strings (tokens) as follows:

Input is a string which contains opening and closing rounded brackets and words (or single symbols) separated by white spaces or brackets. In fact, input strings are intended to represent expressions in prefix notation, e.g.,

```
"(+ (+ x 12) (- y))"
```

The result list of tokens then contains all the opening and closing brackets and the words separated by white spaces. For example, the above input string should give the following list of tokens

```
["(","+","(","+","x","12",")","(","-","y",")",")"]
```

You can use the following functions (see program frame for Assignment 1):

Function `skipWhiteSpaces` skips leading white spaces from an input string.

```
skipWhiteSpaces :: String -> String
skipWhiteSpaces (c:cs) | isSpace c   = skipWhiteSpaces cs
skipWhiteSpaces cs                   = cs
```

Function `nextWordAndRest` gets the next word and the rest of the input string. Note, that an empty token `""` means that no further tokens are available.

```
nextWordAndRest :: String -> (String, String)
nextWordAndRest text =
    nextWordAndRest' (skipWhiteSpaces text) ""
    where
        nextWordAndRest' :: String -> String -> (String, String)
        nextWordAndRest' (s:rest)     word | isSpace s   = (reverse word, rest)
        nextWordAndRest' rest@('(':_) word               = (reverse word, rest)
        nextWordAndRest' rest@(')':_) word               = (reverse word, rest)
        nextWordAndRest' []           word               = (reverse word, [])
        nextWordAndRest' (c:rest)     word               = nextWordAndRest' rest (c:word)
```

With those functions `tokenize` is simple and works as follows:

- First skip white spaces from the input string.
- Then test if there is an opening or closing brackets. They give you a next token `"("` or `")"`.
- For any other case, use function `nextWordAndRest` to get the next word and the rest of the input.
- Recursively get the rest of the tokens from the rest of the input string.

Hints:

- A good design idea is to use a local recursive function which actually does the work.
- Use pattern matching and case expressions to distinguish the different cases.

Assignment 1.2: **(2 Points)**

Write a function `isValidExprTokens`

```
isValidExprTokens :: [String] -> (Bool, [String])
```

which tests if the tokens represent an arithmetic expression `Expr` with the following syntax:

```
Expr = BinExpr | UnExpr | Num | Var.
BinExpr = "(" "+" Expr Expr ")".
UnExpr = " (" "-" Expr ")".
Num = Dig {Dig}.
Var = Alpha {Alpha | Dig}.
```

`Dig` is a digit.

`Alpha` is a letter.

If a character is a digit can be tested by function `isNumber` from module `Data.Char` (import this module by `import Data.Char`). If a character is a letter can be tested by function `isAlpha`. Function `isAlphaNum` tests if a character is a letter or digit.

The function should return a pair of a Boolean for success or failure and the rest list of tokens (compare function `nextWordAndRest` from above).

Hints:

- The function `isValidExprTokens` should have equivalent structure as the syntax definition. As rules for `BinExpr` and `UnExpr` are recursive, your function should be recursive.
- Intensively use pattern matching and case expressions.
- In case of a failure, the returned rest of tokens should start at the failure position.

Submit your solution by electronically

- by Dec 1, 15:30
- through the following Web page using your student id and password received by email for login
  http://www.ssw.uni-linz.ac.at/scripts/upload/upload_form.php?lecture=POPL&lang=EN