Assignment 2

CS 4400 Programming Languages

Due: Thursday, September 24, 2020 at 9pm

Submission:

1. Submit a single file, `Assignment02.hs` via https://handins.ccs.neu.edu/courses/119.
2. At the very top, the file should contain a preamble following this template.

```
{- |
Module      :  Assignment02
Description :  Assignment 2 submission for CS 4400.
Copyright   :  (c) <your name>

Maintainer  :  <your email>
-}

module Assignment02 where

... your code goes here ...
```

   The rest of the file will contain your solutions to the exercises below.

3. Every top-level definition[1] must include a purpose statement (for functions) and a type signature, followed by one or more defining equations.
4. Double-check that you have named everything correctly.
5. Make sure your file loads into GHCi or can be compiled by GHC without any errors.
6. If something is not clear, or you are struggling with some questions, talk to us: in office hours, after class, on Piazza, via email.
7. This assignment is to be completed and submitted individually.

Purpose: The aim of this assignment is to practice working with BNFs and abstract syntax trees, as well as writing higher-order functions with polymorphic types.

[1] A *top-level definition* is one that is not nested inside another one.

Examples and Tests

Starting with this assignment, you are expected to write examples and tests for every data type and function you write. For check-expect-style unit tests, you can use the `SimpleTests` module available through the course website. Its use will be demonstrated in class. Alternatively, you can use one of Haskell's popular unit testing frameworks, `HUnit` or `HSpec`. We might demonstrate them later in the semester. For example variables, use the prefix `ex_`. For tests, the prefix `test_`.

Questions

BNF and Abstract Syntax

An s-expression is:

a) an atom
b) a list of s-expressions, enclosed in parentheses, separated by spaces

For now, an atom is either a number (integer), or a symbol. Textual examples of s-expressions are:

```
(1 20 x 30 foo)
(+ 13 23)
20
(a b (c d))
x
(32 * (30 z) =)
(/ (- 10 2) 4)
```

1. In a `{- -}` comment, write down the BNF for s-expressions.
2. Design a datatype for s-expressions. Name it `SExpr`. Use `String` to represent symbols. Write at least 3 meaningful examples of `SExpr` values.
3. Write a function `showSExpr` which takes an s-expression and prints its string representation as above. Use single spaces between elements of an s-expression list.
4. Recall the SAE language from class:

   ```haskell
   data SAE = Number Integer            -- <SAE> ::= <Integer>
            | Add SAE SAE               --         | <SAE> + <SAE>
            | Sub SAE SAE               --         | <SAE> - <SAE>
            | Mul SAE SAE               --         | <SAE> * <SAE>
            | Div SAE SAE               --         | <SAE> / <SAE>
            deriving (Eq, Show)
   ```

   SAE expressions can be represented as s-expressions, using prefix notation (like Racket). Here are a few examples:

```
32
(+ 12 14)
(- (/ 16 4) (- 5 4))
```

Write two functions:

(a) a function `fromSExpr` which converts an s-expression (with symbols restricted to `+`, `-`, `*`, and `/`) into an `SAE` expression
(b) a function `toSExpr` which converts an `SAE` expression into its s-expression representation.

For 4a, you can assume that only valid SAE s-expression in prefix form will be provided.

Polymorphic higher-order functions

5. A polymorphic binary tree is defined as follows:

```haskell
data BinaryTree a = Empty
                  | Node a (BinaryTree a) (BinaryTree a)
```

Design a function `treeMap` which takes a function and maps it over the given binary tree. In other words, it applies the given function to each element in the tree, preserving its structure.

Examples:

```haskell
treeMap show (Node 1 (Node 42 Empty (Node 4 Empty Empty)) (Node 3 Empty Empty))
  =  Node "1" (Node "42" Empty (Node "4" Empty Empty)) (Node "3" Empty Empty)
```

```haskell
treeMap head (Node [1] (Node [10, 2, -4] Empty Empty) (Node [-4, 12] Empty Empty))
  = Node 1 (Node 10 Empty Empty) (Node (-4) Empty Empty)
```

6. Design a higher-order function `iterateN`, which takes a function `f`, an `Integer` `n` and an initial value `init` (of an arbitrary type) and applies `f` `n`-times, starting with `init`. Write the correct polymorphic signature.

For example, if the following is defined:

```haskell
double :: Integer -> Integer
double x = x + x
```

the following should hold:

```haskell
iterateN double 5 1 = 32

iterateN double 0 42 = 42

iterateN tail 3 ["Alewife", "Davis", "Porter", "Harvard", "Central"] =
  ["Harvard", "Central"]
```