Assignment 3

CS 4400 Programming Languages

Start early and come to us with questions.

Due: 11pm on Thursday, October 1, 2020

Submission:

1. Submit a zip-file `Assignment03.zip` containing only the following files via https://handins.ccs.neu.edu/courses/119:

   - `Assignment03.hs`
   - `Syntax.hs`
   - `Eval.hs`

2. This assignment is meant to be worked on and submitted in pairs, but you can choose to work on your won. Even if you are looking for a partner, start working on the assignment now. You can combine your work with your partner's later. Once you have a partner or decide to work individually, request a team on Handins so we can approve it well ahead of the deadline.

3. At the very top, `Assignment03.hs` should contain a preamble following this template.

   ```
   {- |
   Module      :  Assignment03
   Description :  Assignment 3 submission for CS 4400.
   Copyright   :  (c) <your name>

   Maintainer  :  <your email>
   -}

   module Assignment03 where
   ```

   The rest of the file imports the necessary modules to run all tests and defines a `main` that runs all of the tests.

4. Every top-level definition must include a purpose statement (for functions) and a type signature, followed by one or more defining equations. Every function should have meaningful tests. You can use HSpec, HUnit, or the provided `SimpleTests` module. Data definitions should have a comment with the intended interpretation and meaningful examples.
5. Double-check that you have named everything as required and that functions required by this assignment have the correct type signatures.
6. Make sure your file loads into GHCi or can be compiled by GHC without any errors.

Purpose: The purpose of this assignment is to extend a simple language with a new numeric type, along with implcit coercions. It provides further practice with defining functions by recursion over an algebraic datatype and with manipulating abstract syntax. Furthermore, this assignment practices reading of code and working across multiple Haskell modules.

Questions

In class, we have extended the `SAE` language with bindings: a `let` construct as well as variable references. Let's call the language that resulted from this `protoScheme`. We would like to grow this language to become something closer to actual Scheme. Let's take a step in that direction and start building a baby version Scheme's Numerical Tower by adding floating-point arithmetic.

Assignment Pack

This assignment comes with an "assignment pack" containing the following files:

`Syntax.hs` Contains the definitions related to the syntax of our language. In particular, it contains the abstract syntax defintion. You will notice, that instead of `SAE`, the abstract syntax datatype is now called `Expr`. Defines: types `Variable` and `Expr`, functions `fromSExpression`, `toSExpression`, and `valueToSExpression`.

`Eval.hs` The interpreter proper. Contains the evaluation and substitution functions, as well as a function `runSExpression` for running s-expressions. Defines: `eval`, `subst`, and `runSExpression`.

`SExpression.hs` Abstract syntax of s-expressions. Notice that the s-expression datatype is also called `Expr`. That means that you always have to import it qualified and use a prefix. You don't need to modify this file for this assignment.

`SimpleTests.hs` "Ferd's simple testing module". Slightly improved from previous assignment. You can discard this in favor of `HSpec` or `HUnit` if you prefer those.

Completing fromSExpression and tests

1. Complete `fromSExpression` so that it covers the current incarnation of protoScheme:

   ```
   <Expr> ::= <Integer>
            | <Variable>
            | (+ <Expr> <Expr>)
            | (- <Expr> <Expr>)
            | (* <Expr> <Expr>)
            | (/ <Expr> <Expr>)
            | (let (<Variable> <Expr>) <Expr>)
   ```

   Note: In s-expressions, variables are represented as symbols, just like keywords in the language (currently only `let`). This means that you need to be careful about the order of pattern matching.

2. Write more tests for current implementations of `eval`, `subst`, and `fromSExpression`.

Extension: More than one numeric type

Syntax

3. Introduce floating point numbers (floats) into the language. This change should be introduced in the following steps:

   (a) `eval` should be able to return either an integer or a floating point *value*. Change the abstract syntax of `protoScheme` to accommodate this. In comments, describe and justify the changes you need to make. Do not hesitate to introduce new datatypes if needed. Use `Double` as the representation for floats.

   (b) Amend the type signatures of `eval` and `subst` to reflect these changes.

   (c) Modify the definition of `subst` to accommodate the changes in our abstract syntax.

   (d) The s-expression syntax already contains a case for real numbers. Add the corresponding clause to `fromSExpression`.

   (e) Modify and complete the two functions, along with tests:

   - `toSExpression` which converts a `protoScheme` `Expr` into an s-expression.
   - `valueToSExpression` which converts the return value of `eval` into an s-expression.

   Rationale: s-expressions will serve as our interface. We want to be able to evaluate an s-expression and get an s-expression back by invoking the function `runSExpression` defined in the `Expr` module.

Semantics

4. Update `eval` to implement the semantics of arithmetic operations on reals and integers. Each operand can evaluate to either an integer or a real number. The evaluation should proceed according to the following rules:

   (a) If both operands evaluate to integers, apply the correct operation.
   (b) If both operands evaluate to reals, apply the correct operation.
   (c) If one operand evaluates to an integer and the other to a float, convert the integer to a float and apply the floating point operation.

   You might want to check the documentation for the `Num`, `Integral`, and `Fractional` type classes. For conversions from an `Integer` use `fromInteger`.