Assignment 5

CS 4400 Programming Languages

Start early and come to us with questions.

Due: 11pm on Thursday, October 15, 2020

Submission:

1. Submit the following files via https://handins.ccs.neu.edu/courses/119:

   - `Assignment05.hs`
   - `Eval.hs`
   - `Syntax.hs`

2. This assignment is meant to be worked on and submitted in pairs, but you can choose to work on your own. Note, that you need to have a team on Handins to be able to submit (a singleton team or a pair).

3. At the very top, `Assignment05.hs` should contain a preamble following this template.

   ```
   {- |
   Module      :  Assignment05
   Description :  Assignment 5 submission for CS 4400.
   Copyright   :  (c) <your name>

   Maintainer  :  <your email>
   -}

   module Assignment05 where
   ```

   The rest of the file should contain in-comment answers to questions asked in this assignment and a `main` function running all unit 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. **Your grade might be reduced by up to 50% if your code does not compile and run.**

State of the Union

After the previous assignment, you should have a working implementation of `protoScheme` with the following features:

- let-bindings for introducing variables, and a variable reference expressions (provided to you at the beginning of the assignment and implemented using substitution)
- arithmetic expressions, which you extended with an additional floating point value type
- boolean expressions (including comparisons), if expressions, conditionals

This assignment will add further extensions. Use the `Eval.hs` and `Syntax.hs` you submitted for the previous assignment. The pack contains a new version of `SimpleTests.hs` and an updated `SExpression` module. The updated `SimpleTests` now allows printing simple statistics: the number of tests run and how many passed / failed. See `test_toString` in `SExpression.hs` for an example.

Questions

As before, where applicable, the questions require you to do the following:

(a) extend the BNF specification with the appropriate productions,
(b) extend any appropriate datatypes with new constructors as needed,
(c) extend the translation functions to and from s-expressions, including `valueToSExpression` (if applicable),
(d) implement the semantics in `eval` (with changes to `subst` as needed), and
(e) write tests for any extensions to keep maximal coverage.

---

1. We will add our first composite value type to `protoScheme`: ordered pairs. Pairs are formed using the `pair` constructor.[1] The semantics of the constructor is to evaluate the two given expressions and construct a *pair value*, containing the two values. To select elements of pairs, we will use selectors `left` and `right` for selecting the first and the second element, respectively.[2]

   ```
   <Expr> ::= ...
            | (pair <Expr> <Expr>)
            | (left <Expr>)
            | (right <Expr>)
   ```

   The selectors should satisfy the following equations:

[1] This is a departure from standard Scheme, where pairs are formed using `cons`.
[2] In Scheme, the selectors are `car` and `cdr`.

```
(left (cons a b)) = a
(right (cons a b)) = b
```

Note that the s-expression datatype now contains a case for pairs, strangely called `Dotted`. This is only for representing *pair values*, and is to be used as an output in `valueToSExpression`. This means you shouldn't handle it in `fromSExpression`.

2. Implement type predicates, which evaluate their argument and return *true* if the value is of the corresponding type and *false* if the value is of a different type.

```
<Expr> ::= ...
         | (real? <Expr>)
         | (integer? <Expr>)
         | (number? <Expr>)
         | (boolean? <Expr>)
         | (pair? <Expr>)
```

   - `real?` returns *true* only if the value is a real number
   - `integer?` returns *true* only if the value is an integer
   - `number?` returns *true* only if the value is a number
   - `boolean?` returns *true* only if the value is a boolean
   - `pair?` returns *true* only if the value is a pair

   Examples:

   - `(real? 3.14)` ⇒ `#t`
   - `(integer? 31)` ⇒ `#t`
   - `(number? 3.14)` ⇒ `#t`
   - `(boolean? #f)` ⇒ `#t`
   - `(boolean? (pair 1 2))` ⇒ `#f`
   - `(pair? (pair 1 2))` ⇒ `#t`
   - `(number? (left (pair 1 #t)))` ⇒ `#t`
   - `(number? (right (pair 1 #t)))` ⇒ `#f`

   (⇒ means "evaluates to")

3. Based on the code example from a recent lecture (which is available online), introduce a syntax category for *programs*. A program is a sequence of global definitions, followed by a single expression. A definition can be either a function definition, introduced using `defun`, or a global variable definition, introduced using `define`. Note, that functions can now have *one or more arguments*. This is reflected both in the definition and the call site in the function call expression. We recommend starting with the example from the lecture (which implements functions with one argument) then extend the syntax and semantics to handle multiple arguments. Finally, add global value definitions.

```
<GlobalDef> ::= (defun <Variable> (<Variable> <Variable>*) <Expr>)
              | (define <Variable> <Expr>)


<Program> ::= <GlobalDef>* <Expr>

<Expr> ::= ...
         | (<Variable> <Expr> <Expr>*)
```

A global variable or function can be used inside a function body, in a global variable definition, or in the final expression. Local variables inside expressions should take precedence: only if a variable is not bound locally, should the evaluator check the global definitions. If no global definition is found, evaluation should fail.

For example:

```
(define x 32)

(let (x 3.14) x)
```

should evaluate to `3.14`,

```
(define x 32)

(let (x (* x 2)) x)
```

should evaluate to `64`,

```
(define x 32)

(let (y (* x 2)) x)
```

should evaluate to `32`, and

```
(define x 32)

(let (y (* x 2)) z)
```

should fail.

Name the datatype of programs `Program`.

Implement the function `programFromSExpression` which takes an s-expression and returns a program.

Implement the function `evalProgram` which processes global definitions and evaluates the final expression. Update the `runProgram` function in `Eval.hs` so that it takes `Program` instead of `Expr`.