Assignment 8

CS 4400 Programming Languages

Start early and come to us with questions.

Due: 11pm on Thursday, November 19, 2020

Submission:

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

   - `Assignment08.hs`
   - `Eval.hs`
   - `Syntax.hs`
   - `Repl.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, the file should contain a preamble following this template.

   ```
   {- |
   Module      :  Assignment08
   Description :  Assignment 8 submission for CS 4400.
   Copyright   :  (c) <your name>

   Maintainer  :  <your email>
   -}
   ```

   The file should contain two functions:

   - `main` which starts the REPL, similarly to Assignment 6
   - `allTests` which runs all tests for all modules

4. Every top-level definition must include a purpose statement (for functions) and a type signature, followed by one or more defining equations.

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.**

Purpose: Simplifying the evaluator, building a base library, desugaring function definitions.

State of the Union

The previous assignment asked you to compile a subset of `protoScheme` into pure λ-calculus using Church encodings. In this assignment we will return to working on the main evaluator. Your code base should contain a fairly mature evaluator for `protoScheme`, covering the following features.

1. Let-bindings and a variable references expressions
2. Arithmetic expressions with integer and floating point values
3. Boolean expressions, comparisons and equality checks, if expressions, conditionals
4. Pair values and selectors
5. Type predicates for integers, reals, numbers, pairs, and booleans
6. Global function definitions (with one or more arguments; recursive by default), function calls, global variable definitions

In this assignment, we will concentrate on refactoring and improving the existing evaluator, factoring some operations on values and implementing a base library (a "prelude"). We will also introduce lambdas (à la the Intermediate Student Language) and functions as values, desugaring function definitions into

Assignment Pack

The starter code pack for this assignment contains the following:

`Parser.hs` Changes:

- Functions added:

  ```
  parseSExpressions :: String -> Maybe [S.Expr]
  fromFile :: String -> IO (Maybe [S.Expr])
  ```

  The first one parses a string containing multiple s-expressions and returns a list of those s-expressions if successful. The second reads a list of s-expressions from a file. These can be useful if you want to write more complex test programs. Scheme-like line comments (`;` to the end of line) are ignored

- In addition to `( )`, s-expressions can be equivalently enclosed in `[ ]`, similarly to Racket

`SExpression.hs` As before or minor adjustments.

`Maps.hs` The type of maps has been made abstract, hiding the implementation. Now provides `fromList` for converting a list of key-value pairs to a map and `toList` for the inverse.

`SimpleTests.hs` & `SimpleTestsColor.hs` As before or minor adjustments.

`Result.hs` An implementation of the `Result` datatype. Contains conversion functions `toMaybe`, `fromMaybe`, `fromMaybe'` and `toIO` for converting between different monads.

Example programs (`example`*n*`.pss`) The pack also contains some example programs that you can use to test your interpreter. You can use `fromFile` (in `Parser`) with `runProgram` (after modifying the type – see below). Hint: if you want syntax highlighting for these files in your editor, Lisp or Scheme is the closest approximation.

Questions

Note: It is best not to tackle these questions in sequence one-by-one, but work on them simultaneously. For example, Q2 and Q5 are related by introducing two kinds of functions as values. Any refactoring of `eval` will benefit from reducing the number of abstract syntax constructors – keeping Q5 in mind will reduce the number of cases you need to modify for Q1.

---

1. If you haven't already, change your evaluator to use environments instead of substitution for local variables. Keep a separate environment for globals.
2. To allow defining anonymous functions, introduce `lambda` to the abstract syntax of the language. A lambda has a *list of arguments* and a body. Generalize function calls to allow any expression in the function argument (not just function names).

   ```
   <Expr> := ...
           | (lambda (<Variable>*) <Expr>)     -- anonymous functions
           | (<Expr>+)                         -- function call/application
   ```

   Don't forget to modify any relevant function in `Syntax.hs` and to add tests.
3. Instead of having a special case for function definitions in globals, implement `defun` as a *desugaring*. That means modify your program parser (`programFromSExpression`) to convert a `defun` s-expression into a combination of `define` and `lambda`. See also this Wikipedia article.
4. Change the monad for `eval` and `evalProgram` to `Result`, replacing all uses of `Just` in your evaluator with `return` and `Nothing` with a call to `fail` with an appropriate error message.
5. We have several value operations in our language, which behave pretty uniformly, such as arithmetic operations or comparisons. Each of these evaluates its operands to values, checks that the values are of the right type, and applies an operation to them, wrapping the result back as a value. If any of the arguments fail, the whole evaluation fails. This leads to repetitive code and every time we introduce a new operation, we have to introduce new clauses to all functions processing our abstract syntax. Here, we will remedy this by considering these operations as predefined functions (and values). Introduce a "primitive operation" value to your abstract syntax. This value should be able to represent built-in operations on values that

return a value (or fail). Primitive operations should be indistinguishable to the programmer: wherever we can use a function value, we can use a built-in operation.

Build a "base library" of operations: an environment called `base` which contains all predefined operations. If you are using the provided `Maps` module, Use `fromList` to build an environment from a list of variable-value pairs.

As a minimum, the following operations should be converted into primitives: `+`, `-`, `*`, `/`, `<`, `>`, `=`, `not`.

Add the operations `<=` (less than or equal) and `>=` (greater than or equal).

Extra credit: In addition to the operations listed above, move any additional candidate operations from the evaluator to the new base library. Each converted operation (and the corresponding reduction of the abstract syntax of `protoScheme`) will be assigned a small number of extra points.

6. Change the `runProgram` function to have the type `[S.Expr] -> Result S.Expr`. That is, a program should be expressed as a list of s-expressions. This will allow you (or us) to pair the function with `Parser.fromFile` to read programs from a file.

Here are some test programs, that should run with this version of the evaluator:

```
(defun even? (n)
  (and (integer? n)
       (or (= n 0)
           (odd? (- n 1)))))

(defun odd? (n)
  (and (integer? n)
       (and (not (= n 0))
            (even? (- n 1)))))

(pair (odd? 42) (even? 42))
```

```
(defun pair-map (f p)
  (pair (f (left p)) (f (right p))))

(pair-map (lambda (x) (* 2 x)) (pair 11 -2.5))
```

```
(defun fib (n)
  (if (<= n 1)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))

(fib 10)
```