Assignment 4

CS 4400 Programming Languages

Start early and come to us with questions.

Due: 11pm on Thursday, October 8, 2020

Submission:

1. Submit a zip-file `Assignment04.zip` containing only the following files via https://handins.ccs.neu.edu/courses/119:

   - `Assignment04.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 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      :  Assignment04
Description :  Assignment 4 submission for CS 4400.
Copyright   :  (c) <your name>

Maintainer  :  <your email>
-}

module Assignment04 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.**

Purpose: To extend an abstract syntax of a language, as well as its evaluator with conditionals and boolean operators. To practice writing test cases for new language features.

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.

This assignment will add further extensions. Use the `Eval.hs` and `Syntax.hs` you submitted for the previous assignment. The pack contains `SimpleTests.hs` and an updated `SExpression` module.

Questions

In the previous assignment we have extended the `protoScheme` language with a new numeric value type and overloaded our arithmetic operations. In this assignment we will introduce further data types, along with operations on them.

Booleans

1. Extend `protoScheme` with boolean values `#t` and `#f`. Booleans can be used both in expressions (alongside integers and floats) and be returned from the `eval` function. Amend all functions and tests you need to accommodate the change. The newly provided `SExpression.hs` contains an s-expression representation for booleans.

The answers to the following questions should contain these steps:

(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,
(d) implement the semantics in `eval`, and
(e) write tests for any extensions to keep maximal coverage.

2. Add an if-expression:

```
<Expr> ::= ...
         | (if <Expr> <Expr> <Expr>)
```

   The semantics of `if` is to evaluate the first expression (the condition). If it evaluates to true, evaluate the second expression, otherwise, evaluate the third expression.

3. Implement the basic boolean operations: `and`, `or`, and `not`. Extend your BNF spec with the following productions:

```
<Expr> ::= ...
         | (and <Expr> <Expr>)
         | (or <Expr> <Expr>)
         | (not <Expr>)
```

   Both `or` and `and` should be evaluated following so-called *short-circuit* semantics. That is, if the boolean result can be determined without evaluating the second argument, the evaluation should stop and the result should be returned. Here is a free test case for you:

   `(and #f (/ 42 0))` should evaluate to `#f` and not result in an error.

4. Add binary (= "operates on two arguments") comparison operators with:

   - "less than", `<`, which returns `#t` if the first operand is less than the second operand,
   - "greater than", `>`, which returns `#t` if the first operand is greater than the second one,
   - and "equals", `=`, which returns `#t` only if the two operands are equal.

   While `<` and `>` should only work for numeric values, `=` should work for *any* value type, provided that both operands evaluate to the same type of value.

5. Implement a conditional, `cond`, which should be familiar to you from the student languages of Fundies I / HtDP. A `cond` expression contains a list of clauses, where each clause contains a *condition* and an expression. The clauses are evaluated in the order they appear in the list. The expression from the first clause whose condition evaluates to `#t` is then evaluated for the final result and the remaining clauses are left unevaluated. Optionally, the last clause may be an `else` clause. If none of the previous clauses applied, the expression associated with `else` will be evaluated.

```
<Expr> ::= ...
         | (cond (<Expr> <Expr>)* )
         | (cond (<Expr> <Expr>)* (else <Expr>))
```

   In addition to the usual tests, provide test cases illustrating that an if-expression can be expressed using `cond`. In a comment in `Assignment04.hs` briefly answer the following questions:

   (a) Could any `cond` expression be expressed using just `if` expressions?

(b) One could say that the `else` clause is redundant. That is, if only the first form of `cond` (without the extra `else` clause) was available, we could we still express

```
(cond ((= x 5432) 1)
      ((= x #t) 0)
      (else -1))
```

without resorting to negating all the previous conditions. How?