Assignment 2

CS 4400 / CS 5400 Programming Languages

General

Due: Wednesday, October 16, 11:59pm.

Instructions:

1. Download the assignment pack from https://vesely.io/teaching/CS4400f19/m/cw/02/Assignment2.zip.
2. You can choose to complete this assignment as a pair. If you work as a pair, submit as a pair. Completing an assignment with a partner but submitting individually is considered cheating.
3. Submit via the Khoury Handin server: https://handins.ccs.neu.edu/ (the course is not yet set up on the server, so please wait before checking)
4. Complete the information in `Assignment2.hs` and submit modified Haskell files: `Env.hs`, `ABL.hs`, `StrictEnvABL.hs`, and `Assignment2.hs`.
5. You are free to add top-level definitions, but add them to the bottom of each file, below the separator line.
6. Each top-level function must include a type signature, followed by one or more defining equations.
7. Make sure your file loads into GHCi or can be compiled by GHC without any errors.
8. A complete solution will contain no `undefined`s and will implement all cases from `ABLExpr`.
9. Use the provided `SimpleTests` module to write your own tests in the `tests` function. Follow the examples provided in `Env`, `ABL`, and `StrictEnvABL`.

Grade: To calculate your grade, we will take the following into account:

a) Quality of submission: Does your code compile without errors? Did you follow the above steps?
b) Correctness: How well does it implement the specification?
c) QA: How well did you test your code?
d) Is your code readable?

Partial Functions in Haskel

This assignment is mainly made up of partial functions. A partial function (that is, a function which is not defined for all possible inputs) can be represented using the option type `Maybe`. The type is predefined by Haskell as follows:

```
data Maybe a = Nothing
             | Just a
```

For example, the integer division operation in Haskell, `div` throws an exception if the second argument is `0`. We can convert it into a partial function as follows:

```
maybeDiv :: Integer -> Integer -> Maybe Integer
maybeDiv _ 0 = Nothing
maybeDiv n d = Just (n `div` d)  -- n `div` d is a fancy way of writing div n d
```

Now `maybeDiv 5 0` will return `Nothing`, whereas `maybeDiv 5 2` will return `Just 2`.

To use (and compose) partial operations, we can use the case construct:

```
divideThenAdd2 :: Integer -> Integer -> Maybe Integer
divideThenAdd2 x y =
  case maybeDiv x y of
       Just z -> Just (r + 2)
       Nothing -> Nothing
```

Environments as Association Lists

Exercise 1 Complete `Env.hs` by implementing environments as *association lists*. An association list is a list of pairs – the first member is a variable, the second is the value. For example, `[("x", 1), ("y", 2)]` binds `"x"` to `1` and `"y"` to `2`. The most recent binding overrides any previous ones, that is, the environment `[("x", 100), ("x", 1)]` binds `"x"` to the value `100`, NOT `1`. Provide implementation for:

- `empty` – the empty environment
- `add` – add a binding for the given variable. That is `add "x" 10 env` binds `"x"` to the value `10` in the environment `env`
- `get` – find the binding for the given variable. This is a partial function: `get "x" [("x", 10)]` should return `Just 10`, while `get "y" [("x", 10)]` should return `Nothing`

The following laws, expressed in Haskell, should hold for the above functions. In other words, these Haskell expressions should always return `True` for any `x`, `y`, `v`, and `env`. Assume `x /= y`.

```
get x empty == Nothing
get x (add x v env) == Just v
get x (add y v env) == get x env
```

The ABL language

Syntax

The ABL language has the following syntax:

```
<ABLValue> ::= <Integer>                               // tag: Num
             | <Bool>                                  // tag: Bool

<ABLExpr> ::= <Variable>                               // tag: Var
            | <ABLValue>                               // tag: Val
            | (+ <ABLExpr> <ABLExpr>)                  // tag: Add
            | (- <ABLExpr> <ABLExpr>)                  // tag: Sub
            | (* <ABLExpr> <ABLExpr>)                  // tag: Mul
            | (/ <ABLExpr> <ABLExpr>)                  // tag: Div
            | (= <ABLExpr> <ABLExpr>)                  // tag: Eq
            | (&& <ABLExpr> <ABLExpr>)                 // tag: And
            | (|| <ABLExpr> <ABLExpr>)                 // tag: Or
            | (not <ABLExpr>)                          // tag: Not
            | (let1 (<Variable> <ABLExpr>) <ABLExpr>)  // tag: Let1
            | (if-else <ABLExpr> <ABLExpr> <ABLExpr>)  // tag: If
```

Exercise 2 The file `ABL.hs` contains the initial definitions for the `ABLValue` and `ABLExpr` data types. Following the examples in `ABL.hs`, add the remaining constructor definitions for `ABLExpr`, using tags from the BNF definition above as constructor names.

Exercise 3 In `ABL.hs`, complete the definition for `showABL`, which pretty-prints ABL into a string as s-expressions, based on the above BNF rules. Add new cases for each new construct you add to the language in exercises 7 and 8.

Semantics

Behavior of ABL's constructs is described below. *Current environment* for a construct refers to the set of bindings visible when starting the evaluation of the construct. In other words, it is the unmodified environment passed to the evaluator as an argument together with the expression.

Variables

```
<Variable>
```

A variable reference should evaluate to the value bound to it in the current environment. If the variable is not in scope, the evaluation should result in `Nothing`.

Values

```
<ABLValue>
```

A value should trivially evaluate to itself.

Arithmetic Operations

```
(+ <ABLExpr> <ABLExpr>)
(- <ABLExpr> <ABLExpr>)
(* <ABLExpr> <ABLExpr>)
(/ <ABLExpr> <ABLExpr>)
```

The operands should be evaluated left to right. Then the corresponding operation should be applied to the values. If any of the operands do not evaluate to integer values, the evaluator should return `Nothing`. If the right operand of division (`/`) evaluates to `0`, the evaluator should return `Nothing`.

Equality

```
(= <ABLExpr> <ABLExpr>)
```

Operands should be evaluated left to right. Then the values should be compared for equality. If the values are of a different type, the evaluator is to return `Nothing`.

Boolean Operations

```
(and <ABLExpr> <ABLExpr>)
(or <ABLExpr> <ABLExpr>)
(not <ABLExpr>)
```

The operands should be evaluated left to right. Then the corresponding operation should be applied. If any of the operands do not evaluate to boolean values, the evaluator shall return `Nothing`.

Single Let Bindings

```
(let1 (<Variable> <ABLExpr>) <ABLExpr>)
```

An expression `(let1 (x e1) e2)` should be evaluated as follows. First evaluate the left expression `e1` with the current set of bindings. Then the right expression `e2` should be evaluated with the same set of bindings, except `x` is bound to the value corresponding to `e1`. If `e1` fails to evaluate, then the evaluation of the whole expression result should return `Nothing`.

Conditional

```
(if-else <ABLExpr> <ABLExpr> <ABLExpr>)
```

A conditional expression `(if-else e1 e2 e3)` should be evaluate by first evaluating `e1` to a boolean value. If the result is *true*, then the value resulting from evaluating `e2` shall be returned. If the result is *false*, then the result of evaluating `e3` should be returned. If `e1` does not evaluate to a boolean, evaluation should return `Nothing`.

Complete the definitions in `StrictEnvABL.hs`.

Exercise 4 Checking the type of values before applying each operation is repetitive and tiresome. It is therefore useful to abstract this process into two higher-order functions: `applyIntegerBinOp` for applying integer operations, and `applyBoolBinOp` for applying boolean operation. Then, for example, to perform addition on two `ABLValues`, we can simply call `applyBinIntegerOp (+) v1 v2`. We have implemented the former for you. Your task is to complete `applyBoolBinOp`, following the example in the file.

Exercise 5 Complete the evaluator for ABL, `evalABL`, implementing the behavior of ABL constructs as described above. Use `applyIntegerBinOp` and `applyBoolBinOp` where appropriate. Note: For integer division, use the function `div`.

Exercise 6 Complete the function `scopeCheck` which checks if all variables in an ABL expression are defined before they are used. Trivial examples: for `(let (y 10) (+ y y))` the function should return `True`, while for `(let (y 10) (+ y x))` it should return `False`.

Extensions to ABL

For each extension, update the syntax in `ABL.hs`, as well as the relevant functions in `ABL.hs` and `StringEnvABL.hs`.

Exercise 7 Implement a new construct which "forgets" the current environment and evaluates its argument as if it was a top-level expression (that is, in an empty environment).

The syntax is as follows:

```
...
    | (fresh-env <ABLExpr>)      // tag: Fresh
```

Extend `scopeCheck` to handle this new construct.

Exercise 8 Implement an improved, generalized let-binding construct, named `let*`:

```
...
    | (let* ((<Variable> <ABLExpr>) ...) <ABLExpr>)  // tag: LetStar
```

where `(<Variable> <ABLExpr>) ...` is a, possibly empty, list of bindings. The corresponding Haskell constructor has the following shape:

```
  | LetStar [(Variable, ABLExpr)] ABLExpr
```

The evaluation of `(let* ((x1 e1) (x2 e2) ... (xn en)) e)` is equivalent to `(let1 (x1 e1) (let1 (x2 e2) ... (let1 (xn en) e)))`. If the list of bindings is empty, as in `(let1 () e)`, then the evaluation is equivalent to `e`. For evaluating `let*` implement the auxiliary function `unfoldLetStar` which, given a list of bindings, outputs the corresponding expression formed of nested `let1` expressions. For example:

```
unfoldLetStar [] e == e
unfoldLetStar [(x1, e1), (x2, e2)] e == Let1 x1 e1 (Let1 x2 e2 e)
```

Implement the evaluation of `let*` with the help of `unfoldLetStar`.

Extend `scopeCheck` to handle this construct.