1 Environments and Closures

1.1 Nano: Functions

We now want to add functions. In particular, we want to add

- Lambda abstractions (i.e., function definitions).
- Applications (i.e., function calls).

Our grammar would look something like

```
e :: n
    | e1 + e2
    | e1 - e2
    | e1 * e2
    | x
    | let x = e1 in e2
    | \x -> e                -- abstraction
    | e1 e2                  -- application
```

(Quiz.) What should this evaluate to?

```
let inc = \x -> x + 1
in
    inc 10
```

(a) Undefined variable `x`

(b) Undefined variable `inc`

(c) `1`

(d) `10`

(e) `11`

The answer is **E**. The idea is that we're binding `inc 10` to the `let`-binding, so `inc 10` would call `inc`, which would return `11`.

How would we represent functions?

```
data Expr = Num Int              -- n
        | Bin Binop Expr Expr   -- e1 op e2
        | Var Id                -- x
        | Let Id Expr Expr      -- let x = e1 in e2
        | Lam ???               -- \x -> e
        | App ???               -- e1 e2
```

For the lambda expression, we would have `Lam Id Expr`, since we want a new variable with an expression. For application, we would have `App Expr Expr` since we want two expressions. Thus, our final definition would be

```
data Expr = Num Int              -- n
        | Bin Binop Expr Expr   -- e1 op e2
        | Var Id                -- x
        | Let Id Expr Expr      -- let x = e1 in e2
        | Lam Id Expr           -- \x -> e
        | App Expr Expr         -- e1 e2
```

(Example.) Suppose we want to represent the expression

```
let inc = \x -> x + 1
in
    inc 10
```

using our definition of `Expr` above. This can be done like so:

```
fun1 = Let "inc"
        (Lam "x" (Bin Add (Var "x") (Num 1)))
        (App (Var "inc") (Num 10))
```

We now want to implement functions in our `eval`.

```
eval :: Env -> Expr -> Value
eval env (Num n)          = n
eval env (Bin op e1 e2)   = evalOp op (eval e1) (eval e2)
eval env (Var x)          = lookup x env
eval env (Let x e1 e2)    = eval env' e2
    where
        val  = eval env e1
        env' = (x, val) : env
eval env (Lam x e)        = ???
eval env (App e1 e2)      = ???
```

Recall that, when we have a `let`-body, we put a binding of the variable to its definition that it evaluates to in the environment. So far, all of the values that we've been using are *integers*; however, we cannot store our function as an integer.

1.1.1 Rethinking Values

Until now, we said that a program evaluates to an integer, or fails. However, programs like

```
\x -> x + 1
=> Increment Function

let f = \x y -> x + y
in
    f 1
=> Increment Function
```

will not work, since these are all *functions*. Therefore, we want a program that evaluates to an **integer or a function**, or fails. Thus, we need to change our definition of values.

```
data Value = VNum Int
           | VFun Id Expr
```

So, the value of a function is its code. Hence, our grammar for values is defined by

```
v ::= n
    | <x, e>        -- formal + body
```

We can now try to implement this. But, note that we changed `Value`, so it follows that we need to make some adjustments in our code.

```
eval :: Env -> Expr -> Value
eval env (Num n)          = VNum n                              -- Changed This
...
```

Likewise, for `evalOp` (the helper function), we need to do:

```
evalOp :: Binop -> Value -> Value -> Value
evalOp Add  (VNum n1) (VNum n2)    = VNum (v1 + v2)
evalOp Sub  (VNum n1) (VNum n2)    = VNum (v1 - v2)
evalOp Mult (VNum n1) (VNum n2)    = VNum (v1 * v2)
evalOp _    _         _            = error "Unsupported operation"
```

This way, we can compute any values of type `VNum` while throwing an error if we end up with a `VFun` somehow.

1.1.2 Evaluating Lambdas

Now, we will deal with how to evaluating a lambda.

```
...
eval env (Lam x e)        = ???
```

A lambda should just evaluate to itself. So, we have:

```
...
eval env (Lam x e)        =  VFun x e
```

1.1.3 Evaluating Applications

Now, we need to deal with the application case. To motivate this, consider again

```
let inc = \x -> x + 1
in
    inc 10
```

To evaluate `inc 10`, we want to

1. Evaluate `inc`, get `<x, x + 1>`
2. Evaluate `10`, get `10`
3. Evaluate `x + 1` in an environment *extneded* with `[x := 10]`

Thus,

```
...
eval env (App e1 e2)      = eval env' body
    where
        -- You should do pattern matching in a helper function (like with 'evalOp'),
        -- esp. since you can handle any possible errors yourself instead of getting
        -- a cryptic error message.
        (VFun x body)   = eval env e1
        v2              = eval env e2
        env'            = (x, v2) : env
```

(Quiz.) What should this evaluate to?

```
let c = 1
in
    let inc = \x -> x + c
    in
        inc 10
```

(a) Undefined variable `x`

(b) Undefined variable `c`

(c) `1`

(d) `10`

(e) `11`

The answer is **E**.

Remark: This example is slightly more involved than previous examples since we introduce the function definition appears after a variable definition. Although this example is still simple, the next example will be slightly more complicated.

(Quiz.) What should this evaluate to?

```
let c = 1
in
    let inc = \x -> x + c
    in
        let c = 100
        in
            inc 10
```

(a) Error: multiple definitions of `c`

(b) `11`

(c) `110`

The answer is **B**. By the time we define the `inc` function, there is only one `c` in our environment (when `c` is `1`).s

The answer is not A because we allow multiple definitions.

Remark: This is a classic example of static vs. dynamic scoping.

- For static (or lexical) scoping, which is what this example highlights, each occurrence of a variable refers to the most recent binding in the program text. The definition of each variable is unique and known statically. And, finally, it guarantees referential transparency[1].
- For dynamic scoping, each occurrence of a variable refers to the most **recent** binding during program execution. Thus, we can't tell where a variable is defined just by looking at the function body. Hence, this violates referential transparency.

(Quiz.) Which scoping does our `eval` function, specifically `Lam` and `App`, implement?

(a) Static

(b) Dynamic

(c) Neither

[1] The same expression must always evaluate to the same value. In particular, a function must always return the same output for a given input.

The answer is **B**. To see why this is the case, consider the following example:

```
let c = 1                     -- []
in                            -- ["c" := 1]
    let inc = \x -> x + c     --
    in                        -- ["inc" := <x, x + c>, "c" := 1]
        let c = 100           --
        in                    -- ["c" := 100, "inc" := <x, x + c>, "c" := 1]
            inc 10
```

So, what ends up happening is that, when we evaluate `inc 10`, we shave our *extended environment*

```
["x" := 10, "c" := 100, "inc" := <x, x + c>, "c" := 1]
```

and then the first `c` will be found first (so `c` is 100).