1 Environments and Closures

1.1 Nano: Variables

We now need to add variables. Hence, we modify the grammar like so:

```
e :: n
    | e1 + e2
    | e1 - e2
    | e1 * e2
    | x             -- New
```

This can be represented by the datatype[1]:

```
type Id = String
data Expr = Num Int               -- Number
    | Bin Binop Expr Expr         -- Binary Expression
    | Var Id                      -- Variable
```

We now need to extend the evaluation function.

(Quiz.) What should the following expression evaluate to?

```
x + 1
```

(a) `0`

(b) `1`

(c) Runtime Error.

The answer is **C**. We don't know what the value of `x` is.

Clearly, variables aren't useful unless we can somehow map variable names to values.

1.1.1 Environment

An expression is evaluated in an **environment**. It's like a phone book that maps variables to values.

```
["x" := 0, "y" := 12, ...]
```

We can represent an environment using the following type:

```
type Env = [(Id, Value)]
```

1.1.2 Evaluation in an Environment

We can write

```
eval env expr => value
```

to mean that evaluating `expr` in the *environment* `env` should return `value`.

(Quiz.) What should the result of the following code be?

```
eval ["x" := 0, "y" := 12, ...] (x + 1)
```

(a) `0`

(b) `1`

[1] We don't plan on introducing type checking here.

(c) Runtime Error.

The answer is **B**.

To evaluate a variable, we can just look up its value in the environment.

1.1.3 Evaluating Variables

We now need to update our evaluation function to take the environment as an argument.

```
eval :: Env -> Expr -> Value
eval env (Num n)            = n
eval env (Binop op e1 e2)   = evalOp op (eval env e1) (eval env e2)
eval env (Var x)            = lookup x env
```

Now that we have variables, we now need to find some way of *adding* variables to the environment. In other words, how do variables get into the environment?

1.2 Nano: Let-Bindings

We now need to add `let`-bindings. Our grammar needs to be updated:

```
e :: n
    | e1 + e2
    | e1 - e2
    | e1 * e2
    | x
    | let x = e1 in e2
```

For example, if our environment is `[]` and our expression is `let x = 2 + 3 in x * 2`, then we would end up with `10`. Notice that `x` isn't in our environment; rather, we introduced `x` through a let-binding. Hence, we need to extend the representation of expressions, or the datatype.

```
data Expr = Num Int              -- Number
        | Bin Binop Expr Expr    -- Binary Expression
        | Var Id                 -- Variable
        | Let Id Expr Expr       -- Let-binding
```

But, how do we extend the `eval` function to account for let-bindings?

(Quiz.) What should this evaluate to?

```
let x = 5
in
    x + 1
```

(a) `1`

(b) `5`

(c) `6`

(d) Error: unbound variable `x`

(e) Error: unbound variable `y`

The answer is **C**. `x` is bound to the value `5`, so `5 + 1` gives us `6`.

(Quiz.) What should the following evaluate to?

```
let x = 5
in
    let y = x + 1
    in
        x * y
```

(a) `5`

(b) `6`

(c) `30`

(d) Error: unbound variable `x`

(e) Error: unbound variable `y`

The answer is **C**. Once again, we've bound `x` to 5, then bound `y` to `5 + 1`. Thus, we get the value `5 * (6 + 1)`, which is 30.

(Quiz.) What should the following evaluate to?

```
let x = 0
in
    (let x = 100
    in
        x + 1
    ) + x
```

(a) `1`

(b) `101`

(c) `201`

(d) `2`

(e) Error: multiple definitions of `x`.

The answer is **B**. Here, we note that the inner `x` is shadowing the outer `x`. Hence, the inner `x + 1` is `101`.

1.2.1 Principle: Static (Lexical) Scoping

Every variable use (occurrence) gets its value from its most *local definition* (binding). In a pure language, the value never changes once defined, thus it's easier to tell by looking at a program where the variable's value came from.

1.2.2 Implementing Lexical Scoping

How would we implement this?

(Example.) Consider

```
let x = 5
in
    x + 1
```

Note that its environment is given by

```
let x = 5       -- []
in              -- | [x := 5]
    x + 1       -- |
```

(Example.) Consider

```
let x = 5           -- []
in                  -- | [x := 5]
    let y = x + 1   -- |
    in              -- | | [y := 6, x := 5]
        x * y       -- | |
```

(Example.) Consider

```
let x = 0           -- []
in                  -- | [x := 0]
    (let x = 100    -- | [x := 0]
    in              -- | | [x := 100, x := 0]
        x + 1       -- | |
    ) + x           -- |
```

1.2.3 Evaluating `let` Expressions

To evaluate `let x = e1 in e2` in `env`, we need to do the following.

1. Evaluate `e1` in `env` to `val`.
2. *Extend* `env` with a mapping `["x" := val]`.
3. Evaluate `e2` in this extended environment.

So, we can now extend the `eval` function like so:

```
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
```

(Example.) Let's suppose we wanted to represent

```
let x = 5
in x + 1
```

Using our definition above, we can write this out as

```
let1 = Let "x"
           (Num 5)
           (Bin Add (Var "x") (Num 1))
```