1 Environments and Closures

1.1 Nano: Functions

1.1.1 Evaluating Applications

Our `eval` function uses dynamic binding, but we want static binding. So, we instead want to "preserve" the environment at the point of the function definition. In essense, to implement static scoping, we need to

1. Freeze the environment in the function's value at *definition*.
2. Use the frozen environment, instead of the current environment, to evaluate the body at *call*.

So, in our example, we would have:

```
let c = 1                   -- []
in                          -- ["c" := 1]
    let inc = \x -> x + c   --
    in                      -- ["inc" := <fro, x, x + c>, "c" := 1]
                            --      where fro = ["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 have our *extended environment*, which extends the *frozen environment* instead of the general environment:

```
["x" := 10, "c" := 1]
```

This gives us the desired answer of 11.

1.1.2 Function Values as Closures & Evaluating Function Calls

To implement lexical scoping, we want to represent function values as closures. A closure is essentially a lambda abstraction which has a formal, body, *and* environment at function definition. Our updated grammar is then given by

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

where `env` is defined by

```
env ::= []
      | (x := v) : env
```

Hence, our updated `Value` representation is

```
dataValue = VNum Int
          | VClos Env Id Expr
```

How do we modify our `eval` for `Lam`? Likewise, how do we modify our `eval` for `App`? (HW4[1].)

1.1.3 Advanced Features of Functions

In particular, what can `eval` support? Can it support the following?

- Returning functions from functions (partial application).
- Functions taking functions as arguments (higher-order functions).
- Recursion.

[1]See lecture notes for hints.

(Quiz.) What should the following evaluate to?

```
let add = \x y -> x + y
in
    let add1 = add 1
    in
        let add10 = add 10
        in
            add1 100 + add10 1000
```

(a) Runtime error.

(b) `1102`

(c) `1120`

(d) `1111`

The answer is **D**. Note that `add1` takes in an argument and adds 1 to it. `add10` takes in an argument and adds `10`. So, `add1 100` should give us `101` and `add10 1000` should give us `1010`, so the final answer is `1111`.

In particular, partial application is supported.

(Quiz.) What should the following evaluate to?

```
let inc = \x -> x + 1
in
    let doTwice = \f -> (\x -> f (f x))
    in
        doTwice inc 10
```

(a) Runtime error.

(b) `11`

(c) `12`

The answer is **C**.

So, higher-order functions have been achieved.

(Quiz.) What does this evaluate to?

```
let f = \n -> n * f (n - 1)
in
    f 5
```

(a) `120`

(b) Evaluation does not terminate

(c) Error: unbound variable `f`

The answer is **C**; when we call `f` recursively, we do not have `f` in our environment.

2 Lexing and Parsing

Here, we won't actually be using Nano. Rather, we will be using an even simpler language – a calculator. Its AST representation can be defined by

```
data Aexpr = AConst  Int
    | AVar    Id
    | APlus   Aexpr Aexpr
    | AMinus  Aexpr Aexpr
    | AMul    Aexpr Aexpr
```

Its evaluator function can be defined by

```
eval :: Env -> Aexpr -> Value
...
```

Its implementation is very similar to Nano. Thus, using the evaluator:

```
-> eval [] (APlus (AConst 2) (AConst 6))
8

-> eval [("x", 16), ("y", 10)] (AMinus (AVar "x") (AVar "y"))
6
```

Notice that this is very tedious; most of us are used to writing programs as text. How do we do this? In particular, we want to write a function that *parses* the AST for us.

```
parse :: String -> Aexpr
```

We will incorporate a two-step strategy. In particular, consider the sentence: *He ate a bagel.* How do we read it?

- First, we split the sentence into words: `["He", "ate", "a", "bagel"]`.
- Then, we relate the words to each other. `He` is the subject, `ate` is the verb, and so on.

We can do the same thing when "reading" programs.

2.1 Step 1: Lexing

We want to first convert a string to a bunch of tokens. A string is simply a list of characters:

```
229 + 98 * x2
```

The idea is to aggregate characters that "belong together" into **tokens**, or the "words" of the program. So,

```
229 PLUS 98 TIMES x2
```

Often, we distinguish tokens of different kinds based on their format:

- All numbers: integer constant.
- Alphanumerica: starts with a letter: identifier.
- `+`: Plus operator.
- And so on.

2.2 Step 2: Parsing

Now, we want to convert the tokens to AST. This is usually difficult, so we make use of a set of tools known as **parser generators**.

- Given the description of the token format, generates a lexer.
- Given the description of the grammar, generates a parser.

We will be using parser generators, so we only care about how to describe the token format and the grammar.