Data Abstraction Lecture 1

Wednesday, October 28, 2009
3:49 PM

In previous lectures, we talked a lot about abstracting complex control flow under a clean interface:

- We replaced recursion with a while construct when while was more comprehensible than recursion
- We simplified complex iterators with coroutines
- If we had more time, we would avoid returning and checking error flags by instead using exceptions.

If there is control flow abstraction, what's the other side of the coin? *Data abstractions*, or hiding data representations. Why do we want to do that?:

- Hide representation of a Set from the users of a Set. Is it a hash table? A linked list? A red black tree?
- Allow changing the representation after clients of the set have been written. Optimize it without breaking the client.
- Allow reusing and implementations (eg with inheritance)

Let's start with objects, things that

i) contain fields (encapsulate state)
ii) whose reference we can pass around as a value (they are first class)

Modeling objects. Can we create objects with our simple base language with eager evaluation and lexical scoping?

```
E -> n | v | E + E | cond(E, E, E)
  |  lambda(Id) { S } | E ( E )
S -> E | def v = E | v = E | S ; S | print E
```

Which construct allows creating a location that persists as long as we have a reference to it? Closure.

```
def factory(oldVal) {
    lambda (newVal) {
        def t = oldVal; oldVal = newVal; t
    }
}
def x = factory(1)
def y = factory(2)
print x(3) --> 1
print y(4) --> 2
print x(5) --> 3
```

How can we turn a closure into an object?

- Example 1, from Cs61A: Mutation is just an assignment.
- Example 2, from Lua: The Single-Method Approach

However, we won't use closures in 164. We will use hashes. We'll need less sugar with them. A hash is a set of (key,value) pairs. Like Python dict.

{}	expression that evaluates to a new hash
h[k]	return value associated in h with key k
h[k] = v	assign, in hash h, the key k the value v

How are these useful for creating objects?

> Can you think of ways how to model objects with hashes? There must be more than one way!

Note: this lecture is based on the language Lua described beautifully here

Our way of modeling objects:

```
def x = {}   # empty hash
x[1] = "hello"
print x[1]
```

A hash can act as an structure (an object with fields):

```
def person = {}
person["age"] = 21
print person["age"]  --> 21
```

hash["field"] is clunky so we'll sugarcoat it, translating to the expression get (i.e., E[E]) and the statement put (E[E] = E) over hashes.

> *E.ID --> E["ID.name"]*
> *E.ID = E --> E["ID.name"] = E*

Now the syntax is more familiar and hence more readable (for those of us used to C++ or Java).

```
def person = {}
person.age = 21
print person.age
```

More sugar?

```
a.x = 1
a.y = 2
```

can be replaced with

```
a = { x=1, y=2 }   # this is a hash literal
```

A hash literal is like an int literal (eg 7) or string literal (eg "asdb") except that it evaluates to an hash entity.

Q: does the '=' in the hash literal have anything to do with the assignment statement ID = E ?

How about **methods**? The structs can also contain functions, which will make these structs look like objects (encapsulate data and methods). We simply assign the function to the field printMe.

```
a.printMe = lambda(self) { print self.x; print self.y }
```

We can now invoke the function in almost the same way as in Java.

```
a.printMe(a)
```

The annoyance (and redundancy!) is that we need to pass value of a (the receiver) as an actual argument. To avoid the need to pass the receiver explicitly, we'll introduce another syntactic sugar.

```
e:f(argslist)  -->  def $t=e; $1.f($1,argslist)
```

With this syntactic facility, we can replace

```
a.printMe(a)   with   a:printMe()
```

We will also add syntactic sugar that allows you to define an object's method. We already know that

```
def move(p) { p.x = p.x + 4.5 }
```

is translated to

```
def move = function (p) { p.x = p.x + 4.5 }
```

You can define such sugar also for methods

```
def Foo:changeMe (v) { self.x = v }

     -->

Foo.changeMe = lambda (self,v) {    # self argument added
     self.x = v
}
```

The usage is as you would expect.

```
a:changeMe(v)
a:printMe()
```

Modules. let's support modules, whose motivation is to avoid cluttering the global name space, and avoid name conflicts between your code and libraries.

The example should explain itself. Just one comment: the table stored in List creates a namespace, which encapsulates the methods of the module. To

understand the example, ask yourself where the fields first and last are kept? In the module or in the lists created from the module?

```
List = {}   # a new namespace (this is our module)
function List.new () {
    return {first=0, last=-1}
}
function List.push(list,value) {
     local first = list.first
     list.first = first-1
     list[first] = value
}

myList = List.new()
List.push(myList, 7)

print(myList.first, myList.last, myList[0])
```

Metatables. Let us add new operators to the hash, via overloading. We will do it via extensions to the language through metatables. Metatables allow you to (re)define what the program does when it, say, fails to find a key in a hash.

Metatables also allow you to define operators like + for hashes. In this example, we use define + to add elements into a table (which will be used as an array).

First, we register mt as a metatable for table v.

```
v={}
mt={}
setmetatable(v,mt)
```

Now we add into the metatable a field __add. This field contains a function that appends an element into an array.

TODO: Define len(x)

```
mt.__add = lambda(x,y) {
     print("our custom add is called.  x,y:", x, y)
     x[len(x)+1] = y
     return x
}
```

The function __add stored in v's metatable is called whenever we perform an addition on v, as shown here.

```
w=v+5+7
```

This code outputs

```
our custom add is called.  x,y: table: 0039D158  5
our custom add is called.  x,y: table: 0039D158  7
```

Notice that both w and v point to the same table:

```
print(table.concat(v,","))  -- outputs "5,7"
print(table.concat(w,","))  -- outputs "5,7"
```

Prototypes. Let us support a sort of inheritance. There is another metatable entry, named __index, consulted when a key lookup fails on a hash:

```
-- create a namespace
Window = {}
-- create the prototype with default values
Window.prototype = {x=0, y=0, width=100, height=100, }
-- create a metatable
Window.mt = {}
-- declare the constructor function
function Window.new (o)
     setmetatable(o, Window.mt)
     return o
end
a
```

let us define the __idnex metamethod:

```
Window.mt.__index = function (table, key)
      return Window.prototype[key]
end
```

Now you can say

```
w = Window.new({x=10, y=20})
print(w.width)    --> 100
```