Hack Your Language!

CSE401 Winter 2016

Introduction to Compiler Construction

Ras Bodik
Alvin Cheung
Maaz Ahmad
Talia Ringer
Ben Tebbs

Lecture 12: Data Abstraction

Objects
Inheritance
Prototypes

Announcements

PA3 due next Tuesday 11PM

Makeup lecture this Friday

- Same place and time
- We will be video taping the lecture

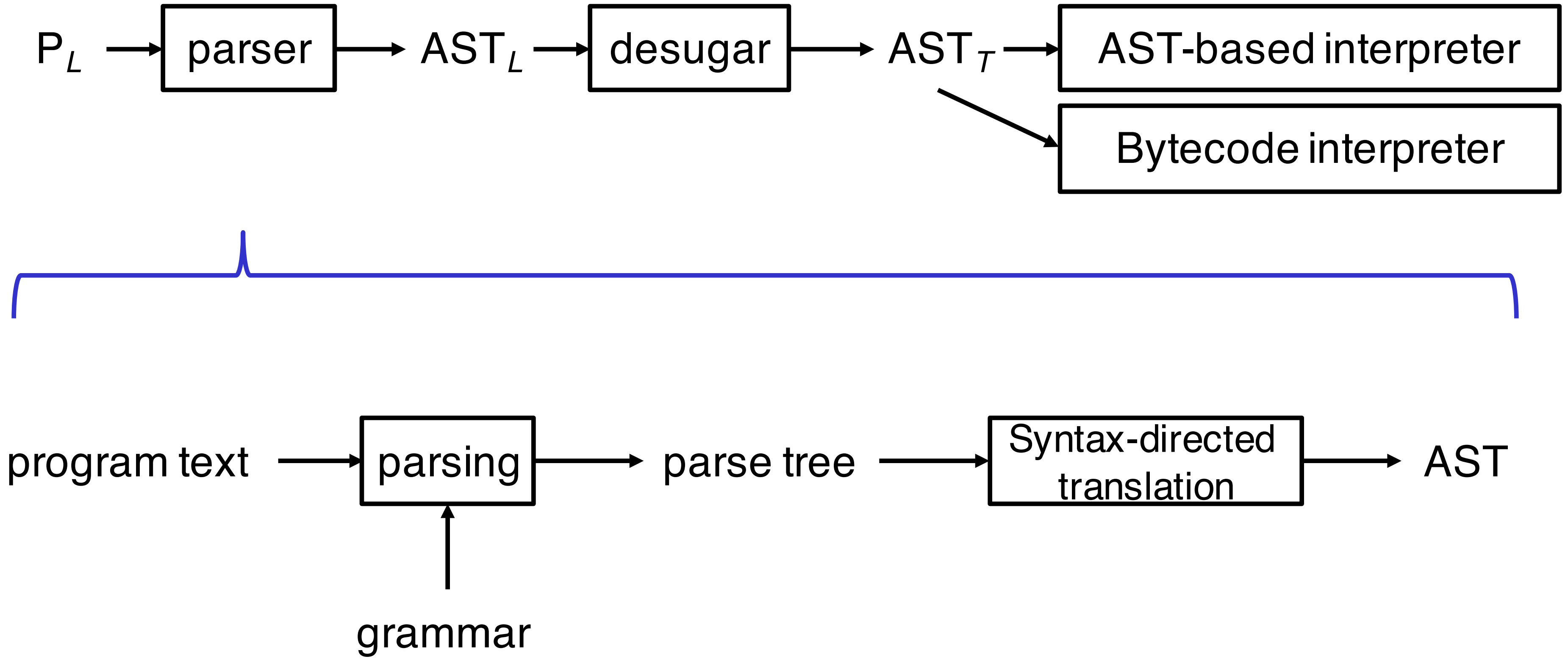
Final project update

We are posting comments to your proposals.

Give us permission to add comments. Turn on email notifications.

Milestone 1 will be due next week: coding tutorial

Next two lectures will expand on the parser



Objects

A review

The 401 language so far

Our constructs dealt with **control abstraction:**

hiding complex changes to program control flow under suitable programming language constructs

Examples:

- iterators, built on closures
- backtracking in regexes, built with coroutines
- reactive programming, built with continuations

Data abstraction

If there are control abstractions, there must also be **data abstractions**

- for hiding complex data representations
- we've seen them in the d3 language
    - selections, transitions

Constructs that abstract data representations:

Language construct for data abstraction	**Hides what details of implementation**
Floating point	Mantissa, exponents
Relations in databases	How rows and columns are stored on disk
Maps and dictionaries	Is it a list, hashtable, etc?

Objects (review from 143)

What are objects

- state (attributes) and
- code (methods)

Why objects?

abstraction: hide implementation using encapsulation

Why inheritance?

reuse: specialization of an object's behavior reuses its code

Minimal core language to support objects?

Can we implement objects as a library?

that is, without changes to the interpreter?

It's the very familiar question:

What is the smallest language on which to build to objects?

Our language already supports closures

which are similar to objects: they carry code *and* state

Can we build objects from this existing mechanism?

rather than any adding lots "native" support for objects?

Summary

Data abstractions support good software engineering

- ie, writing large software that can be maintained
- easing maintenance thanks to code modularity

Modularity is achieved by:

- **reuse:** use existing libraries by extending/modifying them
- **code evolution:** change implementation without changing the interface, leaving client code unchanged

Objects carry code and state

- like closures
- so we will try to build them on top of closures first

Try #1: Objects as Closures

We have seen closure-based objects already

Where did we use closures as objects?

Iterators are <u>*single-method*</u> objects

- on each call, an iterator returns the next element and advances its state to the next element
- in essence, they are single-method objects that support the `next()` method

Multi-method closure-based objects

Can we overcome the single-method limitation?

Yes, of course:

```
d = newObject(0)
print d("get")     --> 0
d("set", 10)
print d("get")     --> 10
```

Multi-method object represented as a closure

```
function newObject (value)
    function (action, v) {
        if (action == "get“) {
            value
        } else if (action == "set“)  {
            value = v
        } else {
            error("invalid action")
}   }   }
```

Summary

Closures carry own state and code

  so we can use them as objects

Closures support only one operation (function call)

  so we can support only one method

By adding an argument to the call of the closure

  we can dispatch the call to multiple "methods"

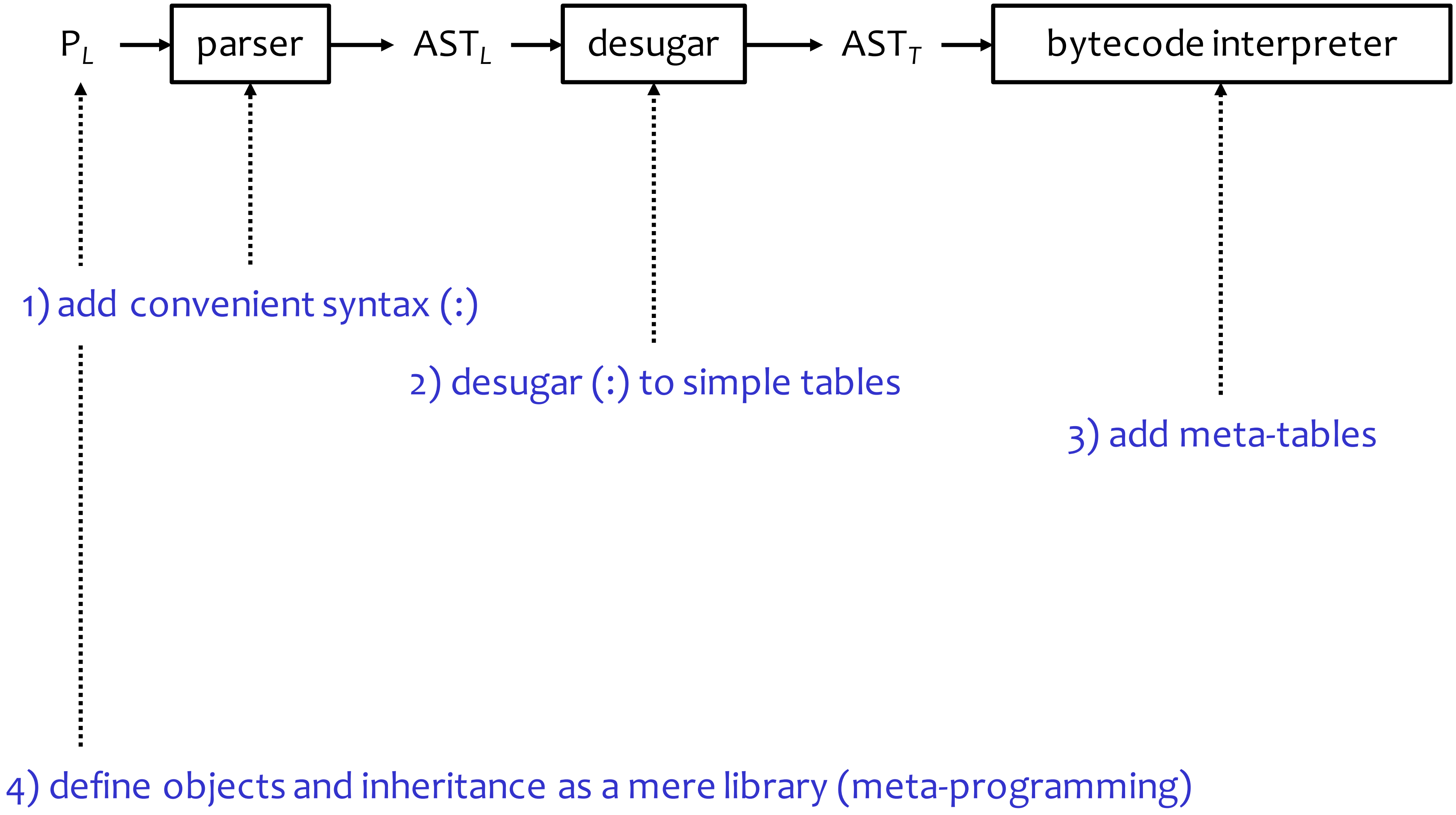
But unclear if we can support inheritance

  i.e., specialize an object by replacing just one of its methods

Try #2: Objects as tables

strawman version

Implementation plan for objects, inheritance


P_L
parser
AST_L
desugar
AST_T
bytecode interpreter
1) add convenient syntax (:)
2) desugar (:) to simple tables
3) add meta-tables
4) define objects and inheritance as a mere library (meta-programming)

Reading for today

Required reading:

Chapter 16 in PiL:

http://www.lua.org/pil/contents.html#16

(Also linked from PA3 assignment doc)

Recall 401 dicts (inspired from Lua)

Create a table

```
{}
{ key1 = value1, key2 = value2 }
```

Add a key-value pair to table (or overwrite a k/w pair)

```
t = {}
t[key] = value
```

Read a value given a key

```
t[key]
```

Implement object as a table of attributes

```
Account = {balance = 0}

Account[“withdraw”] = function(v) {
    Account[“balance”] = Account[“balance”] - v
}

Account[“withdraw”](100.00)
```

This works (!?)

- We don’t need to change anything!

Let’s improve the table-based object design

Method call on an object:

```
    Account[“withdraw”](100.00)
```

This works semantically but is syntactically ugly

Solution?

- Add new constructs through syntactic sugar
- We need to change the parser

The language design discussion

Question 1: What construct we add to the grammar of the *surface language*?

let’s say we want obj.field

```
E ::= E.E   ??
    |  E.ID  ??
    |  ID.E  ??
    |  ID.ID ??
```

Question 2: How do we rewrite (desugar) this construct to the base language?

Get vs. put

Reading an object field:

```
p.f               →     p[“f”]   →        get(p, “f”)
surface language        base language     bytecode
```

We need to distinguish between reading p.f

```
    v = p.f →   get(p, "f")
```

and writing to p.f

```
    p.f = v  →   put(p, "f", v)
```

Defining object methods

We will desugar

```
function Account.withdraw (v) {
    Account.balance = Account.balance - v
}
```

into

```
Account.withdraw = function (v) {
    Account.balance = Account.balance - v
}
```

Try #3: Objects as tables

a more robust version

Object as a table of attributes, revisited

```
Account = {balance = 0}

function Account.withdraw (v) {
    Account.balance = Account.balance - v
}
Account.withdraw(100.00)
a = Account

-- this code will make the next expression fail
Account = nil

a.withdraw(100.00) -- ERROR!
```

Solution: introduce `self`

```
Account = {balance = 0}
-- self “parameterizes” the code
function Account.withdraw (self, v) {
   self.balance = self.balance - v
}

a1 = Account
Account = nil
a1.withdraw(a1, 100.00)

a2 = {balance=0, withdraw = Account.withdraw}
a2.withdraw(a2, 260.00)
```

Hiding `self`: the colon notation

```
-- method definition
function Account:withdraw (v) {
    self.balance = self.balance - v
}
a:withdraw(100.00)  -- method call
```

Which construct to add to the surface language to support method calls?

```
E ::= E:E
E ::= E:ID
E ::= E:ID(args)
```

Desugaring `E:ID(args)`

```
E:ID(args) →

E.ID(E, args) -- this doesn't work (why?)

-- we want instead:
tmp = E;
tmp.ID(tmp, args)
```

Discussion

What is the inefficiency of our current object design?

Each object carries its attributes and methods.

If these are the same across many objects, a lot of space is wasted.

We will eliminate this inefficiency next

Summary of desugaring for objects

Access to an attribute

```
e.x           → e["x"]           -- get
e.x = v       → e["x"] = v       -- put
```

Method definition and call

```
function e:f(params) body        -- def
        →
e.f = function (self,params) body

expr:f(args)                     -- call
        →
def t = expr; t.f(t,args)
```

Meta-Methods

Meta-methods and meta-tables

Meta-methods and meta-tables:

Lua constructs for meta-programing with tables

Meta-programming:

creating a new language construct w/out compiler hacking

Meta-tables will be used for shallow embedding

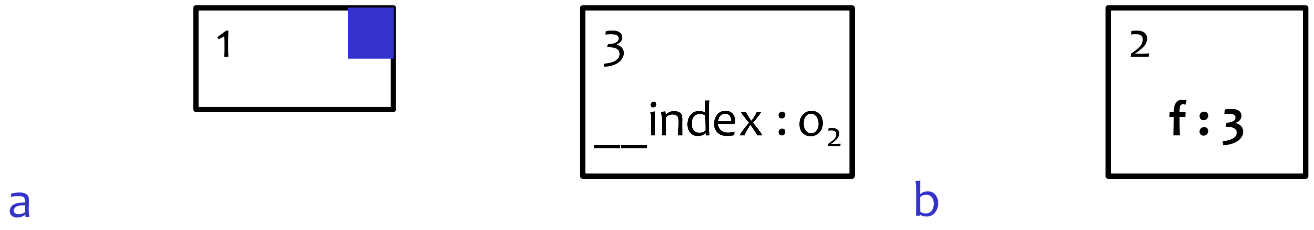
- ie, constructs created by writing library functions
- (sugar will be added to make them more convenient)
- recall: shallow embedding of a DSL = program in the DSL does not exist as a data structure

The __index attribute of the meta-table

Changes how errors are handled:

When a lookup of a field fails, the interpreter consults the __index field in the meta-table.

```
a = {}₁; b = {f=3}₂
print(a["f"]) --> nil
setmetatable(a, {__index = b}₃)
print(a["f"]) --> 3
```



Prototypes

poor man’s classes

Prototype

Prototype:

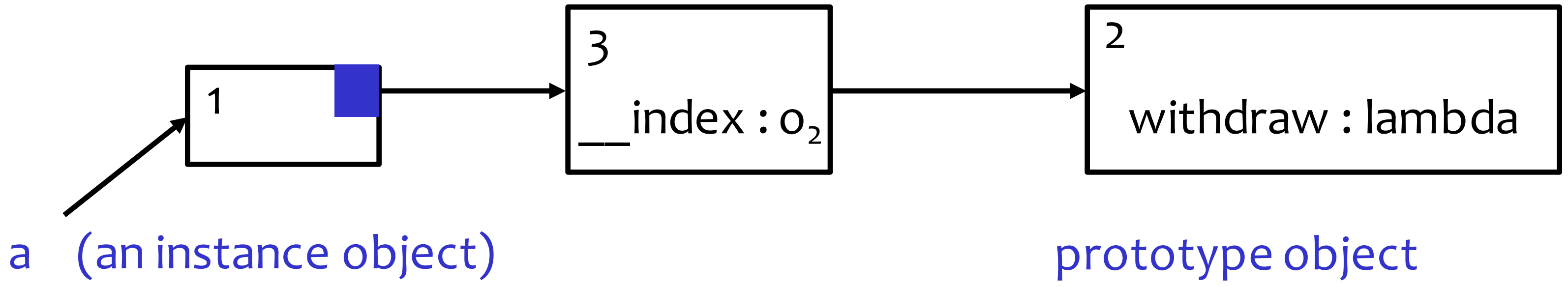
- a template for new objects
- the prototype is a regular object
  (as far as the interpreter can tell)
- you can almost think of it as a “class”

The prototype stores the common attributes

- objects refer to the prototype
- these objects will share properties with the prototype

Runtime setup of objects and prototypes

How are objects and prototypes linked?

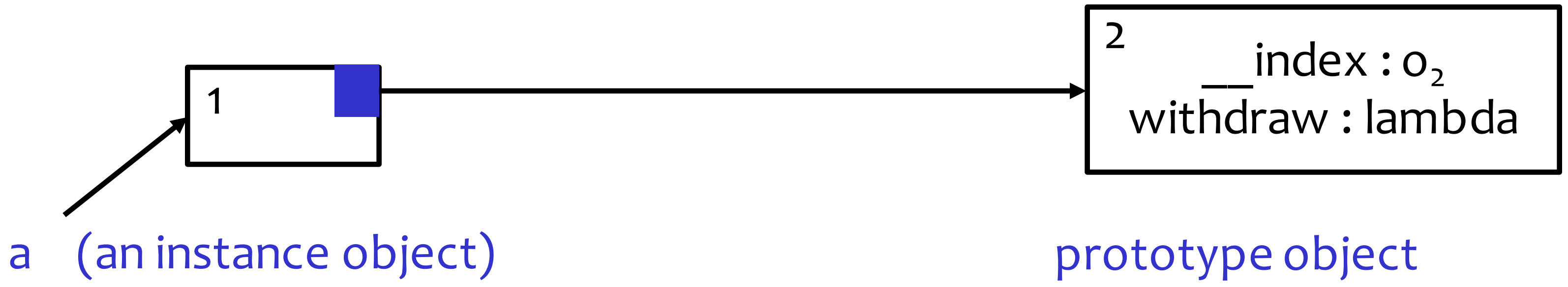


```
a:withdraw(100)
```

Can we avoid the extra meta-table?

Let’s use the prototype also as a meta-table.

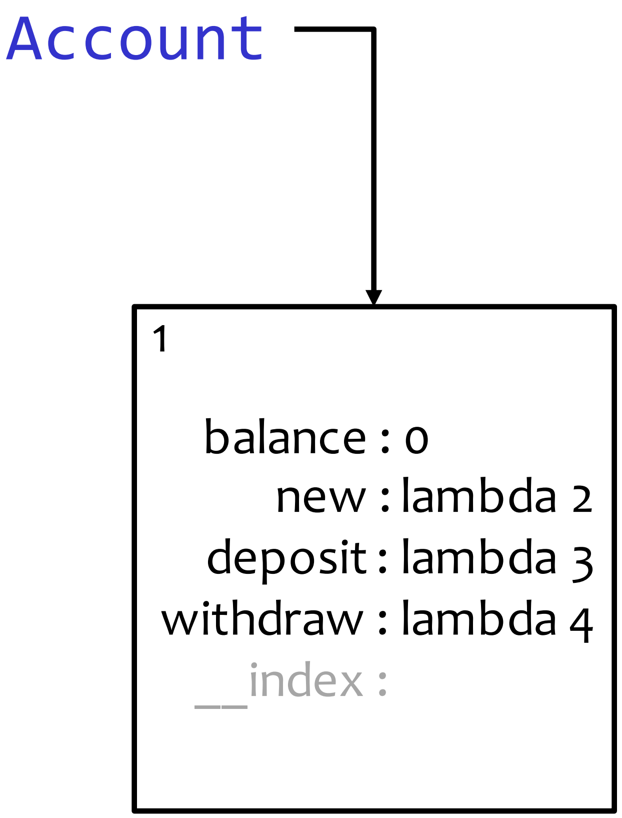
It saves space and memory allocation time.



```
a:withdraw(100)
```

Define the prototype and its methods

```
Account = {balance = 0}₁
function Account:new (o) {
    o = o or {}
    setmetatable(o, self)
    self.__index = self
    o
}₂
function Account:deposit (v) {
    self.balance = self.balance + v
}₃
function Account:withdraw (v) {
   if (v > self.balance) {
       error"insufficient funds"
   }
   self.balance = self.balance - v
}₄
```

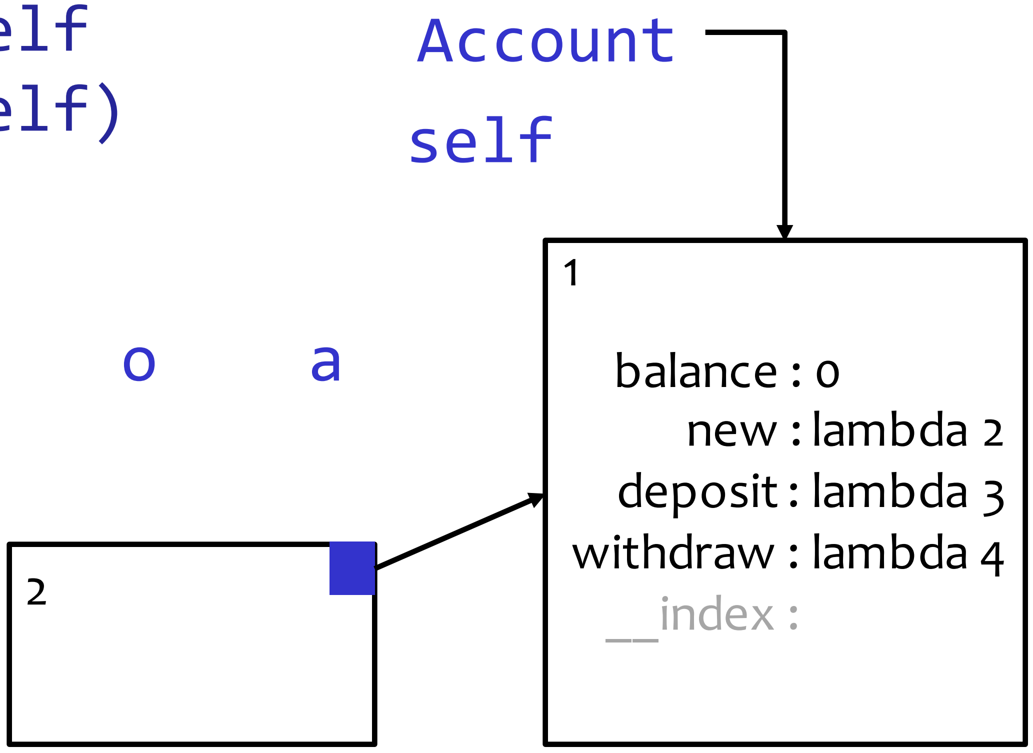


__index is added when new is called

Create an object

```
-- we repeat new() from previous slide
function Account:new (o) {
   -- create new object if not provided
    o = o or {}₂
    self.__index = self
    setmetatable(o,self)
    o
}
```

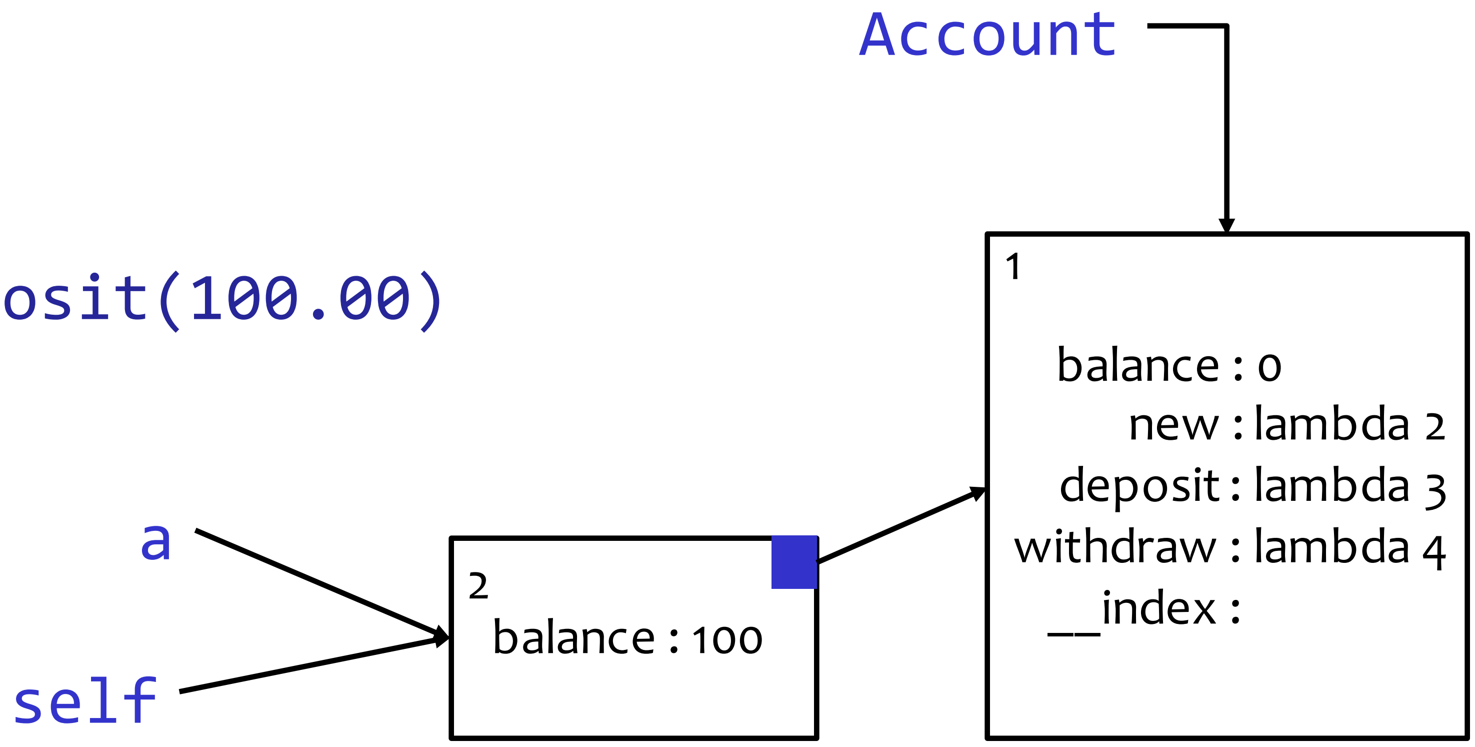
→ `a = Account:new()`



Call a method of an object

```
-- we repeat deposit() from a previous slide
function Account:deposit (v) {
    self.balance = self.balance + v
}
```

```
→ a:deposit(100.00)
```



balance is added during self.balance = …

Note about 401 assignments

We may decide not to use metatables, just the __index field. The code

```
function Account:new (o) {
    o = o or {}
    setmetatable(o,self)
    self.__index = self
    o
}
```

Would become

```
function Account:new (o) {
    o = o or {}
    o.__index = self
    o                       }
```

Which attrs will remain in the prototype?

After an object is created, it has attrs given in new()

```
a = Account:new({balance = 1000000})
```

What if we assign to the object later?

```
a.deposit = function_value?
```

Where will the attribute deposit be stored?

Discussion of prototype-based inheritance

Notice the sharing:

- constant-value object attributes (fields) remain stored in the prototype until they are assigned.

- After assignment, the object stores the attribute rather than finding it in the prototype

Assume field x resides in the prototype?.
What happens when you execute $\underbrace{a.x}_{\text{written to which object?}} = \underbrace{a.x}_{\text{read from which object?}} + 1$