Lecture 3

Growing the language

Scopes, binding, train wrecks, and syntactic sugar.

Ras Bodik
Shaon Barman
Thibaud Hottelier

Hack Your Language!

CS164: Introduction to Programming Languages and Compilers, Spring 2012

UC Berkeley

Administrativia

Wed 1pm discussion section has moved. See piazza.com.

- By the way, you are expected to read all piazza announcements.

In HW1, most wrote their first web mash-up. Congratulations!

Lessons:

- modern programs use multiple languages: HTML, CSS, JS, regex
- learning curve: languages and tools not so easy to learn
- in CS164, we'll learn skills to improve the situation

PA1 assigned today.

- Teams of two.
- Your repos on bitbucket.org. Submissions from bitbucket, too.
- We will require that you exchange files via bitbucket.

Today

Grow a language. Case studies on two languages.

The unit calculator: allow the user to

- add own units
- reuse expressions

Lambda interpreter: add control structures

- if, while, for, comprehensions
- using syntactic desugaring and lambdas

Part 1: Growing the calculator language

In L2, we implemented google constructs

Example:

`34 knots in mph` # speed of S.F. ferry boat

--> 39.126 mph

Example: # volume * (energy / volume) / power = time

half a dozen pints * (110 Calories per 12 fl oz) / 25 W in days

--> 1.704 days

Now we will change the language to be extensible

How we'll grow the language

1. Arithmetic expressions
2. Physical units for (SI only) code 44LOC
3. Add non-SI units code 56LOC
4. Explicit unit conversion code 78LOC

   this step also includes a simple parser: code 120LOC

5. Allowing users to add custom non-SI units

Growing language w/out interpreter changes

We want to design the language to be extensible

- Without changes to the base language
- And thus without changes to the interpreter

For calc, we want the user to add new units

- Assume the language knows about meters (feet, …)
- Users may wan to add, say, Angstrom and light year

How do we make the language extensible?

Our ideas

```
minute = 60 s
yard = 36 inch
```

existing units

units just introduced

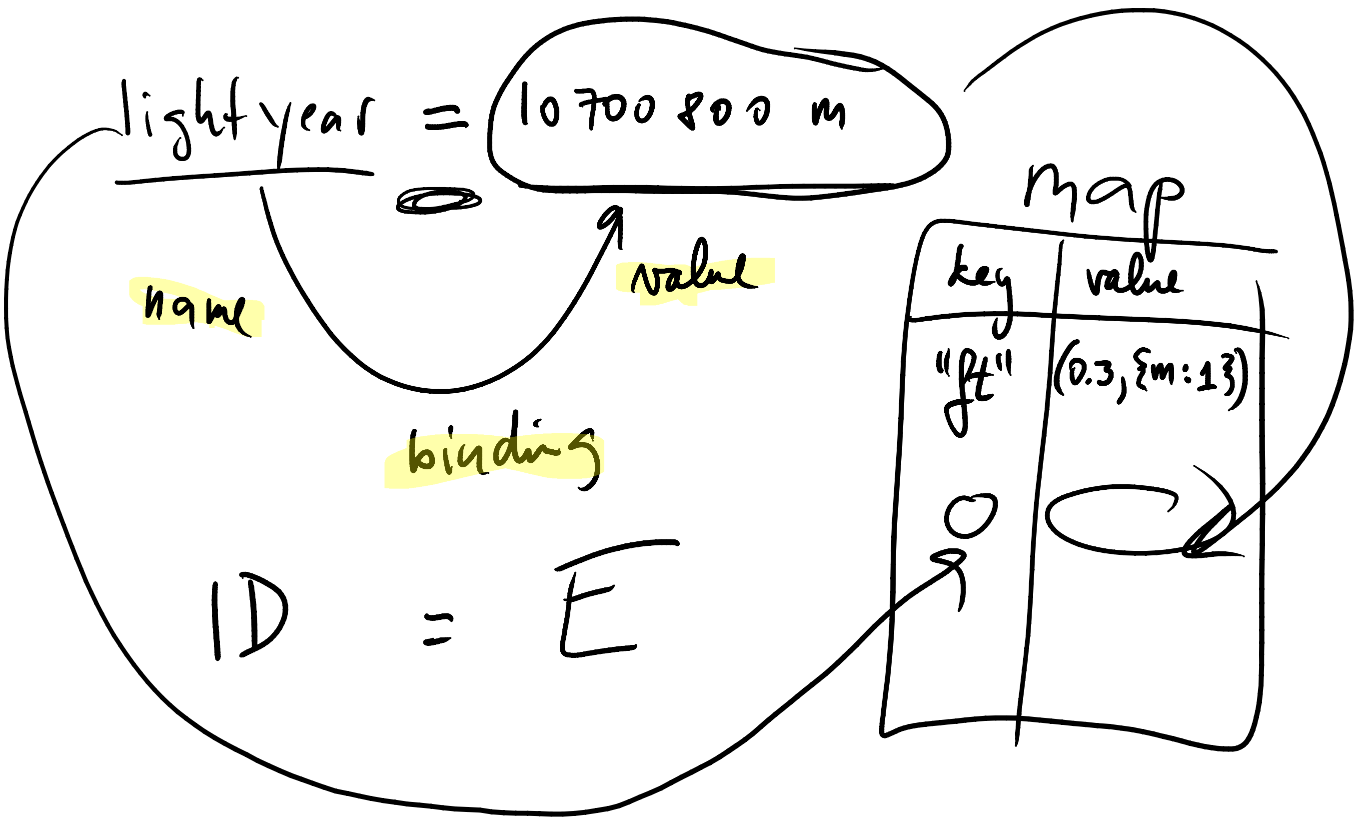
Bind a value to an identifier

```
minute = 60 s
hour = 60 minute
day = 24 hour
month = 30.5 day    // maybe not define month?
year = 365 day
km = 1000 m
inch = 0.0254 m
yard = 36 inch
acre = 4840 yard^2
hectare = (100 m)^2
2 acres in hectare  → 0.809371284 hectare
```

Implementing user units

Assume units extends existing measures.

We want the user to add **ft** when **m** or **yard** is known



How we'll grow the language

1. Arithmetic expressions
2. Physical units for (SI only) code 44LOC
3. Add non-SI units code 56LOC
4. Explicit unit conversion code 78LOC

   this step also includes a simple parser: code 120LOC

5. Allowing users to add custom non-SI units ✓
6. Allowing users to add custom measures

How do we add new measures?

No problem for Joule, as long you have kg, m, s:

$J = kg\ m^2 / s^2$

But other units must be defined from first principles:

Electric current:

- Ampere

Currency:

- USD, EUR, YEN, with BigMac as the SI unit

Coolness:

- DanGarcias, with Fonzie as the SI unit

Our ideas

Attempt 1:

when we evaluate `a = 10 b` and b is not known, add it as a new SI unit.

This may lead to spuriously SI units introduced due to typos.

Attempt 2:

ask the user to explicitly declare the new SI unit:

```
SI Ampere
```

Our solution

Add into language a construct introducing an SI unit

```
SI A                        // Ampere
mA = 0.0001 A
SI BigMac
USD = BigMac / 3.57         // BigMac = $3.57
GBP = BigMac / 2.29         // BigMac = £2.29
```

With “SI <id>”, language needs no built-in SI units

```
SI m
km = 1000 m
inch = 0.0254 m
yard = 36 inch
```

Implementing SI id

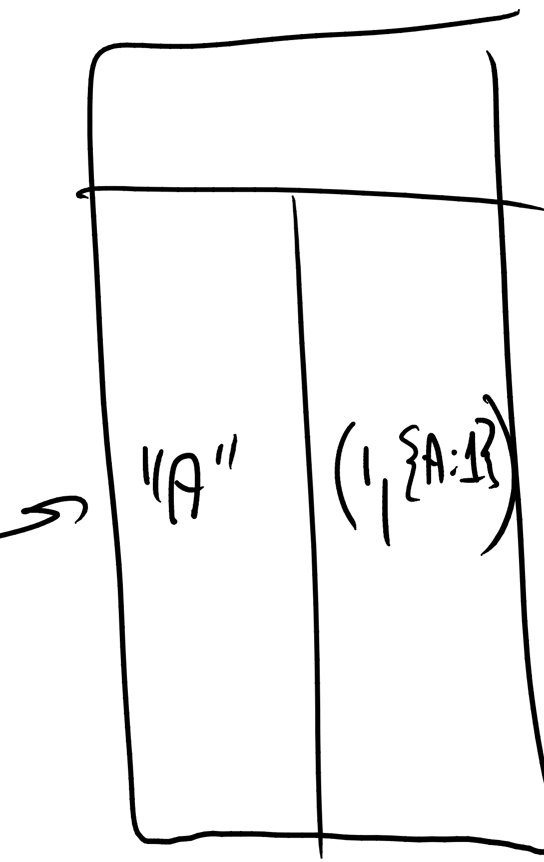
Problem

mA = A / 1000

Solve

declaration: SI ␣ A

mA = 0,001 A



How we'll grow the language

1. Arithmetic expressions
2. Physical units for (SI only) code 44LOC
3. Add non-SI units code 56LOC
4. Explicit unit conversion code 78LOC

   this step also includes a simple parser: code 120LOC

5. Allowing users to add custom non-SI units
6. Allowing users to add custom measures code √
7. Reuse of values

Motivating example

Compute # of PowerBars burnt on a 0.5 hour-long run

```
SI m, kg, s
lb = 0.454 kg;  N = kg m / s^2
J = N m;  cal = 4.184 J
powerbar = 250 cal
```

we wish to remember it as a constant

```
0.5hr * 170lb * (0.00379 m^2/s^3) in powerbar
  --> 0.50291 powerbar
```

Want to retype the formula after each morning run?

0.5 hr * 170 lb * (0.00379 m^2/s^3)

Reuse of values

To avoid typing

170 lb * (0.00379 m^2/s^3)

… we'll use same solution as for introducing units:

Just name the value with an identifier.

```
c = 170 lb * (0.00379 m^2/s^3)
28 min * c
# … next morning
1.1 hour * c
```

Should time given be in min or hours?

Either. Check this out! Calculator converts automatically!

How we'll grow the language

1. Arithmetic expressions
2. Physical units for (SI only) code 44LOC
3. Add non-SI units code 56LOC
4. Explicit unit conversion code 78LOC

   this step also includes a simple parser: code 120LOC

5. Allowing users to add custom non-SI units
6. Allowing users to add custom measures code
7. Reuse of values (no new code needed) √
8. Reuse of expressions (bind names to expressions)

Another motivating example

You want to print the current time left to deadline

now = 2011 year + 0 month + 18 day + 15 hour + 40 minute

--- pretend that now is always set to current time of day

Let's try to compute time to deadline

deadline = 2011 year + 1 month + 3 day   // 2/3/2012

timeLeft = deadline - now

timeLeft in day   --> *time left*

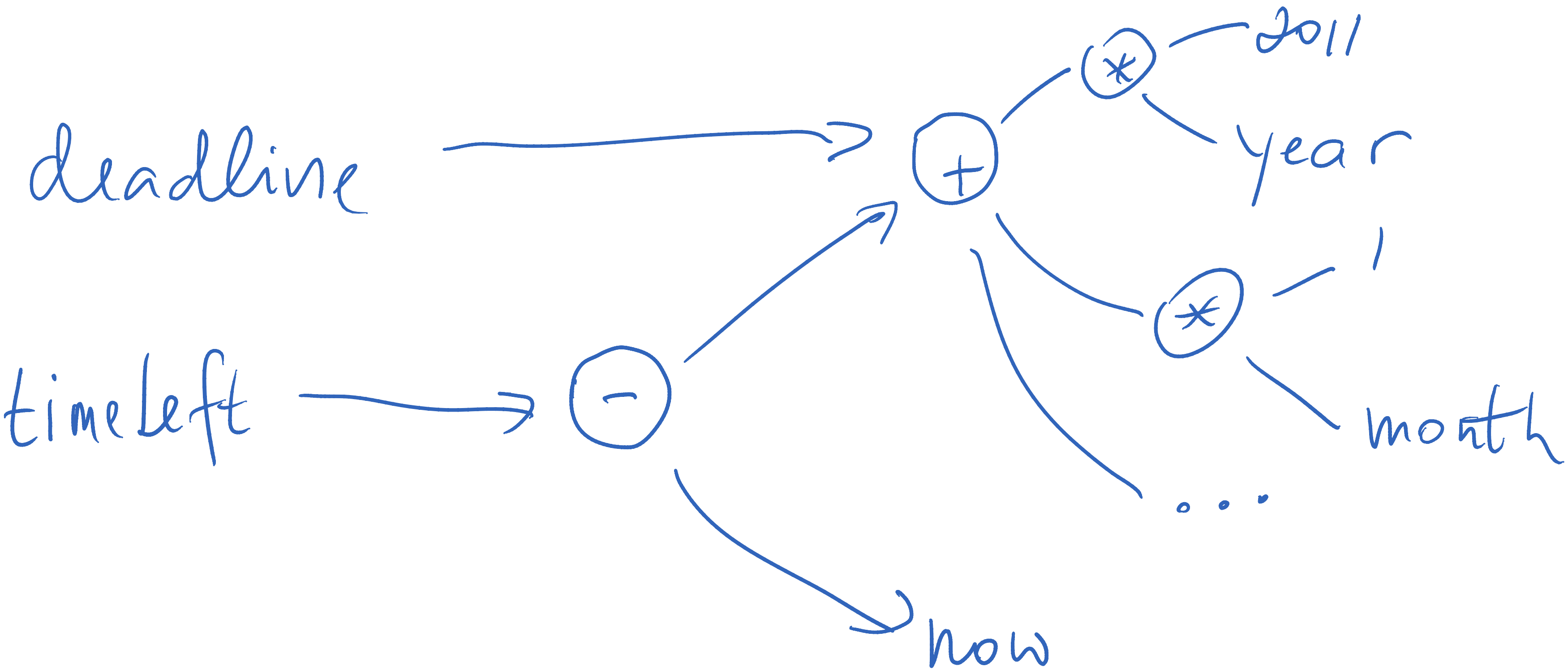
Wait for current time to advance. Print time left now.

What does the following print?

timeLeft in day   --> updated *time left*

How to achieve this behavior?

timeLeft is bound to an expression



year, month are actually expressions, too

Naming values vs. naming expressions

“Naming an expression” means that we evaluate it lazily when we need its value

How we'll grow the language

1. Arithmetic expressions
2. Physical units for (SI only) code 44LOC
3. Add non-SI units code 56LOC
4. Explicit unit conversion code 78LOC

   this step also includes a simple parser: code 120LOC

5. Allowing users to add custom non-SI units
6. Allowing users to add custom measures code
7. Reuse of values (no new code needed)
8. Reuse of expressions code (not fully lazy)

Summary: Calculator is an extensible language

Very little built-in knowledge

- Introduce base units with 'SI name'
- Arithmetic performs general unit types and conversion

No need to define all units in terms of SI units

```
cal = 4.184 J
```

Reuse of values by naming the values.

```
myConstant = 170 lb * (0.00379 m^2/s^3)
0.5 hr * myConstant in powerbar
```

-> Same mechanism as for introduction of non-SI units!

No need to remember units! Both will work fine!

```
0.5 hr * myConstant in powerbar
30 minutes * myConstant in powerbar
```

Limitations of calculator

No relational definitions

- We may want to define ft with ‘12 in = ft’
- We could do those with Prolog
  - recall the three colored stamps example in Lecture 1

Limited parser

- Google parses 1/2/m/s/2 as ((1/2) / (m/s)) / 2
- There are two kinds of / operators
- Their parsing gives the / operators intuitive precedence
- You will implement his parser in PA6

What you were supposed to learn

Binding names to values

– and how we use this to let the user grow the calculator

Introducing new SI units required declaration

- the alternative could lead to hard-to-diagnose errors

names can bind to expressions, not only to values

- these expressions are evaluated lazily

Part 2: Growing a functional language

Let’s move on to richer features

From calculations to “real programs”

We need more abstractions. Abstract code, data.

Abstractions are constructs that **abstract** away the implementation details that we don’t want to see.

We will build control abstractions today.

Focus on scoping, binding, syntactic sugar

Mostly review of CS61A, with historical lessons

Scoping and binding not easy to get right.

- mistakes prevent you from building modular programs

Today, we'll grow this stack of abstractions

comprehensions

for + iterators

if + while

lambda

Our language, now with functions

Let's switch to a familiar syntax and drop units.

Units can be easily added (they just make arithmetic richer)

```
S ::= S ; S
    |  E
    |  def ID ( ID,* ) { S }

E ::= n
    |  ID
    |  E + E
    |  (E)
    |  E ( E,* )
```

0 or more

foo()()()

Our plan

We have just enriched our language with functions.

Now we'll add (local) variables.

Simple enough? Wait to see the trouble we'll get into.

Names are bound to slots (locations)
Scopes are implemented with frames

First design issue: how introduce a variable?

Choice 1: explicit definition (eg Algol, JavaScript)

```
def f(x) {
  var a      # Define ‘a’. This is binding instance of a.
  a = x+1
  return a*a
}
```

Choice 2: implicit definition (Python) <-- *let’s opt for this*

```
def f(x) {
  var a      // under the covers this is what happens
  a = x+1       # existence of assignment a=... effectively
  return a*a    # inserts definition var a into function
}
```

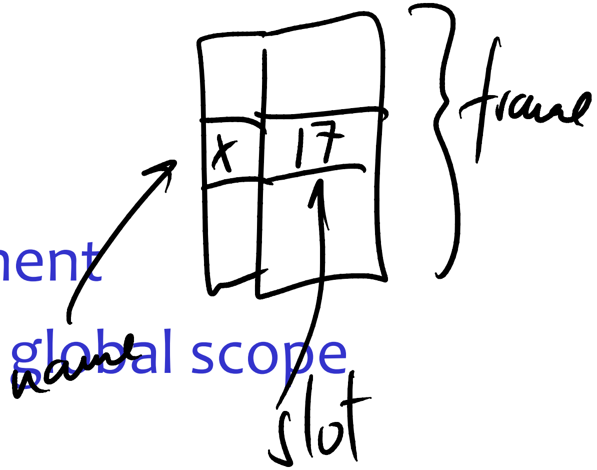
Implementation (outline)

When a function invoked:

1. create an new **frame** for the function
2. **scan** function body: if body contains 'x = E', then …
3. add a slot to the frame, bind name x to that slot

Read a variable:

1. look up the variable in the environment
2. check function scope first, then the global scope



We'll make this more precise shortly

What's horrible about this code?

```
def helper(x,y,date,time,debug,anotherFlag) {
  if (debug && anotherFlag > 2)
   doSomethingWith(x,y,date,time)
}
def main(args) {
  date = extractDate(args)
  time = extractTime(args)
  helper(12,13, date, time, true, 2.3)
  ...
  helper(10,14, date, time, true, 1.9)
  …
  helper(10,11, date, time, true, 2.3)
}
```

Your proposals

Allow nested function definition

```
def main(args) {
   date = extractDate(args)
   time = extractTime(args)
   debug = true
   def helper(x, y, anotherFlag) {
      if (debug && anotherFlag > 2)
         doSomethingWith(x,y,date,time)
   }
   helper(12, 13, 2.3)
   helper(10, 14, 1.9)
   helper(10, 11, 2.3)
}
```

A historical puzzle (Python version < 2.1)

An buggy program

```
def enclosing_function():
    def factorial(n):
        if n < 2:
            return 1
        return n * factorial(n - 1)
    print factorial(5)
```

A correct program

```
def factorial(n):
    if n < 2:
        return 1
    return n * factorial(n - 1)
print factorial(5)
```

video

Explanation (from PEP-3104)

- Before version 2.1, Python's treatment of scopes resembled that of standard C: within a file there were only two levels of scope, global and local. In C, this is a *natural consequence of the fact that function definitions cannot be nested*. But in Python, though functions are usually defined at the top level, a function definition can be executed anywhere. This **gave Python the syntactic appearance of nested scoping without the semantics**, and yielded inconsistencies that were surprising to some programmers.

This **violates the intuition** that a function should behave consistently when placed in different contexts.

Scopes

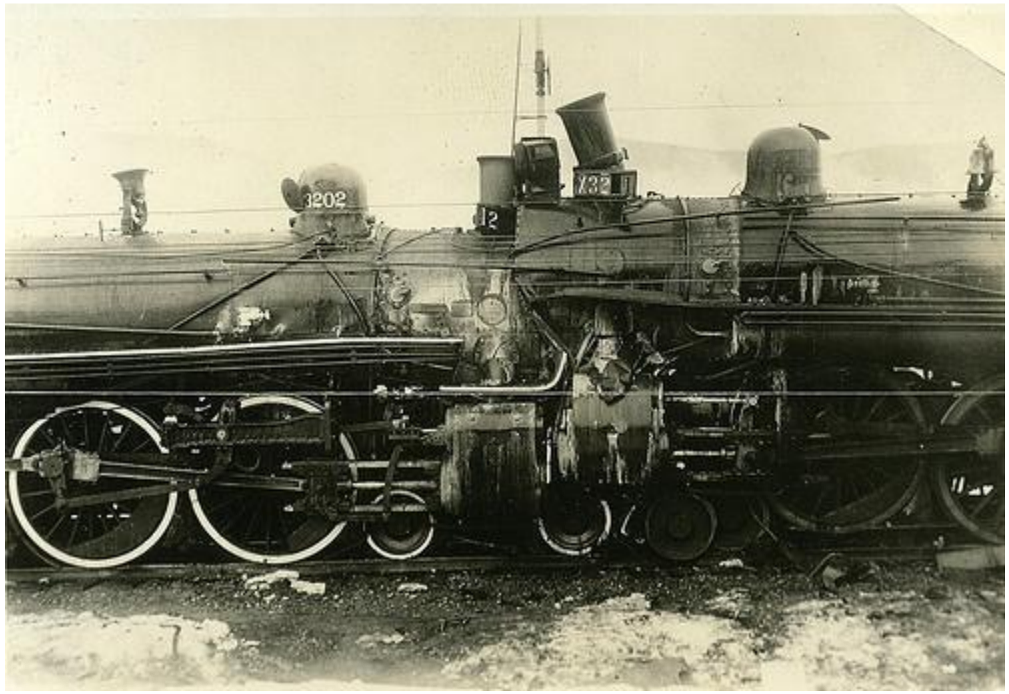
Scope: defines where you can use a name

```
def enclosing_function():

    def factorial(n):

        if n < 2:
            return 1
        return n * factorial(n - 1)

    print factorial(5)
```


3202

Summary

Interaction of two language features:

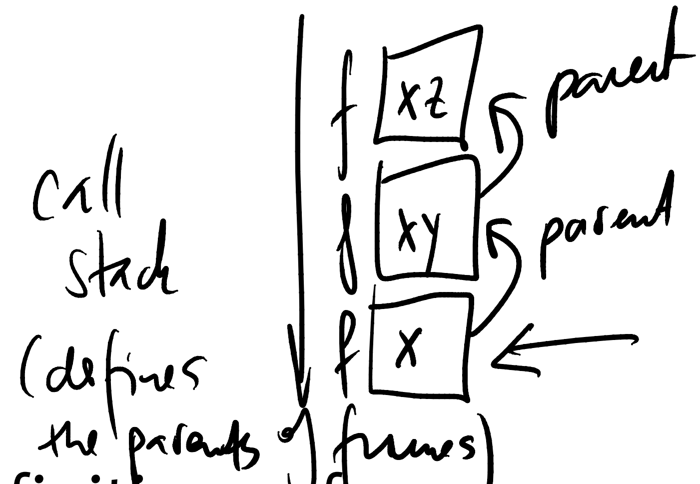
- Scoping rules
- Nested functions

Features must often be considered in concert

A robust rule for looking up name bindings

Assumptions:

1. We have nested scopes.



2. We may have multiple definitions of same name.

   new definition may hide other definitions

3. We have recursion.

   may introduce unbounded number of definitions, scopes

Example

Program

Environment

Rules

At function call:

At return:

When a name is bound:

When a name is referenced:

Control structures

Defining control structures

They change the flow of the program

- if (E) S else S
- while (E) S
- while (E) S finally E

There are many more control structures

- exceptions
- coroutines
- continuations

Assume we are given a built-in conditional

Meaning of ite(v1,v2,v3)

if v1 == true then evaluate to v2,

else evaluate to v3

Can we use it to implement if, while, etc?

```
def fact(n) {
  ite(n<1, 1, n*fact(n-1))
}
```

never terminates

call-by-value

Ifelse

Can we implement ifelse with just functions?

```
def ifelse (c ,t ,f ) {   # in terms of ite
      f = ite(c, t, f); f()
}
def fact(n) {
      ifelse( n<2,
              lambda() { 1 },
              lambda() { n * fact(n-1) } )
}
```

scratch space

Correct **If** : does not evaluate both branches

```
def fact(n) {
  def true_branch() { 1 }
  def false_branch() { n * fact(n-1) }
  ifelse (n<2, true_branch, false_branch)
}

def ifelse (e, th, el) {
  x = ite(e, th, el)
  x()
}
```

Anonymous functions

```
def fact(n) {
    if (n<2, function() { 1 }
           , function() { n*fact(n-1) } )
}
```

If

```
def if(e,th) {
    cond(e,th, lambda(){} )()
}
```

Aside: desugar function definitions

Our language consists of assignments

```
x = expression
```

and function definitions

```
def fact(n) { body }
```

Can we reduce these two features into one? Yes.

```
fact = function(n) { body }
```

Named functions are just variables w/ function-values.

Test yourself. Have these two the same effect?

```
fact(4)                    x=fact; x()
```

While

Can we develop **while** using first-class functions?

While

```
count = 5
fact = 1
while( lambda() { count > 0 },
       lambda() {
             count = count - 1
             fact := fact * count }
)
while (e, body) {
    x = e()
    if (x, body)
    if (x, while(e, body))
}
```

tail call elimination

If, while

With closures, we can define If and While.

- These are high-order functions (i.e., their args are closures).

We first need to extend the base language (ie the interpreter) with ite(e1,e2,e3)

- evaluates all arguments
- evaluates to e2 when e1 is true, and e3 otherwise.

Now, we can define If and While

```
def If (c, B) { ite(c,B,lambda(){})() }
def While(C,B) { def t = C(); If(t,B); While(C,B) }
```

If, while

We can now write a while loop as follows:

```
while(lambda() { x < 10 } ,
        lambda() {
               loopBody
        })
```

this seems ugly, but the popular jQuery does it, too

```
$(".-123").hover(
                function(){ $(".-123").css("color", "red"); },
                function(){ $(".-123").css("color", "black"); }
           );
```

Also see

Guy Lewis Steele, Jr.:

"Lambda: The Ultimate GOTO" pdf

Smalltalk/Ruby actually use this model

Control structure not part of the language

Made acceptable by special syntax for blocks

- which are (almost) anonymous functions

Smalltalk:

```
| count factorial |
count := 5.
factorial := 1.
[ count > 0 ] whileTrue:
    [ factorial := factorial * (count := count - 1) ]
Transcript show: factorial
```

while

Almost the same in Ruby

```ruby
count = 5
fact = 1
while count > 0 do
    count = count - 1
    fact = fact * 1
end
```

not a lambda

"block" ≈ lambda

Syntactic sugar

We can provide a more readable syntax

```
while (E) { S }
```

and desugar this ‘surface’ construct to

```
While(lambda() { E }, lambda() { S })
```

Two ways to desugar

AST rewriting (sits between parsing and interpreter)

while (*E*) {*S*} → parser → AST with *While* node
→ rewriter → AST w/out *While* node

In the parser (during “syntax-directed translation”)

while (**E**) {**S** } → parser → AST w/out *While* node

```
S ::= 'while' '(' E ')' '{' S_list '}'
  %{ return ('exp', ('call', ('var', 'While'),
                     [('lambda',[], [('exp',n3.val)]),
                      ('lambda',[], n6.val)]))          %}
```

AST desugaring algorithms

An example rewrite rule

Traverse the three bottom-up or top-down?

Is one tree traversal sufficient?

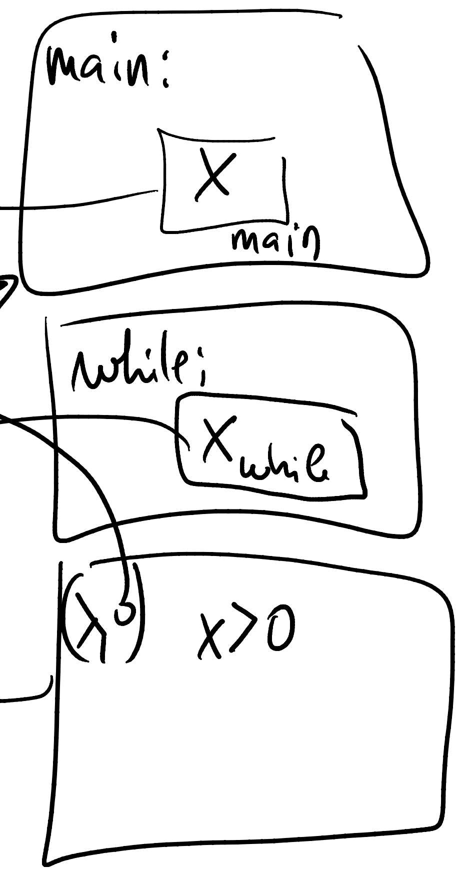
Now let's put our language to a test

```
count = 5
fact = 1
while( lambda() { count > 0 },
       lambda() {
              count = count - 1
              fact := fact * count }
)
```

Now put this to a test

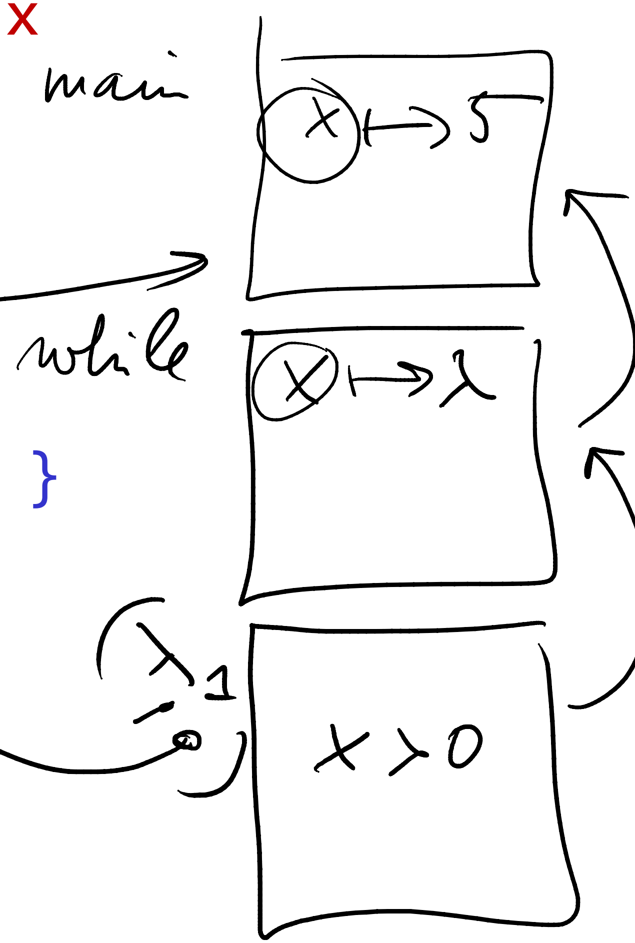
replace count with x

```
x = 5
fact = 1
while( lambda() { x > 0 },
       lambda() {
              x = x - 1
              fact := fact * count }
)
while (e, body) {
    x = e()
    if (x, body)
    if (x, while(e, body))
}
```



Now put this to a test

```
x = 5          replace count with x
fact = 1
while( lambda()_1{ x > 0 },
       lambda()_2{
             x = x - 1
             fact := fact * x }
)
while (e, body) {
    x = e()
    if (x, body)
    if (x, while(e, body))
}
```




DE L'OUEST

Dynamic Scoping

Program

Environment

Rules

At function call:

At return:

When a name is bound:

When a name is referenced:

Our rule (dynamic scoping) is flawed

Dynamic scoping:

find the binding of a name in the execution environment

that is, in the stack of scopes that corresponds to call stack

binds 'x' in loop body to the unrelated 'x' in the while(e,b)

Dynamic scoping is non-compositional:

variables in while(e,b) not hidden

hence hard to write reliable modular code

Find the right rule for rule binding

```
x = 5
fact = 1
while( lambda() { x > 0 },
       lambda() {
              x = x - 1
              fact := fact * count }
)
while (e, body) {
  x = e()
  if (x, while(e, body), function(){} )
}
```

scratch space

Closures

Closure: a pair (function, environment)

this is our new "function value representation"

function:

- it's first-class function, ie a value, ie we can pass it around
- may have free variables

environment:

- it's the environment when the function was created
- when function invoked, will be used to bind its free vars

This is called **static (or lexical) scoping**

Application of closures

From the Lua book

```
names = { "Peter", "Paul", "Mary" }
grades = { Mary: 10, Paul: 7, Paul: 8 }
sort(names, function(n1,n2) {
      grades[n1] > grades[n2]
}
```

Another cool closure

```
c = derivative(sin, 0.001)
print(cos(10), c(10))
   --> -0.83907, -0.83907

def derivative(f,delta)
    function(x) {
        (f(x+delta) - f(x))/delta
    }
}
```

This code will actually break in our language

Where is the problem?

How to fix it?

proper lexical scoping

At function call:

At return:

When a name is bound:

When a name is referenced:

Another cool one, again in Lua:

```
function foo() {
  local i = 0
    return function ()
        i = i + 1
        return i
    end
end
c1 = foo()
c2 = foo()
print(c1())
print(c2())
print(c1())
```

In our language

```
def newCounter() {
    i = 0
    function ()
        i = i + 1
    end
end
c = newCounter()
print(c())
print(c())
```

In Python

```
def foo():
    a = 1
    def bar():
        a = a + 1   <-- Local variable 'a'
        return a    referenced before assignment
    return bar
c = foo()
print(c())
print(c())
```

Same in JS (works just fine)

```
function foo() {
    var a = 1
    function bar() {
        a = a + 1
        return a
    }
    return bar
}
f = foo()
console.log(f())            --> 2
console.log(f())            --> 3
```

g = foo()

Attempt to fix the semantics

```
def foo():
    a = 1
    def bar():
        a = a + 1
        return a
    return bar
```

Current rule: If a name binding operation occurs anywhere within a code block, all uses of the name within the block are treated as references to the current block['s binding].

Fix in Python 3, a new version of language

```
def foo():
    a = 1
    def bar():
        nonlocal a
        a = a + 1
        return a
    return bar
f = foo()
```

Python iterators

What does this python code output?

```
a = []
for i in xrange(0,10):
    a.append(lambda: i)
for j,v in enumerate(a):
    print j,v()
for i,v in enumerate(a):
    print i,v()
```

Broken lambda?

Recall our language

```
E ::= n
    | ID
    | E op E
    | (E)
    | lambda(ID, … , ID) { S }
    | E(E, … , E)
    | def ID = E
    | ID = E
S ::= E
    | S ; S
```

HW2 hint

Your rewrite (desugaring) of

```
for id in E:
    body
```

should not modify the body:

If you are the compiler, you want to translate **for** without regard for what's in the body. Otherwise there will be many special cases. To have a simple, modular compiler, you translate body separately.

Tables

First let's add tables

Tables are arrays and dicts in one

```
def salary = {}
salary["John"] = 123
salary[0] = 7          // to get an array, use numeric index
print salary[0]
```

Support for tables

What operations do we need to add to the language?

{}	*table:* evaluates to a new, empty table
E1[E2]	*get:* evaluate E1 and E2, then evaluates to the value of key E2 in table E1
E1[E2] = E3	*put:* stores value of E2 in the table E1 under key given by the value of E2
E2 in E1	*membership:* is key E2 in table E1

Tables

What semantic issues we need to decide?

What values are allowed as keys?

- for efficiency, you may disallow some data types

What's the result of E[E] when key is not in table?

- in particular, what to do in a language without exceptions?

Evaluation order of E1, E2, E3 in E1[E2]=E3

- in what language the order does not matter?

Implementation of tables

Can they be implemented as sugar on our language?

- that is, do we need to extend the interpreter?
- or does it suffice to add some library functions?

For loops and iterators

For loops

To support libraries, and modularity in general we allow iterators over data structures.

```
   for v in iteratorFactoryExp { S }
->
  $1 = ireratorFactoryExp
  def v = $1()
  while (v != null) {
      v = $1()
      S
  }
```

A counting iterator Factory

From PA2, more or less:

```
def iter(n) {
     def i = 0
     lambda () {
          if (i < n) { i = i + 1; i }
          else { null }
     }
}
for (x in iter(10)) { print x }
```

iterator factory for tables

This one assumes that we are using the table as array:

```
def asArray(tbl) {
     def i = 0
     lambda () {



     }
}
def t = {}; t[0] = 1; t[1] = 2
for (x in asArray(t)) { print x }
```

Comprehensions

Comprehensions

A *map* operation over anything itererable. Example

```
[toUpperCase(v) for v in elements(list)]
```

--->

```
$1 = []
for v in elements(list) { append($1, toUpperCase(v)) }
$1
```

In general:

[***E*** for ***ID*** in ***E***]

Nested comprehensions

Does our desugaring work on nested comprehensions?

```
mat = [[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9],
      ]
print [[row[i] for row in mat]
               for i in [0, 1, 2]
      ]
--> [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

"To avoid apprehension when nesting list comprehensions, read from right to left"

Our abstraction stack is growing nicely

comprehensions

for + iterators

if + while

lambda