Growing a general purpose language

Functions, scopes and famous train wrecks.

CS164: Introduction to Programming Languages and Compilers

Fall 2009

Instructor: **Ras Bodik**

GSI: **Joel Galenson**

Courseware: **Tim Trutna**

UC Berkeley

Administrativia

Sign up your Project Teams.

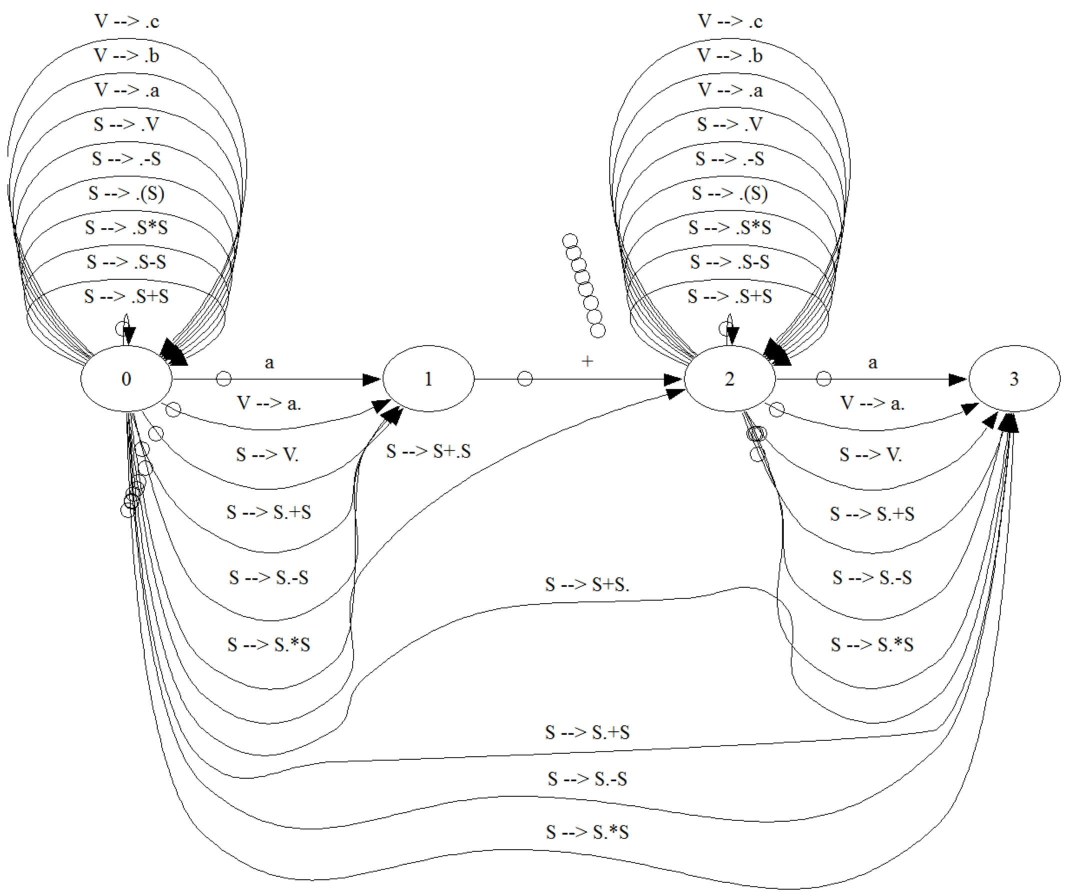
Milestone of Project 1 due on Monday!

- Set up your repository.
- Understand the provided Earley parser code. Add visualization.
- Understand the provided front-end parser.
- Modify the provided Earley code to use the grammar AST generated by the front-end parser.
- Add a lexer.
- Test the resulting recognizer.

Turn off your cell phones and close laptops.
Or face difficult questions.

A visualization of Earley parse

source code for this graph has been posted in the Project 2 document



Remember life before parsing …

Unit-crunching Super-calculator: key plot turns

SI m, kg, s

N = kg m / s^2

J = N m

cal = 4.184 J

powerbar = 250 cal

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

--> 0.50291 powerbar

Take cs164. Become unoffshorable.

"We design them here, but the labor is cheaper in Hell."

Growing a general-purpose language

A challenge problem we ran into

Do you want to retype the formula after each run?

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

Our solution

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

28 min * c

1.1 hour * c

Good: should time be in minutes or hours?

No need to remember. Calculator converts automatically!

Bad: the real formula depends on speed. Approx:

30 min * 170 lb * (**6 mph^2 * const** m^2/s^3)

→ We need a better way to reuse our code

Reuse code (avoid retyping, debugging, etc)

Previously, we remembered the value of c

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

This fails when we need to reuse this calculation:

30 min * 170 lb * (**(3 mile / 30 min)^2 * const** m^2/s^3)

Reusing an expression

Parameterize it!

time * weight * ((distance / time)^2 * const m^2/s^3)

And give it a name!

def nrg: time * weight * ((distance /time)^2 * const m^2/s^3)

It is now reusable – if we can instantiate it with values.

time = 30 min; distance = 3 miles; weight = 170lb;

nrg()

What have we deisgned:

The named expression has free variables.

Free variables are bound when the expression is evaluated.

They are bound to definitions in the evaluation environment.

Better

We reused the expression but did not hide its details.

the names of free variables remained visible

A fix?

```
def nrg(time, weight,distance):
   time * weight * ((distance /time)^2 * const m^2/s^3)
```

Call args set the values of formal function parameters

nrg(30 min, 170lb, 3 miles)

nrg is a function with no free variables.

it is an abstraction (hides the implementation)

nrg's body does have free variables

these are bound to parameters (which are definitions)

Our calculator language with functions

S ::= S ; S | E | E **in** C | ID **=** E | **SI** ID | **def** ID (IDlist) : E

C ::= U | C / C | C * C | C C | C^n

E ::= n | ID | E op E | (E) | f{ Elist } | f{}

Elist ::= E | Elist , E

Idlist ::= [similar]

op ::= + | - | '*' | ε | /

f*(g)

Let's simplify it for further development

Drop unit. Use the more usual syntax.

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

E ::= n | ID | E op E | (E) | f(Elist) | f()

We omit the obvious when this causes no confusion.

Elist ::= E | Elist , E

op ::= + | - | * | /

We dropped ε for multiplication.

Notice absence of variable definition

How do we introduce a local variable?

```
def f(x, y) {
    z = 1
      = z
    local w = 2
}
```

(1) z = 1 — defines a new local var

= z — not an introduction of z

(2) local w — explicit DECLARATION

Two alternatives

Explicit definition (eg Algol, JavaScript)

```
def f(x) {
  var a
  a =  x+1
  return a*a
}
```

OUR CHOICE

Second choice (Python)

```
def f(x) {
  global a
  a =  x+1
  return a*a
}
```

Implementation (outline)

When a function invoked:

1. create an new scope for the function
2. scan the body: if function body contains 'x = E', then …
3. bind x: add x to the scope of the function

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)
}
```

bindings

date = "..."

date = "..."

date, time are nonlocals

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)
```

need to bind this name

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

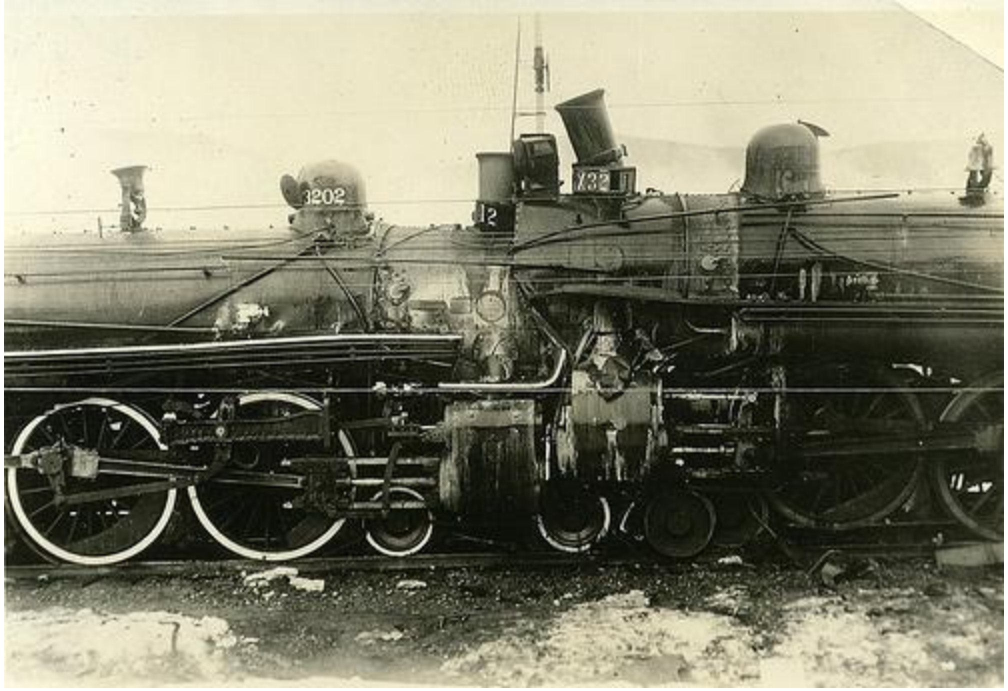
Scope: defines where you can use a name

```
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.

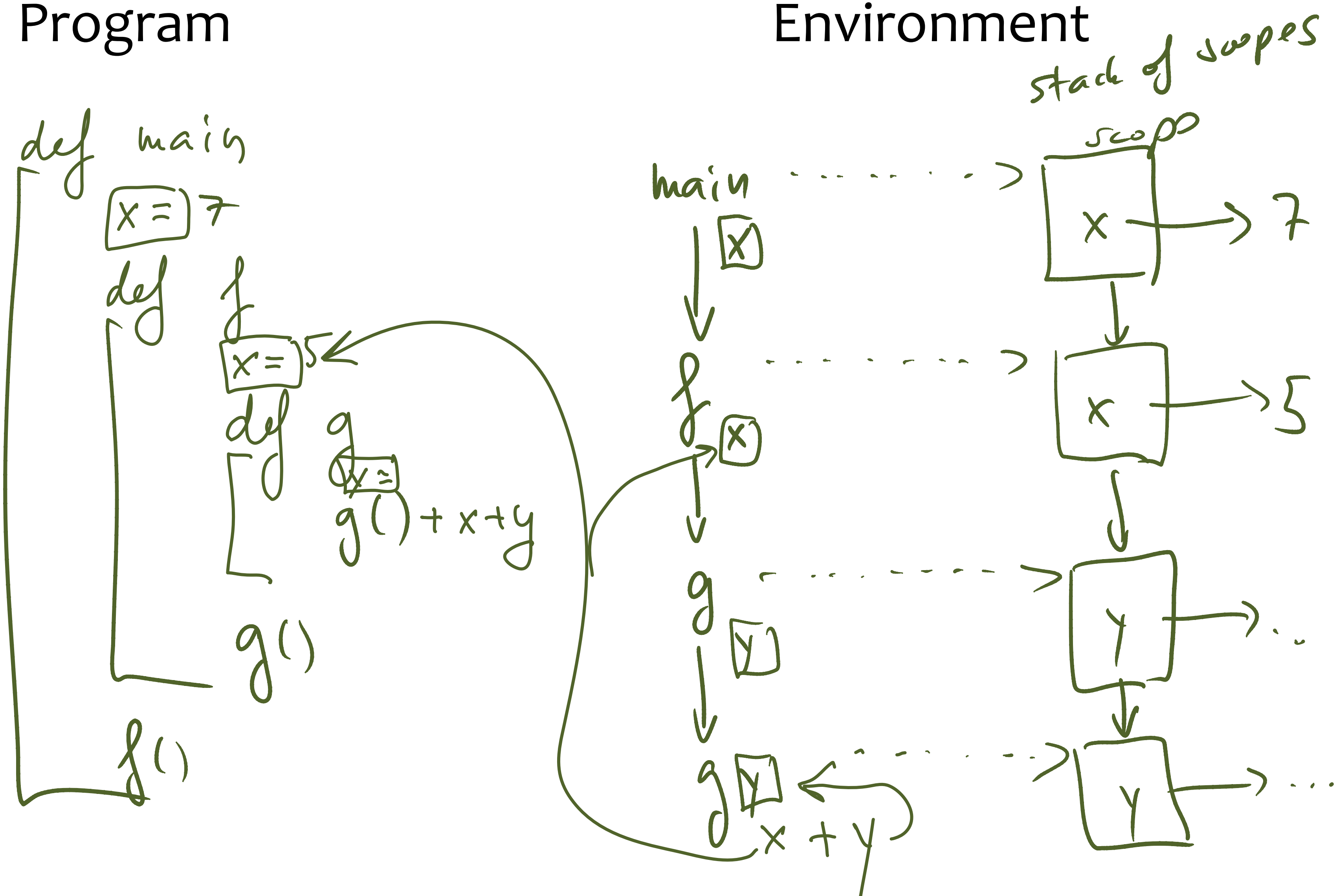
2. We may have multiple definitions of same name.
   new definition may hide other definitions

3. We have recursion.
   may introduce unbounded number of definitions, scopes

Example

Program

Environment



Rules

At function call: create a scope
push it

At return: pop a scope

When a name is bound: at x =
add it to the scope

When a name is referenced:

walk scopes down the stack, looking for the name

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 cond(v1,v2,v3)   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) {
  cond(n<1, 1, n*fact(n-1))
}
```

fact(n-2)

fact(n-3)

...

Ifelse

Can we implement ifelse with just functions?

```
def ifelse (c, th, el) {    # in terms of cond
        x = cond(c, th, el)
        x()
}
```

scratch space

If that does not evaluate both branches

```
def fact(n) {
  ret = 0
  def true_branch() { ret = 1 }
  def false_branch() { ret = n * fact(n-1) }
  if else (n<2, true_branch, false_branch)
  ret
}
def ifelse (e, th, el) {
  x = cond(e, th, el)
  x()
}
```

Anonymous functions

```
def fact(n) {
  ret = 0
  if (n<2, function() { ret = 1 }
         , function() { ret = n*fact(n-1) }
  )
  ret
}
```

If

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

Aside: first-class functions and function defs

Anonymous functions clarify function definitions

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

can be expressed as syntactic sugar over assignments to variables

fact

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

First-class functions are just values stored in variables.

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))
}
```

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
```

whileTrue(B1, B2)

B1

Same in Ruby

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

not a block

block / anonymous functions

Also see

Guy Lewis Steele, Jr.:

"Lambda: The Ultimate GOTO" pdf

Now put this to a test

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

Now put this to a test

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


L'OUEST

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 the body of while loop to x in the while loop

Thus is non-compositional:

variables in while 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:

a first-class function (it's a value, we can pass it around)

with free variables

environment:

at the time when function is created

used to bind free variables in function

This is called static (or lexical) scoping

Cool 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 one

```
def derivative(f)
    delta = 0.0001
    function(x) {
        (f(x+delta) – f(x))/delta
    }
}

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

And another one, in Lua:

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

In our language

```
def newCounter() {
  i = 0
  function ()
    i = i + 1
    i
  end
end
c1 = newCounter()
c2 = newCounter()
print(c1())
print(c2())
print(c1())
```

In Python

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

local variable 'a' referenced before assignment

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
```

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()
```

LESSONS

1)

2)

3)