‡2: Functions, Expressions, Environments

Public-Service Announcement

⁄ Consulting is a student-run consulting group on cam- a group of 30 students that complete 4 projects a ‹ Fortune 500 firms, startups, and nonprofit organi- solve problems for and provide solutions to compa- industries like Google, Dropbox and Khan Academy. ently recruiting and would love to have you join us! ng for students from all majors who are driven, crit- , team players, and able to think outside the box. If ested in joining up please visit `bc.berkeley.edu` for ation. Also make sure to come to one of our info ses- ary 24th and 26th to learn more and attend our case January 27th to prepare for the interview process. see you at one of our events next week!"

From Last Time

cture: *Values* are data we want to manipulate and in

e values that perform computations on values.

denote computations that produce values.

ook at them in some detail at how functions operate on nd how expressions denote these operations.

hough our concrete examples all involve Python, the ac- s apply almost universally to programming languages.

Functions

ure, we're going to use this notation to show function n are created by evaluating function *definitions*):

s(number):

add(left, right)

fy this in a bit to make it easier to write.)

arenthesized lists indicate the number of *parameter uts* the functions operate on (this information is also unction's *signature*).

oses, the blue name is simply a helpful comment to sug- he function does, and the specific (green) parameter kewise just helpful hints.

ally maintains this *intrinsic name* and the parameter ally, but this is not a universal feature of programming

Pure Functions

ental operation on function values is to *call* or *invoke* means giving them one value for each formal parameter hem produce the result of their computation on these

-5 ▷ *abs*(number): ▷ 5

29, 13) ▷ *add*(left, right) ▷ 42

unctions are *pure:* their output depends only on their ters' values, and they do nothing in response to a call a value.

Impure Functions

ay do additional things when called besides returning a

things *side effects.*

e built-in print function:

-5 ▷ print(• • •) ▷ None

display text '-5'

ext is print's side effect. Its value, in fact, is generally ys the null value).

Other Kinds of Impurity

ffects involve changing the value of some variable.

e function random.randint:

```
randint(0, 100)   # Random number in 0--100.

randint(0, 100)
                  # Something must have changed!
```


Call Expressions

ssion denotes the operation of calling a function.

l(2, 3):

$\underbrace{\text{add}}_{\textit{Operator}}(\underbrace{2}_{\textit{Operand 0}}, \underbrace{3}_{\textit{Operand 1}})$

r and the operands are all themselves expressions (re-
).

this call expression:

he operator (let's call the value C);

the operands in the order they appear (let's call the
and P_1)

ich must be a function) with parameters P_0 and P_1.

h the definitions for base cases (mostly literal expres-
nbolic names), this describes how to evaluate any call.

xample: From Expression to Value

e expression mul(add(2, mul(0x4, 0x6)), add(0x3, 005)).
sequence, values are shown in boxes.
ide a box is an expression.

ul(0x4, 0x6)), add(0x3, 005))

(add(2, mul(0x4, 0x6)), add(0x3, 005))

(add(left, right) (2 , mul(left, right) (4 , 6)),
add(0x3, 005))

(add(left, right) (2 , 24), add(0x3, 005))

(26 , add(0x3, 005))

(26 , add(left, right) (3 , 5))

(26 , 8)

Example: Print

xpression with side effects?

), print(2))

(print(• • •) (1), print(2))

(None , print(2))

'1'.

(None , print(• • •) (2))

(None , None))

'2'.

'None None'.

Names

xpressions that are literals is easy: the literal's text
information needed.

I evaluate names like add, mul, or print?

here must be another source of information.

y a simple approach: *substitution* of values for names.

ver all the cases, however, and so we'll introduce the
n *environment*.

Substitution

xplain the effect of

ach assignment (=) as a *definition*.

`x = 3`	`x = 3`	`x = 3`
`y = 3 * 2`	`y = 6`	`y = 6`
`z = y ** 3`	`z = 6 ** 3`	`z = 216`

eplace names by their definitions (values).

Substitution and Functions

r a simple function definition:

```
:(x, y):
 (x * y) ** x
te(3, 2))
```

ment is sort of like an assignment, but specialized to
lues.

ement above defines compute to be "the function of x
turns $(xy)^x$."

a common notation for that (due to Church):

$y)^x$.

stitution for compute, we have

`(λ x,y: (xy)^z) (3, 2) )`

bstitution and Formal Parameters

ll such as

`: (xy)^z) (3, 2)`

de is like a *simultaneous assignment* to or substitution

e the whole expression with

lly), just 216.

Getting Fancy

this?

```
:
):
urn n + x
f

5)(6))
```

ction returns a function. The argument to **print** then
ction on 6.

s?

Answer

ith incr:

```
r(n):
 f(x):
 return n + x
urn f

ncr(5)(6))        print((λ n: return λ x:  n + x)(5)(6))
```

ts substituted for n:

```
λ x:  5 + x)(6)


 + 6)


1)
```


Trouble

latively simple (if tedious) approach doesn't work.

```
x)
```

bstitute for the first x as before:

```
    # or even worse:  4 = 8
4)
```

rong result (4 instead of 8).

bstitution, x isn't around any more to substitute for.

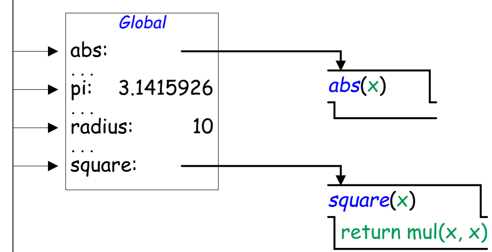
ething stronger.

Environments

ent is a mapping from names to values.

a name is *bound to* a value in this environment.

st form, it consists of a single *global environment frame*:

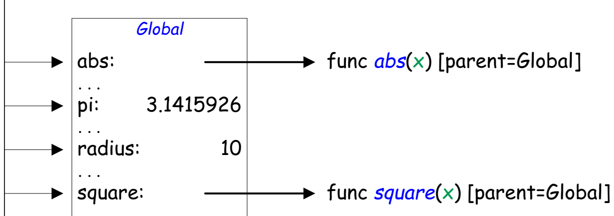



```
mport pi

x): return x**2
```

Slight Change of Notation

ng the Python Tutor from time to time, which uses a
fferent notation for function values. Might as well get
e'll explain the "parent=" stuff in a later lecture):




```
mport pi

x): return x**2
```


Environments and Evaluation

ssion is evaluated in an environment, which supplies the
any names in it.

expression typically involves first evaluating its subex-
e operators and operands of calls, the operands of con-
pressions such as x*(y+z), ...).

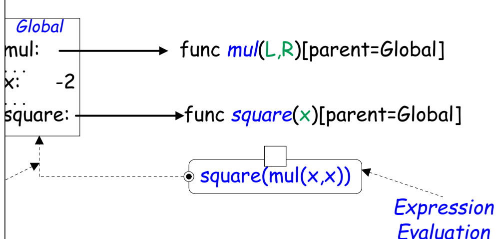
pressions are evaluated in the same environment as the
hat contains them.

ubexpressions (operator + operands) are evaluated, calls
ned functions must evaluate the expressions and state-
the definition of those functions.

uating User-Defined Function Calls

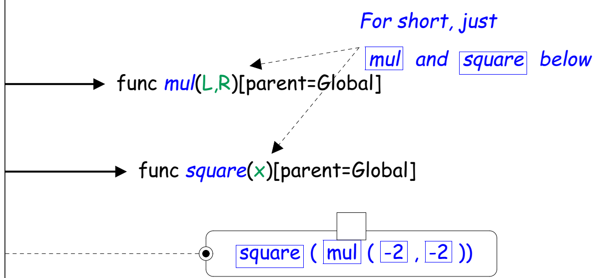
expression square(mul(x, x)) after executing

```
import mul
:
(x,x)
```



ting User-Defined Function Calls (II)

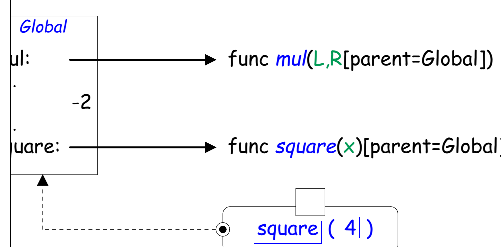
e the subexpressions of square(mul(x, x)) in the global



ubexpressions x, mul, and square take values from the
environment.

ing User-Defined Functions Calls (III)

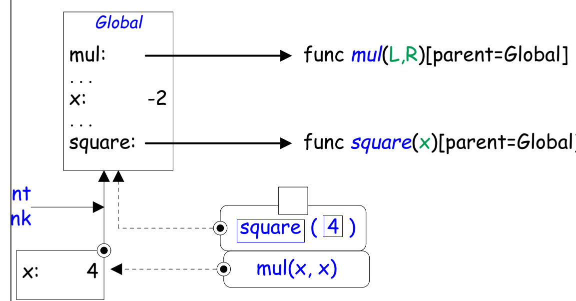
n the primitive multiply function:



ing User-Defined Functions Calls (IV)

arameter to user-defined square function, extend envi-
a *local environment frame*, attached to the frame in
was defined (the global one in this case), and giving x
value.

original call with evaluating body of square in the new
ment.



So How Does This Help?

problem that led to this whole environment diagram
w to deal with:

x)

. Each time we assign to *x*, we create a new binding for
ent evaluation frame (replacing the old one, if any).

new (last assigned) value when we look up *x* in the modi-
ment.

ting User-Defined Functions Calls (V)

luate mul(x, x) in this new environment, we get the same
re for mul, but the local value for x.

ting an identifier in a chain of environments, follow the
onment links to the first frame containing its definition.