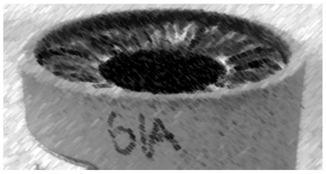


CS61A Lecture 13

Amir Kamil
UC Berkeley

February 20, 2013

Announcements

- HW4 due today at 11:59pm
- Hog contest deadline on Friday
  - Completely optional, opportunity for extra credit
  - See website for details

Converting Recursion to Iteration

Can be tricky! Iteration is a special case of recursion

Idea: Figure out what state must be maintained by the function

```
def summation(n, term):
    if n == 0:
        return 0
    return summation(n - 1, term) + term(n)
```

Initial value

Termination condition

What's summed so far?

How to get each incremental piece

```
def summation_iter(n, term):
    total = 0
    while n > 0:
        total, n = total + term(n), n – 1
    return total
```

Converting Iteration to Recursion

More formulaic: Iteration is a special case of recursion

Idea: The state of iteration can be passed as parameters

```
def fib_iter(n):
    if n == 0:
        return 0
    fib_n, fib_n_1, k = 1, 0, 1
    while k < n:
        fib_n, fib_n_1 = fib_n + fib_n_1, fib_n
        k = k + 1
    return fib_n
```

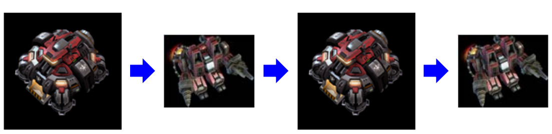
Local names become…

```
def fib_rec(n, fib_n, fib_n_1, k):
    if n == 0:
        return 0
    if k >= n:
        return fib_n
    return fib_rec(n, fib_n + fib_n_1, fib_n, k + 1)
```

Parameters in a recursive function

Mutual Recursion

Mutual recursion is when the recursive process is split across multiple functions



Decorating a recursive function generally results in mutual recursion

```
@trace1
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```

Example: http://goo.gl/4LZZv

Currying

We have used higher-order functions to produce a function to add a constant to its argument

What if we wanted to do the same for multiplication?

```
def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder
```

```
def make_multiplier(n):
    def multiplier(k):
        return mul(n, k)
    return multiplier
```

```
>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```

```
>>> make_multiplier(2)(3)
6
>>> mul(2, 3)
6
```

Same relationship between functions

How can we do this in general without repeating ourselves?

Currying

First, identify common structure.

Then define a function that generalizes the procedure.

```
def make_adder(n):
    def adder(k):
        return add(n, k)
    return adder

>>> make_adder(2)(3)
5
>>> add(2, 3)
5
```

```
def curry2(f):
    def outer(n):
        def inner(k):
            return f(n, k)
        return inner
    return outer

>>> curry2(mul)(2)(3)
6
>>> mul(2, 3)
6
```

This process of converting a multi-argument function to consecutive single-argument functions is called *currying*.

Functional Abstractions

```
def square(x):
    return mul(x, x)
```

```
def sum_squares(x, y):
    return square(x) + square(y)
```

What does **`sum_squares`** need to know about **`square`**?

- **`square`** takes one argument. Yes
- **`square`** has the intrinsic name square. No
- **`square`** computes the square of a number. Yes
- **`square`** computes the square by calling mul. No

```
def square(x):
    return pow(x, 2)
```

```
def square(x):
    return mul(x, x-1) + x
```

If the name "square" were bound to a built-in function, sum_squares would still work identically

What is Data?

Data: the things that programs fiddle with

Primitive values are the simplest type of data

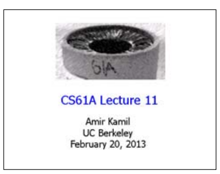
Integers: 2, 3, 2013, -837592010

Floating point (decimal) values: -4.5, 98.6

Booleans: True, False

How do we represent more complex data?

We need data abstractions!



Data Abstraction

Compound data combine smaller pieces of data together

- A date: a year, month, and day
- A geographic position: latitude and logitude

An abstract data type lets us manipulate compound data as a unit

All Programmers

Isolate two parts of any program that uses data

- How data are represented (as parts)
- How data are manipulated (as units)

Data abstraction: A methodology by which functions enforce an abstraction barrier between representation and use

Great Programmers

Rational Numbers

$$\frac{\text{numerator}}{\text{denominator}}$$

Exact representation of fractions

A pair of integers

As soon as division occurs, the exact representation is lost!

Assume we can compose and decompose rational numbers:

Constructor **`rational(n, d)`** returns a rational number *x*

Selectors

- **`numer(x)`** returns the numerator of *x*
- **`denom(x)`** returns the denominator of *x*

Rational Number Arithmetic

Example:

$$\frac{3}{2} * \frac{3}{5} = \frac{9}{10}$$

$$\frac{3}{2} + \frac{3}{5} = \frac{21}{10}$$

General Form:

$$\frac{nx}{dx} * \frac{ny}{dy} = \frac{nx*ny}{dx*dy}$$

$$\frac{nx}{dx} + \frac{ny}{dy} = \frac{nx*dy + ny*dx}{dx*dy}$$

Rational Number Arithmetic Code

```
def mul_rational(x, y):
    return rational(numer(x) * numer(y),
                    denom(x) * denom(y))
```

Constructor

Selectors

```
def add_rational(x, y):
    nx, dx = numer(x), denom(x)
    ny, dy = numer(y), denom(y)
    return rational(nx * dy + ny * dx, dx * dy)

def eq_rational(x, y):
    return numer(x) * denom(y) == numer(y) * denom(x)
```

Wishful thinking

- **rational(n, d)** returns a rational number *x*
- **numer(x)** returns the numerator of *x*
- **denom(x)** returns the denominator of *x*

Tuples

```
>>> pair = (1, 2)
>>> pair
(1, 2)
```

A tuple literal:
Comma-separated expression

```
>>> x, y = pair
>>> x
1
>>> y
2
```

"Unpacking" a tuple

```
>>> pair[0]
1
>>> pair[1]
2
>>> from operator import getitem
>>> getitem(pair, 0)
1
>>> getitem(pair, 1)
2
```

Element selection

More tuples next lecture

Representing Rational Numbers

```
def rational(n, d):
    """Construct a rational number x that represents
    n/d."""
    return (n, d)
```

Construct a tuple

```
from operator import getitem

def numer(x):
    """Return the numerator of rational number x."""
    return getitem(x, 0)

def denom(x):
    """Return the denominator of rational number
    x."""
    return getitem(x, 1)
```

Select from a tuple