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.

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

What's summed so far?
How to get each incremental piece?

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

Termination condition
Initial value

Converting Iteration to Recursion
More formulaic: Iteration is a special case of recursion.
Idea: The state of iteration can be passed as parameters.

```python
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
```

```python
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)
```

Local names become...
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.

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

Example: http://pyclipz.com

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?

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

```python
>>> make_adder(2)(3)
5
```

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

```python
>>> make_multiplier(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.

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

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

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

Functional Abstractions

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

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

• square takes one argument.
• square has the intrinsic name square.
• square computes the square of a number.

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

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

Data Abstraction

Compound data combine smaller pieces of data together

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

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

Rational Numbers

```
umerator
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:

```python
• rational(n, d) returns a rational number x
• numer(x) returns the numerator of x
• denom(x) returns the denominator of x
```

Rational Number Arithmetic

```
3 3
2 5
```

```
3 3
2 5
```

General Form:

```
x
dx
```

```
y
dy
```

```
x*ny

nx*dy + ny*dx
```

```
dx*dy
```

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

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)

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

A tuple literal:
Comma-separated expression
"Unpacking" a tuple
Element selection
More tuples next lecture

def rational(n, d):
    return (n, d)

def numer(x):
    return getitem(x, 0)

def denom(x):
    return getitem(x, 1)

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)

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

A tuple literal:
Comma-separated expression
"Unpacking" a tuple
Element selection
More tuples next lecture

def rational(n, d):
    return (n, d)

def numer(x):
    return getitem(x, 0)

def denom(x):
    return getitem(x, 1)

A tuple literal:
Comma-separated expression
"Unpacking" a tuple
Element selection
More tuples next lecture