61A LECTURE 7 – DATA ABSTRACTION

Steven Tang and Eric Tzeng

July 3, 2013

Announcements

- Trends project released later today. Due in ~2 weeks
- Extra midterm office hours Sunday, from noon to 6pm in 310 Soda
- Come relax at the potluck this Friday! 6-8pm at the Wozniak Lounge, 4th floor Soda.

Fibonacci sequence

- The Fibonacci sequence is defined as

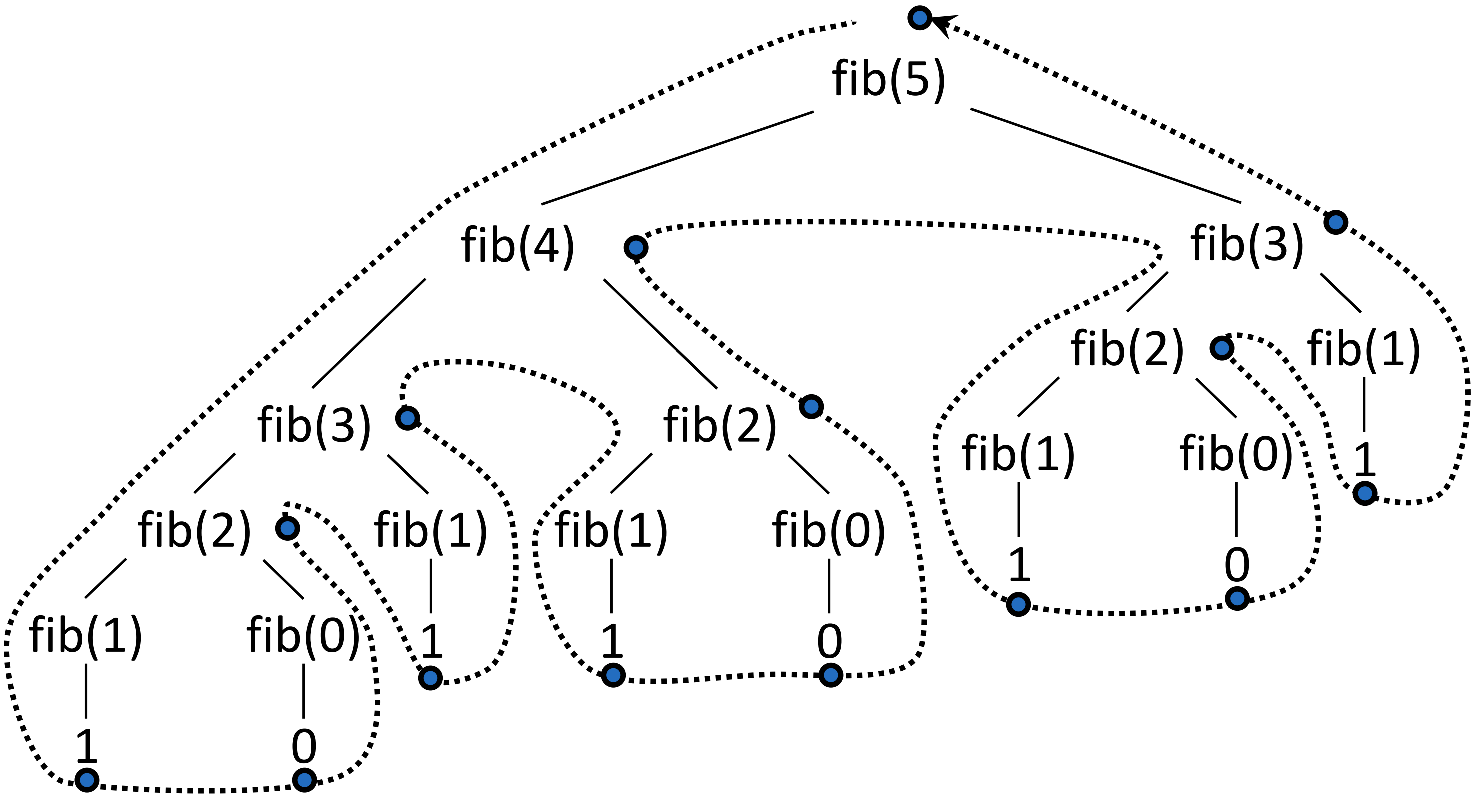
```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

Two recursive calls!

Tree recursion

Executing the body of a function may entail more than one recursive call to that function

This is called *tree recursion*



Trace demo

Tracing the Order of Calls

We can use a higher-order function to see the order in which calls are made and complete

```
def trace1(fn):
    """Return a function equivalent to fn that
    also prints trace output."""
    def traced(x):
        print('Calling', fn, '(', x, ')')
        res = fn(x)
        print('Got', res, 'from', fn, '(', x, ')')
        return res
    return traced

# Rebind the name fib to a traced version of fib
fib = trace1(fib)
```

Function Decorators

Function decorator

```
@trace1
def triple(x):
    return 3 * x
```

Decorated function

is identical to

Why not just use this?

```
def triple(x):
    return 3 * x
triple = trace1(triple)
```

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)
```

Termination condition

Initial value

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
```

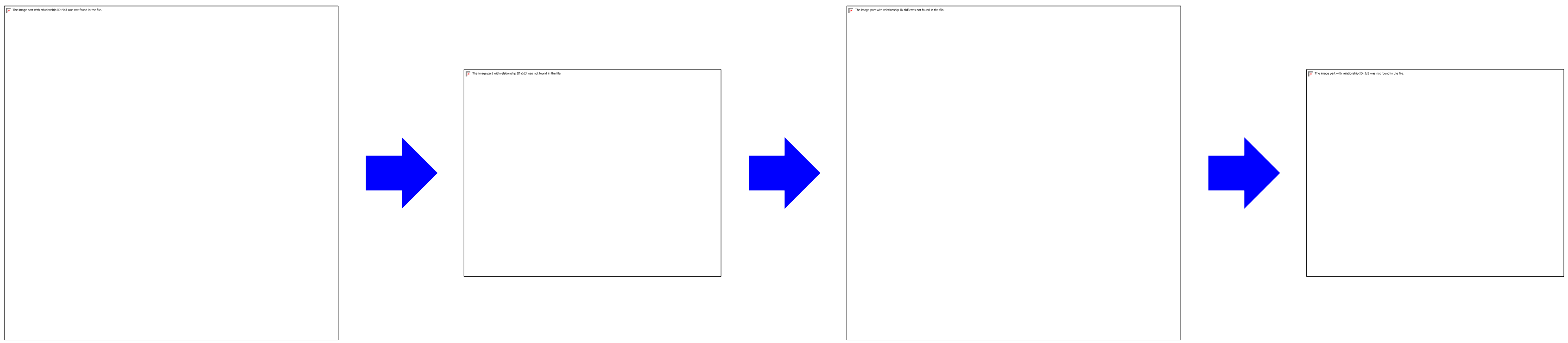
Local names become...

```
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/aNiDA

Currying

Recall the make_adder function from previous lectures.

We converted a multi-argument function, *add*, into a series of single-argument functions

```
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*.

Interpreter Session

Break

“Currying” is named after Haskell Curry

The concept of “currying” was originated by Moses Schönfinkel. A better name for currying might be “*schönfinkeling*”

Functional Abstractions

```
def square(x):              def sum_squares(x, y):
    return mul(x, x)            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):              def square(x):
    return pow(x, 2)            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 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

All Programmers

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."""
    ###YOUR CODE HERE


from operator import getitem

def numer(x):
    """Return the numerator of rational number x."""
    ###YOUR CODE HERE

def denom(x):
    """Return the denominator of rational number
    x."""
    ###YOUR CODE HERE
```

Interpreter Session

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

Reducing to Lowest Terms

Example:

$$\frac{3}{2} * \frac{5}{3} = \frac{5}{2} \qquad \frac{2}{5} + \frac{1}{10} = \frac{1}{2}$$

$$\frac{15}{6} * \frac{1/3}{1/3} = \frac{5}{2} \qquad \frac{25}{50} * \frac{1/25}{1/25} = \frac{1}{2}$$

```
from fractions import gcd
```

Greatest common divisor

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

Abstraction Barriers

Rational numbers as whole data values

`add_rational   mul_rational   eq_rational`

Rational numbers as numerators & denominators

`rational   numer   denom`

Rational numbers as tuples

`tuple   getitem`

However tuples are implemented in Python

Violating Abstraction Barriers

Does not use constructors

Twice!

```
add_rational( (1, 2), (1, 4) )
```

```
def divide_rational(x, y):
    return (x[0] * y[1], x[1] * y[0])
```

No selectors!

And no constructor!

What is an Abstract Data Type?

- We need to guarantee that constructor and selector functions together specify the right behavior.
- Behavior condition: If we construct rational number *x* from numerator *n* and denominator *d*, then **`numer(x)/denom(x)`** must equal *n/d*.
- An abstract data type is some collection of selectors and constructors, together with some behavior condition(s).
- If behavior conditions are met, the representation is valid.

You can recognize data types by behavior, not by bits

Behavior Conditions of a Pair

To implement our rational number abstract data type, we used a two-element tuple (also known as a pair).

What is a pair?

Constructors, selectors, and behavior conditions:

If a pair *p* was constructed from elements *x* and *y*, then

- `getitem_pair(p, 0)` returns *x*, and
- `getitem_pair(p, 1)` returns *y*.

Together, selectors are the inverse of the constructor

Generally true of container types.

Not true for rational numbers because of GCD

Functional Pair Implementation

```
def pair(x, y):
    """Return a functional pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch
```

This function represents a pair

Constructor is a higher-order function

```
def getitem_pair(p, i):
    """Return the element at index i of pair p."""
    return p(i)
```

Selector defers to the functional pair

Using a Functionally Implemented Pair

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

>>> getitem_pair(p, 0)
1

>>> getitem_pair(p, 1)
2
```

As long as we do not violate the abstraction barrier, we don't need to know that pairs are just functions

If a pair *p* was constructed from elements *x* and *y*, then

- `getitem_pair(p, 0)` returns *x*, and
- `getitem_pair(p, 1)` returns *y*.

This pair representation is valid!