MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.099—Introduction to EECS I
Spring Semester 2006

Lecture Notes – February 14

Capturing Common Patterns with Higher-Order Procedures

Answer to the nanoquiz problem (one of many possible good answers)

```
def div(n,d):
    x = n+1
    while True:
        if x % d == 0 and not x % d**2 == 0:
            return x
        x = x + 1
```

Answer to the homework problem on generating the binary representation of a number:

```
def bin(n):
    if n==0:
        return '0'
    elif n==1:
        return '1'
    else:
        return bin(n/2)+bin(n \% 2)
```

A procedure for computing the Fibonacci numbers, which generates a *tree-recursive process* that has exponential growth.

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

Three procedures for computing sums

```
def sumint(low,high):
    s=0
    x=low
    while x < high:
        s = s + x
        x = x + 1
    return s
```

```
def sumsquares(low,high):
    s=0
    x=low
    while x < high:
        s = s + x**2
        x = x + 1
    return s

##pi**2/8
def piSum(low,high):
    s=0
    x=low
    while x < high:
        s = s + 1.0/x**2
        x = x + 2
    return s
```

The general idea of summation, expressed as a procedure that captures the common pattern:

```
def sum(low,high,f,next):
    s=0
    x=low
    while x < high:
        s = s + f(x)
        x = next(x)
    return s
```

The sumint procedure, expressed as a general sum

```
def sumint(low,high):
    def identity(x): return x
    def add1(x): return x+1
    return sum(low,high,identity,add1)
```

The three sums, expressed in terms of the general idea of summation, using lambda to avoid having to name the internal procedures:

```
def sumsquares(low,high):
    return sum(low,
               high,
               lambda x: x**2,
               lambda x: x+1
               )

def piSum(low,high):
    return sum(low,
               high,
               lambda x: 1.0/x**2,
               lambda x: x+2
               )
```

Expressing a general method of finding a fixed point of a function f:

```
def fixedPoint(f,firstGuess):
    def close(g1,g2):
        return abs(g1-g2)<.0001
    def iter(guess,next):
        while True:
            if close(guess, next):
                return next
            else:
                guess=next
                next=f(next)
    return iter(firstGuess,f(firstGuess))
```

Then we can compute square roots as fixed points:

```
def sqrt(x):
    def average(a,b): return (a+b)/2.0
    return fixedPoint(lambda g: average(g,x/g),1.0)
```

Computing derivatives: Given a function f, the derivative Df is another function. Therefore D *itself* is a function whose value is a function:

```
def deriv(f):
    dx=0.0001
    return lambda x:(f(x+dx)-f(x))/dx
```

We can write this equivalently, without using `lambda`:

```
def deriv(f):
    dx=0.0001
    def d(x):
        return (f(x+dx)-f(x))/dx
    return d
```

In either case, if we apply `deriv` to a procedure, the result is another procedure, that we can then apply to a number, e.g.,

```
deriv(square)(10)
```

produces 20 (approximately) becasuse the derivative of $x \mapsto x^2$ is $x \mapsto 2x$.

Once we can express derivative, we can express Newton's method:

```
def newtonsMethod(f,firstGuess):
    return fixedPoint(
        lambda x: x - f(x)/deriv(f)(x),
        firstGuess)
```

and we can express computing square roots as an application of Newton's method:

```
def sqrt(x):
    return newtonsMethod(
        lambda y: y**2 - x,
        1.0)
```

Rights and privileges of first-class citizens in programming languages (Christopher Strachey)

- May be named by variables
- May be passed as arguments to procedures
- May be returned as results of procedures
- May be included in data structures

Memoization to avoid redundant computation:

```
def memoize(f):
    storedResults={}
    def doit(n):
        if storedResults.has_key(n):
            return storedResults[n]
        else:
            value = f(n)
            storedResults[n] = value
            return value
    return doit
```

Now we can remove redundant computaton for `fib` by executing

```
fib = memoize(fib)
```