Cursing and Recursing
CS21b: Structure and Interpretation of Computer Programs
Spring Term, 2016
Computing Square Roots -- a “fast path” to a real program

...an example of the use of recursion,
  a beginning methodology of program design,
  and a use and explanation of lexical scoping of variables...

Recall: `sqrt(x)` is the value y such that \( y^2 = x \)

\( (a \text{ DECLARATIVE DEFINITION [what is]} -- \text{by contrast, programs are IMPERATIVE DEFINITIONS [how to}}) \)

"Wishful thinking" method of programming

\[
\text{(define (sqrt-iter guess x)}
  \text{(if (good-enough? guess x)}
    \text{guess}
    \text{(sqrt-iter (improve-guess guess x) x) }}))
\]

Now, we need code for \( \text{good-enough?} \) and \( \text{improve-guess} \) ...
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve-guess guess x) x)))

(define (good-enough? guess x)
  (< (abs (- (square guess) x)) .001))

Now we use Newton's Method to generate new guesses:

initial guess: \( g = 1 \)
next, better guess: \( g' \) (a function of \( g \)) = \( (g + x/g)/2 \)

Why does this method work?? The "square box" argument...

Claim (to be shown): This approximation method gains one bit of accuracy for every iteration...

(define (improve-guess guess x)
  (average guess (/ x guess)))

(define (sqrt x) (sqrt-iter 1 x))

[1 is the initial guess...]

To compute \( \sqrt{n} \), start with an initial guess \( g \) (say, \( g = n \)). If you think \( g \) is good enough (i.e., \( g^2 \) is close enough to \( n \)), then stop. Otherwise, replace \( g \) by the improved guess \( I(g) \), where we define \( I(x) = \frac{1}{2}(x + \frac{n}{x}) \).

Stated alternatively, we compute \( g, I(g), I(I(g)), I(I(I(g))), \ldots \) until we have found a good enough approximation to \( \sqrt{n} \).

**Claim:** If \( x \geq \sqrt{n} \), then \( I(x) \geq \sqrt{n} \) also.

Observe that the stated conclusion is equivalent to the following:

\[
I(x) \geq \sqrt{n} \quad \equiv \quad \frac{1}{2}(x + \frac{n}{x}) \geq \sqrt{n} \\
\iff (x + \frac{n}{x}) \geq 2\sqrt{n} \\
\iff x^2 - 2\sqrt{n}x + n \geq 0.
\]

The quadratic describes a parabola \( ax^2 + bx + c \) opening upwards, and takes its minimum value at \( x = -\frac{b}{2a} = \frac{2\sqrt{n}}{2} = \sqrt{n} \), at which point its value is 0.
Claim:
If \( x \geq \sqrt{n} \), then
\[
\frac{I(x) - \sqrt{n}}{x - \sqrt{n}} \leq 1/2
\]

Note that the inequality is equivalent to the following:
\[
\frac{\frac{1}{2}(x + \frac{n}{x}) - \sqrt{n}}{x - \sqrt{n}} \leq 1/2 \iff x + \frac{n}{x} - 2\sqrt{n} \leq x - \sqrt{n}
\]
\[\iff x + \frac{n}{x} \leq x + \sqrt{n}\]
\[\iff \frac{n}{x} \leq \sqrt{n}\]
\[\iff \sqrt{n} = \frac{n}{\sqrt{n}} \leq x\]

Because the last inequality is true (by assumption), so is the first (the one we wanted to prove).
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve-guess guess x) x)))

(define (good-enough? guess x)
  (< (abs (- (square guess) x)) .001))

(define (improve-guess guess x)
  (average guess (/ x guess)))

(define (sqrt x) (sqrt-iter 1 x))

Substitution model:

(sqrt 2)
(sqrt-iter 1 2)
(if (good-enough? 1 2) 1 (sqrt-iter (improve-guess 1 2) 2))
(sqrt-iter (improve-guess 1 2) 2)
(sqrt-iter (average 1 (/ 2 1)) 2)
(sqrt-iter 1.5 2)
...
(sqrt-iter 1.41666666666667 2)
...

[recall the answer is 1.4142...]
Naming and the environment

Idea: the names of formal parameters ("internal variables") don’t matter, but the names of external variables do matter.

(Notice the binding [definition] of a parameter, as opposed to the occurrence giving its use...)

So which are the same? Use the substitution model to find out:

\[
\begin{align*}
(\text{define } (\text{square } x) & \ (\ast \ x \ x)) \\
(\text{define } (\text{square } z) & \ (\ast \ z \ z)) \\
(\text{define } (\text{squareplus } x) & \ (+ \ (\ast \ x \ x) \ y)) \\
(\text{define } (\text{squareplus } z) & \ (+ \ (\ast \ z \ z) \ y)) \\
(\text{define } (\text{squareplus } x) & \ (+ \ (\ast \ x \ x) \ y)) \\
(\text{define } (\text{squareplus } z) & \ (+ \ (\ast \ z \ z) \ w))
\end{align*}
\]

Try \(\text{square } 5\)

Try \(\text{squareplus } 10\)

Where do the values of the external, free variables come from?
**Block structure---as supported by the substitution model...**

Idea: when \((\sqrt{2})\) is evaluated, 2 is substituted for \(x\) in the three definitions, which are *internal* to \(\sqrt{\cdot}\).

```
(define (sqrt x)
    (define (good-enough? guess)
        (< (abs (- (square guess) x)) .001))
    (define (improve-guess guess)
        (average guess (/ x guess)))
    (define (sqrt-iter guess)
        (if (good-enough? guess)
            guess
            (sqrt-iter (improve-guess guess))))
    (sqrt-iter 1))
```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) .001))
  (define (improve-guess guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve-guess guess))))
  (sqrt-iter 1))

Evaluating (sqrt 2) in the substitution model, we get:

  (define (good-enough? guess)
    (< (abs (- (square guess) 2)) .001))
  (define (improve-guess guess)
    (average guess (/ 2 guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve-guess guess))))
  (sqrt-iter 1))
Block structure: another version...

(define (sqrt x)
    (define (sqrt-iter guess)
        (define (good-enough?) a procedure with no parameters!
            (< (abs (- (square guess) x))
                .001))
        (define (improve-guess) ...and another one...
            (average guess (/ x guess)))
        (if (good-enough?)
            guess
            (sqrt-iter (improve-guess))))
    (sqrt-iter 1))
(define (sqrt x)
  (define (sqrt-iter guess)
    (define (good-enough?) a procedure with no parameters!
      (< (abs (- (square guess) x))
         .001))
    (define (improve-guess) ...and another one...
      (average guess (/ x guess)))
    (if (good-enough?)
      guess
      (sqrt-iter (improve-guess)))
  (sqrt-iter 1))

(sqrt 2) evaluates to:

  (define (sqrt-iter guess)
    (define (good-enough?)
      (< (abs (- (square guess) 2))
         .001))
    (define (improve-guess)
      (average guess (/ 2 guess)))
    (if (good-enough?)
      guess
      (sqrt-iter (improve-guess)))
  (sqrt-iter 1)
(define (sqrt-iter guess)
  (define (good-enough?) a procedure with no parameters!
    (< (abs (~- (square guess) 2))
        .001))
  (define (improve-guess) ...and another one...
    (average guess (~2 guess))
  (if (good-enough?)
      guess
      (sqrt-iter (improve-guess))))

(sqrt-iter 1) evaluates to

(define (good-enough?)
  (< (abs (~- (square 1) 2))
      .001))
(define (improve-guess)
  (average 1 (~2 1))
  (if (good-enough?)
      1
      (sqrt-iter (improve-guess))))

and (if ...) evaluates to (sqrt-iter 1.5)
Commands versus expressions...

Commands do something (read-eval-print, input and output), and the order in which you execute commands matters (e.g., pie à la mode, with ice cream on top).

Expressions (read-eval-print) evaluate to something, and the order in which you evaluate expressions basically doesn’t matter---you should get the same answer, though perhaps with different efficiency...

Commands “change the world”, expressions only “observe” the world. Example: a bank account function (deposit \( n \)), returning a balance, versus (factorial \( n \)).
Scheme: First you **curse**, then you **recurse**...

*That old sawhorse: computing factorials:*

\[
\begin{align*}
0! &= 1 \\
n! &= n \times (n-1)!
\end{align*}
\]

```
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1))))
)
```

;Value: factorial

```
(factorial 5)
```

;Value: 120
Substitution model:

(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))

(factorial 5)
(if (= 5 0) 1 (* 5 (factorial (- 5 1))))
(* 5 (factorial (- 5 1)))
...

Note the **special form** (why?)
(if <predicate> <consequent> <alternative>)

Evaluation rule for (if ...):

1. Evaluate <predicate>;
2. If evaluation returns #t (true), entire expression evaluates to what <consequent> evaluates to;
3. Otherwise, entire expression evaluates to what <alternative> evaluates to.
Substitution model:

(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1))))
)

(factorial 5)
(if (= 5 0) 1 (* 5 (factorial (- 5 1))))
(* 5 (factorial (- 5 1)))
...
(* 5 (factorial 4))
...
(* 5 (* 4 (factorial 3)))
...
(* 5 (* 4 (* 3 (* 2 (* 1 (factorial 0))))))
(* 5 (* 4 (* 3 (* 2 (* 1 (if (= 0 0) 1 (* 0 (factorial (- 0 1))))))))))
(* 5 (* 4 (* 3 (* 2 (* 1 1)))))
(* 5 (* 4 (* 3 (* 2 1))))
(* 5 (* 4 (* 3 2)))
(* 5 (* 4 6))
(* 5 24)
120
Time and space resources

(factorial 5)
(if (= 5 0) 1 (* 5 (factorial (- 5 1))))
(* 5 (factorial (- 5 1))
(* 5 (factorial 4))
...
(* 5 (* 4 (factorial 3)))
...
(* 5 (* 4 (* 3 (* 2 (* 1 (factorial 0))))))
(* 5 (* 4 (* 3 (* 2 (* 1 (if (= 0 0) 1 (* 0 (factorial (- 0 1))))))))
(* 5 (* 4 (* 3 (* 2 (* 1 1)))))
(* 5 (* 4 (* 3 (* 2 1))))
(* 5 (* 4 (* 3 2)))
(* 5 (* 4 6))
(* 5 24)
120

Time = vertical axis; Space = horizontal axis Why?

This computational process is linear in time and space -- horizontal, vertical grow linearly with parameter.
Alternative iterative version of factorial:

(define (fact-iter prod n)
  (if (= n 0)
      prod
      (fact-iter (* prod n) (- n 1))))
;Value: fact-iter

(define (factorial n) (fact-iter 1 n))
;Value: factorial

(factorial 5)
(fact-iter 1 5)
(if (= 5 0) 1 (fact-iter (* 1 5) (- 5 1)))
(fact-iter 5 4)
(if (= 4 0) 5 (fact-iter (* 5 4) (- 4 1)))
(fact-iter 20 3)
...
(fact-iter 60 2)
...
(fact-iter 120 1)
...
(fact-iter 120 0)
(if (= 0 0) 120 (fact-iter (* 120 0) (- 0 1)))
120

This process is
Linear time, constant space
Why?
Another belabored example of recursion: Fibonacci numbers

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 145, ...

(define (fib n)
    (if (< n 2)
        n
        (+ (fib (- n 1)) (fib (- n 2)))))

;Value: fib

(fib 10)
;Value: 55

(fib 5)
(+ (fib 4) (fib 3))
(+ (+ (fib 3) (fib 2)) (+ (fib 2) (fib 1)))
(+ (+ (+ (fib 2) (fib 1)) (+ (fib 1) (fib 0)))
    (+ (+ (fib 1) (fib 0)) 1))
(+ (+ (+ (fib 1) (fib 0)) (fib 1)) (+ (fib 1) (fib 0)))
(+ (+ (fib 1) (fib 0)) 1))

(fib n) grows exponentially in n, around \((1/\sqrt{5}) \left[\left(1+\sqrt{5}\right)/2\right]^n\) --- recall \(\sqrt{5} = 2.236\ldots\)

How many calls to \((\text{fib} \ 0)\) or \((\text{fib} \ 1)\) --- also exponential!!

\(C(n) = C(n-1) + C(n-2)\quad C(0) = C(1) = 1\quad [F(n) \text{ “shifted over” by } 1]\)
Iterative Fibonacci numbers (why does it work? what is it doing?)

```
(define (fib-iter a b count max)
    (if (= count max)
        b
        (fib-iter b (+ a b) (1+ count) max)))
;Value: fib-iter

(define (fib n) (fib-iter 1 0 0 n))
;Value: fib

(fib 10)
;Value: 55
```

Substitution model:

```
(fib 10)
(fib-iter 1 0 0 10)
(if (= 0 10) 0 (fib-iter 0 (+ 1 0) (1+ 0) 10))
(fib-iter 0 1 1 10)
(if (= 1 10) 1 (fib-iter 1 (+ 0 1) (1+ 1) 10))
(fib-iter 1 1 2 10)
(if (= 2 10) 1 (fib-iter 1 (+ 1 1) (1+ 2) 10))
(fib-iter 1 2 3 10)
(fib-iter 2 3 4 10)
(fib-iter 3 5 5 10)
...
(fib-iter 34 55 10 10)
55
```

Analysis: linear time, constant space (why)?
Another example: Fast exponential

\[ b^0 = 1 \]
\[ b^{2n} = (b^n)^2 \]
\[ b^{2n+1} = b \times b^{2n} \]

```
(define (expt b n)
    (cond ((= n 0) 1)
          ((even? n) (square (expt b (/ n 2))))
          (else (* b (expt b (- n 1))))))
```

;Value: expt

```
(expt 2 3)
```
;Value: 8

Note use of conditional cond ... a nested if, with a catchall else clause ...
Substitution model: (leaving b indeterminate)

```
(expt b 11)
```
Fast exponential: substitution model (leaving b indeterminate)

(define (expt b n)
  (cond ( (= n 0) 1)
       ((even? n) (square (expt b (/ n 2))))
       (else (* b (expt b (- n 1))))))

;Value: expt

(expt b 11)
(* b (expt b 10))
(* b (square (expt b 5)))
(* b (square (* b (expt b 4))))
(* b (square (* b (square (expt b 2)))))
(* b (square (* b (square (square (expt b 1))))))
(* b (square (* b (square (square (square (* b (expt b 0))))))))
(* b (square (* b (square (square (* b 1))))))
(* b (square (* b (square (square b)))))
(* b (square (* b (square b^2))))
(* b (square (* b b^4)))
(* b (square b^5))
(* b b^10)
b^{11}

Analysis: logarithmic time and space
Iterative version of fast exponentiation:

```
(define (expt-iter acc b e)
  (cond ((= e 0) acc)
        ((even? e) (expt-iter acc (square b) (/ e 2)))
        (else (expt-iter (* acc b) b (- e 1))))))
;Value: expt-iter
```

```
(define (expt b e) (expt-iter 1 b e))
;Value: expt
```

```
(expt 2 3)
;Value: 8
```

Termination variant: every call to `expt-iter` decreases $e$

Substitution model:

```
(expt b 11)
(expt-iter 1 b 11)
(expt-iter b b 10)
(expt-iter b b^2 5)
(expt-iter b^3 b^2 4)
(expt-iter b^3 b^4 2)
(expt-iter b^3 b^8 1)
(expt-iter b^{11} b^8 0)
b^{11}
```

Correctness invariant:

```
(expt-iter acc b e) = acc * b^e
(by induction!)
```

Analysis: logarithmic time, constant space
Using fast exponentiation to derive a fast Fibonacci algorithm

\[
\begin{pmatrix}
1 & 1 \\
1 & 0
\end{pmatrix}
\begin{pmatrix}
F_{k+1} \\
F_k
\end{pmatrix}
= 
\begin{pmatrix}
F_{k+2} \\
F_{k+1}
\end{pmatrix}
\]

Idea: to compute \( F_k \), take square matrix \( M \) above, compute \( M^{k-1} \)

\[
\begin{pmatrix}
1 & 1 \\
1 & 0
\end{pmatrix}^{k-1}
\begin{pmatrix}
1 \\
0
\end{pmatrix}
= 
\begin{pmatrix}
F_k \\
F_{k-1}
\end{pmatrix}
\]

(define (matrix-expt b n)
  (cond ((= n 0) 1)
        ((even? n)
         (matrix-square (matrix-expt b (/ n 2)))))
    (else
     (matrix-* b (matrix-expt b (- n 1))))))
Why this works...

\[
\begin{pmatrix}
1 & 1 \\
1 & 0
\end{pmatrix}
= 
\begin{pmatrix}
F_2 & F_1 \\
F_1 & F_0
\end{pmatrix}
\]

\[
\begin{pmatrix}
F_k & F_{k-1} \\
F_{k-1} & F_{k-2}
\end{pmatrix}
\begin{pmatrix}
1 & 1 \\
1 & 0
\end{pmatrix}
= 
\begin{pmatrix}
F_{k+1} & F_k \\
F_k & F_{k-1}
\end{pmatrix}
\]
Another logarithmic way to compute Fibonacci numbers...

\[ \phi = \frac{1 + \sqrt{5}}{2} \quad \text{ (} \phi + 1 = \phi^2 \text{)} \]

\[ F_k = \frac{\phi^k - (1 - \phi)^k}{\sqrt{5}} \quad \text{(proof by induction)} \]

Since the \( k \)th number is defined by an exponential (in \( k \)), we can compute it in \( O(\log k) \) time...
Conclusion:

Combining the clever multiplication (logarithmic time in exponent) with 2x2 matrix multiplication (constant time), we get a **logarithmic** time algorithm for computing Fibonacci numbers -- this a reduction from the original **exponential** time algorithm.

Is this really true? What cost assumptions are we making? (Think about size of numbers, cost of multiplying and adding big integers --- which we've considered to be "constant time".)
Tail Recursion and the Actor Model

Euclid’s algorithm for computing greatest common divisors:

(define (gcd a b)
  (if (= b 0)
    a
    (gcd b (remainder a b))))

Substitution model:

(gcd 21 13)
(gcd 13 8)
(gcd 8 5)
(gcd 5 3)
(gcd 3 2)
(gcd 2 1)
(gcd 1 0)

Where have you seen these numbers before?
Correctness

\[
\text{(define (gcd a b)}
\text{ (if (= b 0)
\text{ a
\text{ (gcd b (remainder a b))))})
\]

Why does this algorithm terminate? Observe that if \( b < a \), then
\[
b + \text{rem}(a, b) < a + b.
\]

Why does \text{gcd} give the right answer? Observe that if \( a = kb + r \), then
\[
gcd(a, b) = gcd(kb + r, b) = gcd(b, r).
\]

Why does \text{gcd} give the answer in \( O(\log a + \log b) \) iterations? Observe that the number of bits decreases:
\[
|b| + |\text{rem}(a, b)| < |a| + |b|
\]
Tail recursion: no work builds up

(define (gcd a b)
  (if (= b 0)
    a
    (gcd b (remainder a b))))

(gcd 21 13)
(gcd 13 8)
(gcd 8 5)
(gcd 5 3)
(gcd 3 2)
(gcd 2 1)
(gcd 1 0)
1

(Proof by example) that
(gcd F_{k+1} F_k)=1

This recursion takes \( k+O(1) \) steps -- but \( F_k \) is about \( 1.6^k \) -- thus \( \Omega(\log n) \) steps are required by gcd.
Tail recursion

There are different syntactic kinds of recursion. The tail recursive version is easier to implement.

(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1))))

(fact 5)
(* 5 (fact 4))
(* 5 (* 4 (fact 3))
(* 5 (* 4 (* 3 (fact 2))))
(* 5 (* 4 (* 3 (* 2 (fact 1))))
(* 5 (* 4 (* 3 (* 2 (* 1 (fact 0))))))
(* 5 (* 4 (* 3 (* 2 (* 1 1))))))
...

ordinary recursion: work builds up...
Tail recursion

There are different syntactic kinds of recursion. The tail recursive version is easier to implement.

\[
\text{(define (fact-iter n a)}
\text{(if (= n 0)}
\text{ a)

\text{((- n 1)}
\text{ (* n a)))})
\]

tail recursion: no work builds up...

(fact-iter 5 1)
(fact-iter 4 5)
(fact-iter 3 20)
(fact-iter 2 60)
(fact-iter 1 120)
(fact 0 120)
120
recursive
delayed work
recursive call
tail recursive

\[
\text{fact 5) = 5! = 120}
\]
\[
\text{fact 4) = 4! = 24}
\]
\[
\text{fact 3) = 3! = 6}
\]
\[
\text{fact 2) = 2! = 2}
\]
\[
\text{fact 1) = 1! = 1}
\]
\[
\text{fact 0) = 0! = 1}
\]

\[
\text{fact-iter 5 1)
\]
\[
\text{fact-iter 4 5)
\]
\[
\text{fact-iter 3 20)
\]
\[
\text{fact-iter 2 60)
\]
\[
\text{fact-iter 1 120)
\]
\[
\text{fact-iter 0 120)
\]

why the procedural redundancy here?
recursive

(fact 5)
(* 5 □) 120
(* 6 24) 120
(* 4 □) 24
(* 4 6) 24
(fact 4)
(* 3 □) 6
(* 3 2) 6
(fact 3)
(* 2 □) 2
(* 2 1) 2
(fact 2)
(* 1 □) 1
(* 1 1) 1
(fact 1)
(* 1 □) 1
(* 1 1) 1
(fact 0)

(there’s really only one “process”)

tail recursive

(fact-iter 5 1)
(fact-iter 4 5)
(fact-iter 3 20)
(fact-iter 2 60)
(fact-iter 1 120)
(fact-iter 0 120)
“What is that lambda thing?”

(define (square x) (* x x))

or, if you prefer...

(define square (lambda (x) (* x x)))

The reason you give names to things is so that you can refer to them repeatedly...