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 DECLARATIVE DEFINITION [what is] -- by contrast, programs are
IMPERATIVE DEFINITIONS [how to])

"Wishful thinking" method of programming

(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve-guess guess x)
                 x)))

Now, we need code for good-enough? and 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 \( g \)) = \( (g + \frac{x}{g})/2 \)

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

Claim: 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...]
(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 1.4166666666666667 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:

```
(define (square x) (* x x))
(define (square z) (* z z))

(define (squareplus x) (+ (* x x) y))
(define (squareplus z) (+ (* z z) y))

(define (squareplus x) (+ (* x x) y))
(define (squareplus z) (+ (* z z) w))
```

Try (square 5)
Try (squareplus 10)

Where do the values of the external, free variables come from?
Block structure

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

```
(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)
(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)
That old sawhorse: computing factorials:

\[ 0! = 1 \]
\[ n! = n \times (n-1)! \]

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

(Value: factorial)

(factorial 5)

(Value: 120)
Substitution model:

\[
\text{(define (factorial n)} \\
\quad \text{(if (= n 0)} \\
\quad \quad 1 \\
\quad \quad (* n (factorial (- n 1)))))}
\]

\[
\text{(factorial 5)} \\
\quad \text{(if (= 5 0) 1 (* 5 (factorial (- 5 1))))} \\
\quad (* 5 (factorial (- 5 1))) \\
\quad \ldots
\]

Note the special form (why?)
\[
\text{(if <predicate> <consequent> <alternative>)}
\]

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:

\[
\text{(define (factorial n)}
\begin{align*}
& \quad \text{if} \; (= \; n \; 0) \\
& \quad \quad 1 \\
& \quad (* \; n \; (\text{factorial} \; (- \; n \; 1)))
\end{align*}
\]

\[
\text{(factorial 5)} \\
\text{(if} \; (= \; 5 \; 0) \; 1 \; (* \; 5 \; (\text{factorial} \; (- \; 5 \; 1)))) \\
(* \; 5 \; (\text{factorial} \; (- \; 5 \; 1)) \\
\ldots \\
(* \; 5 \; (\text{factorial} \; 4)) \\
\ldots \\
(* \; 5 \; (* \; 4 \; (\text{factorial} \; 3))) \\
\ldots \\
(* \; 5 \; (* \; 4 \; (* \; 3 \; (* \; 2 \; (* \; 1 \; (\text{factorial} \; 0)))))) \\
(* \; 5 \; (* \; 4 \; (* \; 3 \; (* \; 2 \; (* \; 1 \; (\text{if} \; (= \; 0 \; 0) \; 1 \; (* \; 0 \; (\text{factorial} \; (- \; 0 \; 1)))))))))) \\
(* \; 5 \; (* \; 4 \; (* \; 3 \; (* \; 2 \; (* \; 1 \; 1))))) \\
(* \; 5 \; (* \; 4 \; (* \; 3 \; (* \; 2 \; 1)))) \\
(* \; 5 \; (* \; 4 \; (* \; 3 \; 2))) \\
(* \; 5 \; (* \; 4 \; 6)) \\
(* \; 5 \; 24) \\
120
\]
Time and space resources

\[(\text{factorial } 5)\]
\[(\text{if } (= 5 0) 1 (* 5 (\text{factorial} \ (- 5 1))))\]
\[(* 5 (\text{factorial} \ (- 5 1)))\]
\[\ldots\]
\[(* 5 (\text{factorial} \ 4))\]
\[\ldots\]
\[(* 5 (* 4 (\text{factorial} \ 3)))\]
\[\ldots\]
\[(* 5 (* 4 (* 3 (* 2 (* 1 (\text{factorial} \ 0))))))\]
\[(* 5 (* 4 (* 3 (* 2 (* 1 (\text{if } (= 0 0) 1 (* 0 (\text{factorial} \ (- 0 1))]))))))\]
\[(* 5 (* 4 (* 3 (* 2 (* 1 1)))))\]
\[(* 5 (* 4 (* 3 (* 2 1))))\]
\[(* 5 (* 4 (* 3 2)))\]
\[(* 5 (* 4 6))\]
\[(* 5 24)\]
\[120\]

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 \((1/\sqrt{5}) [(1+\sqrt{5})/2]^n\) --- recall \(\sqrt{5}= 2.236...\)

How many calls to (fib 0) or (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?)

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

(defun (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

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

\[
\text{(define (expt b n)}
\begin{array}{ll}
\text{(cond } & (\text{if } n = 0 \text{ then } 1)) \\
\text{((even? n) } & \text{square (expt b (/ n 2)))} \\
\text{(else } & (* b \text{ (expt b (- n 1))))}) \\
\end{array}
\]

;Value: expt

(expt 2 3)
;Value: 8

Note use of conditional \texttt{cond} ... a nested \texttt{if}, with a catchall \texttt{else} clause ...
Substitution model: (leaving \texttt{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 (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
```

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)
```

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

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

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

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

(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))))))
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

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

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

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

Why does gcd give the answer in \( O(\log a + \log b) \) iterations? Observe that \( |b| + |a \text{ remainder } b| < |a| + |b| \).
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 \( \text{gcd} \).