Announcements / Notes

- Deity and Lisp
  - http://xkcd.com/c224.html (last Friday’s xkcd comic)

- The “One λ” (useless brownie points for interpreting the inscription!)

- Documentation Resources
  - DrScheme’s help system
  - http://www.drscheme.org (when the webserver is working)

- Personal edification: last Friday’s prime? problem

**let Special Form:** (**let** **bindings** **body**)

Binds the given **bindings** for the duration of the body. The **bindings** is a list of **(name value)** pairs. The **body** consists of one or more expressions which are evaluated in order and the value of last is returned.

Desugaring example:

```scheme
(let ((a 10)
      (b 20))
  (+ a b))
```

is *exactly* equivalent to:

```scheme
((lambda (a b)
      (+ a b))
  10 20)
```

This will be handy in project 1. C++ programmers: Scheme’s way of making **const** local variables (later we’ll talk about modifying the values).
How to identify recursive and iterative processes

By staring at code (rules of thumb):

<table>
<thead>
<tr>
<th>Recursive Process</th>
<th>Iterative Process (tail recursive)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is there a RECURSIVE CALL? (it may be indirect)</td>
<td></td>
</tr>
<tr>
<td>Is there something “wrapped” around the recursive call? (deferred operations)</td>
<td></td>
</tr>
<tr>
<td>Is there an extra variable that stores the INTERMEDIATE RESULT?</td>
<td></td>
</tr>
<tr>
<td>Is there a COUNTER variable?</td>
<td></td>
</tr>
<tr>
<td>Are there helper functions (often needed to keep track of these other variables)</td>
<td></td>
</tr>
</tbody>
</table>

By putting an example through the substitution model:

What is the “shape” of the rewrites?

By analysis of space and time, i.e. order of growth (the real way):

Space – width of the substitution model (characters have to be stored somewhere... )

Time – length of substitutions (each simplification/rewrite takes 1 unit of time)

Analysis Problem

Consider the following problem:

```
(define (bar a b)
  (bar-helper 0 a b))
(define (bar-helper c a b)
  (if (> a b)
    c
    (bar-helper (+ c a) (+ a 1) b)))
```

What’s the order of growth in space and time? ?

Is it recursive or iterative? 

Write the other:

```
(define (bar-rec a b)
```

2


Cubic Roots

We are now going to create and analyze some methods of finding the zeros of a cubic equation, i.e. given \( a, b, c, \) and \( d \) find values of \( x \) for which \( ax^3 + bx^2 + cx + d = 0 \).

Assume we’ve been supplied with two guesses for \( x \) and the coefficients. If either guess gives a solution that is close enough to zero, return the guess. If not, then you have lots of choices. One is to move each guess towards the other by a slight amount and continue, another is to split the domain in two and try both halves (e.g. if the guesses are \( g_1 \) and \( g_2 \), then try \( g_1 \) and \((g_1 + g_2)/2\) and \((g_1 + g_2)/2\) and \( g_2 \).

In class, we’ll now go through the process of designing the above two versions of \texttt{find-cubic-root}, discussing modularity, orders of growth, recursion vs. iteration, accuracy, and other fun concepts. Your instructor’s solution will be posted at \url{http://people.csail.mit.edu/dalleyg/6.001/SP2007/index.html}.

\texttt{;; find-cubic-root solutions...}
Challenge Problem

;; Challenge Problem:
;; a) Is this function iterative or recursive?
;; b) What is its order-of-growth in time? space?
;; c) What does this thing actually do (hint: 18.02)?
;; d) Rewrite as recursive/iterative (which ever this is not).
;; e) What is the order of growth for your new version in time? space?
(define (baz n)
  (define (qux a b c)
    (if (> a b)
      c
      (qux (+ a 1)
        b
        ((if (even? a) - +)
          c (/ (- (* a 2) 1)))))
  (qux 1 n 0))
6.001 Recitation 5: More Orders of Growth
RI: Gerald Dalley, dalleyg@mit.edu
21 Feb 2007

Announcements / Notes

• Deity and Lisp
  – http://xkcd.com/c224.html (last Friday’s xkcd comic)

• The “One λ” (useless brownie points for interpreting the inscription!)

• Documentation Resources
  – DrScheme’s help system
  – http://www.drscheme.org (when the webserver is working)

• Personal edification: last Friday’s prime? problem

let Special Form: (let bindings body)

Binds the given bindings for the duration of the body. The bindings is a list of (name value) pairs. The body consists of one or more expressions which are evaluated in order and the value of last is returned.

Desugaring example:

(let ((a 10)
      (b 20))
  (+ a b))

is exactly equivalent to:

((lambda (a b)
    (+ a b))
  10 20)

This will be handy in project 1. C++ programmers: Scheme’s way of making const local variables (later we’ll talk about modifying the values).
How to identify recursive and iterative processes

By staring at code (rules of thumb):

<table>
<thead>
<tr>
<th>Recursive Process</th>
<th>Iterative Process (tail recursive)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is there a RECURSIVE CALL? (it may be indirect)</td>
<td>yes</td>
</tr>
<tr>
<td>Is there something “wrapped” around the recursive call? (deferred operations)</td>
<td>yes</td>
</tr>
<tr>
<td>Is there an extra variable that stores the INTERMEDIATE RESULT?</td>
<td>no</td>
</tr>
<tr>
<td>Is there a COUNTER variable?</td>
<td>yes</td>
</tr>
<tr>
<td>Are there helper functions (often needed to keep track of these other variables)</td>
<td>rarely</td>
</tr>
</tbody>
</table>

By putting an example through the substitution model:

| What is the “shape” of the rewrites? | wide (deferred ops) and long (recursive calls) | narrow (no deferred ops, doesn’t get wider with “larger” inputs) and long (recursive calls) |

By analysis of space and time, i.e. order of growth (the real way):

| Space – width of the substitution model (characters have to be stored somewhere...) | not Θ(1) | Θ(1) |
| Time – length of substitutions (each simplification/rewrite takes 1 unit of time) | anything | anything, same as recursive |

Analysis Problem

Consider the following problem:

```
(define (bar a b)
  (bar-helper 0 a b))
(define (bar-helper c a b)
  (if (> a b)
    c
    (bar-helper (+ c a) (+ a 1) b)))
```

What’s the order of growth in space Θ(1) and time Θ(a) ?

Is it recursive or iterative? iterative

Write the other:

```
(define (bar-rec a b)
  (if (< a b) 0
    (* a (bar-rec (+ a 1) b)))
```

6
Cubic Roots

We are now going to create and analyze some methods of finding the zeros of a cubic equation, i.e. given \(a, b, c, \) and \(d\) find values of \(x\) for which \(ax^3 + bx^2 + cx + d = 0\).

Assume we’ve been supplied with two guesses for \(x\) and the coefficients. If either guess gives a solution that is close enough to zero, return the guess. If not, then you have lots of choices. One is to move each guess towards the other by a slight amount and continue, another is to split the domain in two and try both halves (e.g. if the guesses are \(g_1\) and \(g_2\), then try \(g_1\) and \((g_1 + g_2)/2\) and \((g_1 + g_2)/2\) and \(g_2\).

In class, we’ll now go through the process of designing the above two versions of \texttt{find-cubic-root}, discussing modularity, orders of growth, recursion vs. iteration, accuracy, and other fun concepts. Your instructor’s solution will be posted at \url{http://people.csail.mit.edu/dalleyg/6.001/SP2007/index.html}.

;;; \texttt{find-cubic-root} solutions...

;;;; Some constants (defined here so we give them a name and
;;;; so we have just one place to look at to change them).
(define eps 0.0001) ; How close is close enough (range)?
(define delta 0.00001) ; How far do we move each time (domain)?

;;;; Helper functions
(define (eval-cubic a b c d x)
  (+ (* a x x x) (* b x x) (* c x) d))

(define (close-enuf? g)
  (< (abs g) eps))

(define (sign x)
  (cond ((> x 0) 1)
        ((= x 0) 0)
        (else -1)))

;;;; Assumes \(g_1 < g_2\), there is some root between \(g_1 \& g_2\) that can
;;;; be found by searching using fixed-size steps of size delta.
(define (find-cubic-root a b c d g1 g2)
  (let ((y1 (eval-cubic a b c d g1))
        (y2 (eval-cubic a b c d g2)))
    (cond ((close-enuf? y1) g1)
          ((close-enuf? y2) g2)
          (else (find-cubic-root a b c d (+ g1 delta) (- g2 delta))))))

;; Extra questions:
;; Iterative or recursive? iterative (even though no helper)
;; Space order of growth? 1
;; Time OOG (assume root is closer to \(g_1\))? \(\Theta(n)\), where \(n=(\text{root}-g_1)\)

;;;; Assumes \(g_1 < g_2\) and there is at least one root between \(g_1 \& g_2\)
(define (find-cubic-root a b c d g1 g2)
  (let ((gmid (/ (+ g1 g2) 2)))
    (let ((y1 (eval-cubic a b c d g1))
          (ymid (eval-cubic a b c d gmid))
          (y2 (eval-cubic a b c d g2)))
      (cond ((close-enuf? y1) g1)
            ((close-enuf? y2) g2)
            (else (find-cubic-root a b c d (+ g1 delta) (- g2 delta)))))))
((close-enuf? y2) g2)
((>= g1 g2) #f)
((find-cubic-root a b c d g1 gmid) (find-cubic-root a b c d g1 gmid))
(else (find-cubic-root a b c d gmid g2))))

(find-cubic-root 1 0 0 0 -1 1)

;; Extra questions:
; Iterative or recursive? iterative
; Space OOG? 1
; Time OOG? Theta(log2(n)), where n=(root-g1) or n=(root-g2)

Challenge Problem

;; Challenge Problem:
;; a) Is this function iterative or recursive?
;; b) What is its order-of-growth in time? space?
;; c) What does this thing actually do (hint: 18.02)?
;; d) Rewrite as recursive/iterative (which ever this is not).
;; e) What is the order of growth for your new version in time? space?
(define (baz n)
  (define (qux a b c)
    (if (> a b)
        c
        (qux (+ a 1) b
            ((if (even? a) - +) c (/ (- (* a 2) 1)))))
    (qux 1 n 0)))

;; Answers:
;; a) It's iterative (notice that the recursive calls to qux
; are not embedded in any expression that requires deferred
; operations).
;; b) Time: Theta(n), space: Theta(1)
;; c) It computes π/4 (see the de-obfuscated version below).
;; This uses Leibniz et al.'s series. It converges very slowly.
;; d) See below for a recursive version
;; e) The version below is Theta(n) in time and space

;; De-obfuscated version
(define (pi/4 n)
  (define (helper i n answer)
    (if (> i n)
        answer
        (helper (+ i 1) n
            ((if (even? i) - +) answer (/ (- (* i 2) 1))))))
  (helper 1 n 0))

;; Recursive version
(define (pi/4-recursive n)
  (if (= n 1)
      1
      ((if (even? n) - +) (pi/4-recursive (- n 1)) (/ (- (* n 2) 1)))))

;; Automated test code
(define (check x expected)
  (if (not (equal? x expected))
      (error "Error: " x " not equal to " expected)))
(check (pi/4 1) 1)
(check (pi/4 2) (- 1 (/ 3)))
(check (pi/4 3) (+ (pi/4 2) (/ 5)))
(check (pi/4 4) (- (pi/4 3) (/ 7)))
(check (baz 1) 1)
(check (baz 2) (- 1 (/ 3)))
(check (baz 3) (+ (pi/4 2) (/ 5)))
(check (baz 4) (- (pi/4 3) (/ 7)))
(check (pi/4-recursive 1) (pi/4 1))
(check (pi/4-recursive 2) (pi/4 2))
(check (pi/4-recursive 3) (pi/4 3))
(check (pi/4-recursive 100) (pi/4 100))
(check (pi/4-recursive 101) (pi/4 101))