You're Invited!
Course VI Freshman Open House!
Friday, April 7, 2006
3:30-5:00 PM
34-401
FREE Course VI T-Shirts (while supplies last) and Department Memorabilia
Faculty Research Presentations & Robot Competition Demonstrations
LOTS of Food!
Talk to faculty and staff about Course VI
New Flyers and brochures about Course VI
ALL freshmen are invited, especially those who have decided to or are thinking about majoring in Course VI!

Algorithms & Data Structures

- Lists
  - Append vs. append!, reverse vs. reverse!, folding, ...
  - List accessors: list-ref, list-tail, list-head, ...
  - Sort & merge
- Trees
  - ADT for trees
  - Tree-fold, subst
- Compression via Huffman coding

Lists: Constructors, Selectors, Operations

- Basics of construction, selection
  - cons, list, list-ref, list-head, list-tail
- Operations
  - Combining: reverse, append
  - Process elements: map, filter, right-fold, left-fold, sort
- Abstraction: ... just use Scheme's

Selectors: Beyond car, cdr

> (define ex '(a b c d e f))
> (list-ref ex 3)
d
> (define (list-ref lst n)
  (if (zero? n)
    (car lst)
    (list-ref (cdr lst) (- n 1))))

> (list-tail ex 3)
(d e f)
> (list-tail ex 0)
(a b c d e f)
> (list-head ex 3)
(a b c)

Selectors: Beyond car, cdr

> (define (list-tail lst n)
  (cond ((zero? n) lst)
        ((null? lst)
         (error "Cannot take list-tail" (list n lst)))
        (else (list-tail (cdr lst) (- n 1))))

> (list-tail ex 3)
d
> (list-tail ex 0)
a b c d e f
Selectors: Beyond car, cdr

> (list-head ex 3)
(a b c)

(define (list-head lst n)
  (if (or (null? lst) (zero? n))
    '()
    (cons (car lst)
          (list-head
           (cdr lst) (- n 1))))))

List-head!

> (define ex '(a b c d e f))
> (list-head! ex 0)
()
> ex
(a b c d e f)
> (list-head! ex 2)
(a b)
> ex
(a b)

Destructive list-head!

(define (list-head! lst n)
  (let ((lroot (cons '() lst)))
    (define (iter l i)
      (if (zero? i)
          (set-cdr! l '())
          (iter (cdr l) (- i 1)))))
    (iter lroot n)
    (cdr lroot)))

Append

(define (append a b)
  (if (null? a)
      b
      (cons (car a)
            (append (cdr a) b))))

T(n) = Θ(n)
S(n) = Θ(1)

Append!

> (define a '(1 2))
> (define b '(3 4))
> (append! a b)
(1 2 3 4)
> a
(1 2 3 4)
> b
(3 4)

Append!

(define (append! a b)
  (define (iter l)
    (if (null? (cdr l))
        (set-cdr! l b)
        (iter (cdr l)))))
  (cond ((null? a) b)
        (else (iter a)))))

T(n) = Θ(n)
S(n) = Θ(1)
Reverse

\(\text{Reverse}()\)

\[
\text{reverse0}\(\text{lst}\) = \begin{cases} 
\text{null? lst} & \text{if}\ 0 \\
\text{append}(\text{reverse0}(\text{cdr}\ \text{lst})) & \text{else} \\
(\text{car}\ \text{lst}) & \text{if}\ 1 
\end{cases}
\]

Substitution model:

\[
\text{reverse0}\('1 2 3')\]

\[
\text{append}\(\text{reverse0}(\text{'2 3')&&(\text{list}\ '1')})\]

\[
\text{append}\(\text{append}(\text{append}(\text{reverse0}(\text{'1')&&(\text{list}\ '3'))))\text{'}(\text{list}\ '2'))\]

\[
(\text{list}\ '1'))\]

\(T(n) = \Theta(n^2)\)

\(S(n) = \Theta(n)\)

Reverse (better)

\(\text{reverse}\()\)

\[
\text{reverse}\(\text{lst}\) = \begin{cases} 
\text{null? lst} & \text{if}\ 0 \\
\text{append}(\text{reverse}\(\text{cdr}\ \text{lst})) & \text{else} \\
(\text{car}\ \text{lst}) & \text{if}\ 1 
\end{cases}
\]

\(T(n) = \Theta(n)\)

\(S(n) = \Theta(1)\)

Reverse!

\(\text{reverse!}\()\)

\[
\text{reverse!}\(\text{lst}\) = \begin{cases} 
\text{null? current} & \text{if}\ 0 \\
\text{set-cdr! current last} & \text{else} \\
(\text{car}\ current) & \text{if}\ 1 
\end{cases}
\]

\(T(n) = \Theta(n)\)

\(S(n) = \Theta(1)\)

Two map’s & filter

\(\text{map}\()\)

\[
\text{map}\(\text{f}\ \text{lst}\) = \begin{cases} 
\text{null? lst} & \text{if}\ 0 \\
\text{cons}(\text{f}\ (\text{car}\ \text{lst})) & \text{else} \\
(\text{map0}\ f\ (\text{cdr}\ \text{lst})) & \text{if}\ 1 
\end{cases}
\]

\(T(n) = \Theta(n)\)

\(S(n) = \Theta(n)\)

\(\text{map!}\()\)

\[
\text{map!}\(\text{f}\ \text{lst}\) = \begin{cases} 
\text{null? lst} & \text{if}\ 0 \\
\text{set-car! current f}\ (\text{car}\ current) & \text{else} \\
(\text{map}\ f\ (\text{cdr}\ \text{lst})) & \text{if}\ 1 
\end{cases}
\]

\(T(n) = \Theta(n)\)

\(S(n) = \Theta(n)\)

\(\text{filter}\()\)

\[
\text{filter}\(\text{f}\ \text{lst}\) = \begin{cases} 
\text{null? lst} & \text{if}\ 0 \\
\text{null? (\text{cdr}\ \text{lst})} & \text{else} \\
\text{cons}(\text{f}\ (\text{car}\ \text{lst})) & \text{else} \\
(\text{filter}\ f\ (\text{cdr}\ \text{lst})) & \text{else} 
\end{cases}
\]

\(T(n) = \Theta(n)\)

\(S(n) = \Theta(n)\)
Fold Operations

(define (fold-right0 fn init lst)
  (if (null? lst)
      init
      (fn (car lst)
        (fold-right0 fn init (cdr lst)))))

;; T(n)=O(n), S(n)=O(n)

(define (fold-left fn init lst)
  (define (iter l ans)
    (if (null? l)
        ans
        (iter (cdr l) (fn ans (car l))))
  (iter lst init))

;; T(n)=O(n), S(n)=O(1)

(define (fold-right fn init lst)
  (fold-left (lambda (x y) (fn y x)) init (reverse lst)))

;; T(n)=O(n), S(n)=O(1)

Sorting a list

1. Split in half
2. Sort each half
3. Merge the halves
   - Merge two sorted lists into one
   - Take advantage of the fact they are sorted

(4 1 7 9 4 2 11 5)
(4 1 7 9) (4 2 11 5)
(1 4 7 9) (2 4 5 11)
(1 2 4 4 5 7 9 11)

Merge

(define (merge x y less?)
  (cond ((and (null? x) (null? y)) '())
        ((null? x) y)
        ((null? y) x)
        ((less? (car x) (car y))
          (cons (car x) (merge (cdr x) y less?))
        (else (cons (car y) (merge x (cdr y) less?)))))

> (merge '(1 4 7 9) '(2 4 5 11) <)
(1 2 4 4 5 7 9 11)

> (merge '(4 1 7 9) '(5 2 11 4) <)
(4 1 5 2 7 9 11 4)

Of course, there is merge!

(define (merge! x y less?)
  (let ((xroot (cons '() x))
        (yroot (cons '() y)))
    (define (iter ans)
      (cond ((and (null? (cdr xroot)) (null? (cdr yroot)))
              ans)
            ((null? (cdr xroot))
              (append! (reverse! (cdr yroot)) ans))
            ((null? (cdr yroot))
              (append! (reverse! (cdr xroot)) ans))
            ((less? (cadr xroot) (cadr yroot))
              (let ((current (cdr xroot)))
                (set-cdr! xroot (cdr current))
                (set-cdr! current ans)
                (iter current)))
            (else
              (let ((current (cdr yroot)))
                (set-cdr! yroot (cdr current))
                (set-cdr! current ans)
                (iter current))))))

Sort

(define (sort lst less?)
  (cond ((null? lst) '())
        ((null? (cdr lst)) lst)
        (else (let ((halves (halve lst)))
                (merge (sort (car halves) less?)
                        (sort (cdr halves) less?)
                        less?)))))

> (sort '(4 1 5 2 7 9 11 4) <)
(1 2 4 4 5 7 9 11)

X GIGO

GIGO
... and sort! and halve!

\[
\text{(define (sort! lst less?)}
\]
\[
\text{(cond ((null? lst) '())}
\]
\[
\text{((null? (cdr lst)) lst)}
\]
\[
\text{((else (let ([halves (halve! lst)])}
\]
\[
\text{(merge! (sort! (car halves) less?)}
\]
\[
\text{(sort! (cdr halves) less?) less?)))}})
\]
\[
\text{(define (halve! lst)}
\]
\[
\text{(cond ((null? lst) (error "Can't halve" lst))}
\]
\[
\text{((null? (cdr lst)) (cons lst '()))}
\]
\[
\text{((else}
\]
\[
\text{[let* ([mid (list-tail lst (- (quotient (length lst) 2)
\]
\[
\text{1)]])}
\]
\[
\text{(2nd-half (cdr mid)))}
\]
\[
\text{(set-cdr! mid '())}
\]
\[
\text{(cons lst 2nd-half)))}})
\]

Sort of final word on sort

- Finding midpoint of list is expensive, and we keep having to do it
- Instead, nibble away from left
  - Pick off first two sublists of length 1 each
  - Merge them to get a sorted list of length 2
  - Pick off another sublist of length 2, sort it, then merge with previous \(\Rightarrow\) length 4
  - ...
  - Pick off another sublist of length \(2^n\), sort, then merge with prev \(\Rightarrow\) length \(2^{n+1}\)

Trees

- Abstract Data Type for trees
  - Tree\(<C> = \text{Leaf}\(<C> | \text{List}\(<\text{Tree}\(<C>)>)$
  - \text{Leaf}\(<C> = C$
  - Note: C had best \textit{not} be a list

\[
\text{(define (leaf? obj) (not (pair? obj))) ;; () can be a leaf}
\]
\[
\text{(define (leaf? obj) (not (list? obj))) ;; () is the empty tree}
\]

Counting leaves

\[
\text{(define (count-leaves tree)}
\]
\[
\text{(cond ((leaf? tree) 1)}
\]
\[
\text{(else (fold-left}
\]
\[
\text{+ 0}
\]
\[
\text{(map count-leaves tree)))}})
\]
\[
\text{(define tr (list 4 (list 5 7) 2))}
\]
\[
\text{(define tr2 (list 4 (list '() 7) 2))}
\]
\[
\text{> (count-leaves tr)}
\]
\[
\text{4}
\]
\[
\text{> (count-leaves tr2)}
\]
\[
\text{4}
\]

General operations on trees

\[
\text{(define (tree-map f tree)}
\]
\[
\text{(if (leaf? tree)}
\]
\[
\text{(f tree)}
\]
\[
\text{(map (lambda (e) (tree-map f e))
\]
\[
\text{tree)))}})
\]
\[
\text{> tr}
\]
\[
\text{(4 (5 7) 2)}
\]
\[
\text{(4 (5 7) 2)}
\]
\[
\text{(4 (5 7) 2)}
\]
\[
\text{(16 (25 49) 4)}
\]
Using tree-map and tree-fold

```
(define (tree-fold leaf-op combiner init tree)
  (if (leaf? tree)
      (leaf-op tree)
      (fold-right combiner init
               (map (lambda (e) (tree-fold leaf-op combiner init e))
                    tree))))
```

```
> (tree-fold (lambda (x) 1) + 0 tr)
4
```

Huffman Coding

- If some symbols in an alphabet are more frequently used than others, we can compress messages
- ASCII uses 7 or 8 bits/char (128 or 256)
- In English, “e” is far more common than “z”, which in turn is far more common than Ctl-K (vertical tab(?)
- Huffman: use shorter bit-strings to encode most common characters
  - Prefix codes: no two codes share same prefix

Making a Huffman Code

- Start with a list of symbol/frequency nodes, sorted in order of increasing freq
- Merge the first two into a new node. It will represent the union of the symbols and sum of frequencies; sort it back into the list
- Repeat until there is only one node

Example of building a Huffman Tree

```
(H 1) (G 1) (F 1) (E 1) (D 1) (C 1) (B 3) (A 8)
(F 1) (E 1) (D 1) (C 1) (G 2) (B 3) (A 8)
(D 1) (F 2) (E 2) (H 2) (B 3) (A 8)
(H G 2) (B 3) (D C F E 4) (A 8)
(D C F E 4) (H G B 5) (A 8)
(A B) (D C F E H G B 9)
((A D C F E H G B) 17)
```

AHA = 0 1 1 0 0 0

Leaf holds symbol & weight

```
(define (make-leaf symbol weight)
  (list 'leaf symbol weight))
(define (leaf? obj)
  (and (pair? obj)
       (eq? (car obj) 'leaf)))
(define symbol-leaf cadr)
(define weight-leaf caddr)
```
Our training sample

(define text1 "The algorithm for generating a Huffman tree is very simple. The idea is to arrange the tree so that the symbols with the lowest frequency appear farthest away from the root. Begin with the set of leaf nodes, containing symbols and their frequencies, as determined by the initial data from which the code is to be constructed. Now find two leaves with the lowest weights and merge them to produce a node that has these two nodes as its left and right branches. The weight of the new node is the sum of the two weights. Remove the two leaves from the original set and replace them by this new node. Now continue this process. At each step, merge two nodes with the smallest weights, removing them from the set and replacing them with a node that has these two as its left and right branches. The process stops when there is only one node left, which is the root of the entire tree."")

Statistics

((leaf |H| 1) (leaf |B| 1) (leaf |R| 1) (leaf |A| 1) (leaf q 2) (leaf |N| 2) (leaf |T| 4) (leaf v 5) (leaf |,| 5) (leaf u 7) (leaf b 7) (leaf y 8) (leaf |.| 9) (leaf p 10) (leaf g 17) (leaf c 17) (leaf l 19) (leaf f 19) (leaf m 20) (leaf i 43) (leaf h 57) (leaf t 84) (leaf a 109) (leaf | | 170))

How efficient?

• Our sample text has 887 characters, or 7096 bits in ASCII.
• Our generated Huffman code encodes it in 3648 bits, ≈51% (4.1 bits/char)
• Because code is built from this very text, it’s as good as it gets!
• LZW (Lempel-Zip-Welch) is most common, gets ≈50% on English.
Summary

• Lists: standard and mutating operators...
• Sort & merge
• Trees
• Compression via Huffman coding
• The organization of the code reflects the organization of the data it operates on.