CS3: Introduction to Symbolic Programming

Lecture 10: Finishing HOF
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Mar 19-23</td>
<td>Lecture: Higher-order function review</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lab: More Higher order functions</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Miniproject #3 assigned</strong></td>
</tr>
<tr>
<td></td>
<td>Mar 26-30</td>
<td><em>Spring Break</em></td>
</tr>
<tr>
<td>11</td>
<td>April 2</td>
<td>Lecture: Midterm review, tree recursion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lab: Lists, tree-recursion</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Miniproject #3 due Tuesday</strong></td>
</tr>
<tr>
<td>12</td>
<td>April 9-13</td>
<td>Lecture (5-7 pm): <em>Midterm #2</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td>145 Dwinelle</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lab: Advanced list processing</td>
</tr>
<tr>
<td>13</td>
<td>April 16-20</td>
<td>Lecture: CS3 Projects</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lab: Begin work on CS3 Big Project</td>
</tr>
</tbody>
</table>
Lab materials

• Last week:
  - day-span using higher order procedures
  - tic-tac-toe

• This week:
  - A half day working further on tic-tac-toe (T/W)
  - Some "Challenging review problems", with solutions (T/W)
  - Work on the miniproject (all week)
• You can mimic 2-stage recursion, applying a function to each letter of each word.

• You can get combinatoric effects:

```
(define (pair-all sent)
  (every (lambda (one)
            (every (lambda (two)
                    (word one two)
                    sent))
              sent))

(pair-all '(a b c)) ➔ ???
```
(make-kw '(s t) '(a o)) ➞
(sas sat sos sot tas tat tos tot)
(make-kw '(l n k t s) '(a e i o u)) ➞ 225 words!

(define (make-kw consonants vowels)
  (every (lambda (c)
    (every (lambda (v)
      (every (lambda (v) vowels))
    consonants))
  vowels))
accumulate can return a sentence...

- the *first* time accumulate is run, it reads the last two words of the input sentence, and returns a sentence

- in *later* calls, it uses the return value of its procedure (which is a sentence) as its 2\textsuperscript{nd} argument, and the next work as its 1\textsuperscript{st}.

Write \texttt{pair-conseq}:

\texttt{(pair-conseq ' (a b c d)) ⇒ (ab bc cd)}
lambda
the lambda form

• "lambda" is a special form that returns a function:

```
(lambda (arg1 arg2 ...)  
    statements
  )
```

```
(l lambda (x) (* x x))
```

⇒  ⇒  ⇒  ⇒  ⇒

a procedure that takes one argument and multiplies it by itself
Use lambda anywhere you need a function

\[
\begin{align*}
\text{(define square} & \quad (\text{lambda} \ (x) \ (* \ x \ x)) \\
\text{(every } & \quad (\text{lambda} \ (x) \ (* \ x \ x)) \\
& \quad (1 \ 2 \ 3)) \\
& \Rightarrow (1 \ 4 \ 9) \\
\text{(}(\text{lambda} \ (x) \ (* \ x \ x)) & \quad 3) \\
& \Rightarrow 9
\end{align*}
\]
Write make-bookends, which is used this way:

(((make-bookends 'o) 'hi) ➜ ohio
((make-bookends 'to) 'ron) ➜ toronto
(define tom-proc (make-bookends 'tom))
(tom-proc "") ➜ tomtom
You *need* lambda when...

...you need a procedure to make reference to more values than you can pass it.

For instance, when a procedure for use in an `every` needs two parameters

```
(prepend-every 'sir- '(sam mary loin))
⇒ (sir-sam sir-mary sir-loin)
```

Write `prepend-every`

Write `appearances`
Problems
Write successive-concatenation

(sc ' (a b c d e))
⇒ (a ab abc abcd abcde)

(sc ' (the big red barn))
⇒ (the thebig thebigred thebigredbarn)

(define (sc sent)
  (accumulate
   (lambda ??
   )
    sent))
make-decreasing

- Takes a sentence of numbers
- Returns a sentence of numbers, having removed elements of the input that were not larger than all numbers to the right of them.

(make-decreasing '(9 6 7 4 6 2 3 1))
⇒ (9 7 6 3 1)

(make-decreasing '(3))
⇒ (3)

Write first as a recursion, then as a HOF
## Schedule

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 10  | Mar 19-23 | Lecture: Higher-order function review  
Lab: More Higher order functions  
Miniproject #3 assigned |
|     | Mar 26-30 | Spring Break                                                  |
| 11  | April 2  | Lecture: Midterm review, tree recursion  
Lab: Lists, tree-recursion  
Miniproject #3 due Tuesday |
| 12  | April 9-13 | Lecture (5-7 pm): Midterm #2  
145 Dwinelle  
Lab: Advanced list processing |
| 13  | April 16-20 | Lecture: CS3 Projects  
Lab: Begin work on CS3 Big Project |
Lab materials

• Last week:
  - day-span using higher order procedures
  - tic-tac-toe

• This week:
  - A half day working further on tic-tac-toe (T/W)
  - Some "Challenging review problems", with solutions (T/W)
  - Work on the miniproject (all week)
• You can mimic 2-stage recursion, applying a function to each letter of each word.

• You can get combinatoric effects:

```
(define (pair-all sent)
  (every (lambda (one)
    (every (lambda (two)
      (word one two))
      sent))
    sent))

(pair-all '(a b c)) ⇒ ???
```
every containing every containing...

(make-kw '(s t) '(a o)) ➞
(sas sat sos sot tas tat tos tot)
(make-kw '(l n k t s) '(a e i o u)) ➞ 225 words!

(define (make-kw consonants vowels)
  (every (lambda (c)
    (every (lambda (v)
      vowels))
    consonants))

(define (make-kw consonants vowels)
  (every (lambda (c)
    (every (lambda (v)
      (every (lambda (c2)
        (word c v c2))
      consonants))
    vowels))
  consonants))
accumulate can return a sentence...

- the first time accumulate is run, it reads the last two words of the input sentence, and returns a sentence

- in later calls, it uses the return value of its procedure (which is a sentence) as its 2nd argument, and the next work as its 1st.

Write pair-conseq:

\[(\text{pair-conseq } '(a\ b\ c\ d)) \rightarrow (ab\ bc\ cd)\]
lambda

Click to add text
the lambda form

• "lambda" is a special form that returns a function:

```
(lambda (arg1 arg2 ...) statements)
```

```
(lambda (x) (* x x))
```

goes to

```
(lambda (x) (* x x))
```

a procedure that takes one argument and multiplies it by itself
Use lambda anywhere you need a function

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

(every (lambda (x) (* x x))
  '(1 2 3))
⇒ (1 4 9)

(((lambda (x) (* x x)) 3)
  ⇒ 9)
```
**make-bookends (a small problem)**

- **Write** `make-bookends`, which is used this way:

  ```scheme
  ((make-bookends 'o) 'hi) \rightarrow ohio
  ((make-bookends 'to) 'ron) \rightarrow toronto
  (define tom-proc (make-bookends 'tom))
  (tom-proc "") \rightarrow tomtom
  ```

  ```scheme
  (define (make-bookends wd)
    (lambda (inner-wd) (word wd inner-wd wd)))
  ```
You *need* lambda when…

…you need a procedure to make reference to more values than you can pass it.

For instance, when a procedure for use in an `every` needs two parameters

```lisp
(prepend-every 'sir- '(sam mary loin))
⇒ (sir-sam sir-mary sir-loin)
```

Write `prepend-every`

Write `appearances`
Problems

Click to add text
Write successive-concatenation

(sc '(a b c d e))
⇒ (a ab abc abcd abcde)

(sc '(the big red barn))
⇒ (the thebig thebigred thebigredbarn)

(define (sc sent)
  (accumulate
    (lambda (wd sent-so-far)
      ;; initial invocation
      (if (word? sent-so-far)
          (se wd (word wd sent-so-far)) ;; prepend-each
          ;; other invocations
          (se wd
               ;; prepend-each
               (every
                (lambda (sent-so-far-element)
                  (word wd sent-so-far-element))
                sent-so-far)))
    )
  )
  sent))
make-decreasing

• make-decreasing
  - Takes a sentence of numbers
  - Returns a sentence of numbers, having removed elements of the input that were not larger than all numbers to the right of them.

(make-decreasing '(9 6 7 4 6 2 3 1))
  ⇒ (9 7 6 3 1)
(make-decreasing '(3))  ⇒ (3)

Write first as a recursion, then as a HOF

;; recursion -- left to right
(define (make-decreasing sent)
  (cond ((or (empty? sent)
               (empty? (bf sent)))
         sent)
        ((bigger-than-all? (first sent) (bf sent))
         (se (first sent)
             (make-decreasing (bf sent))))
        (else (make-decreasing (bf sent))))
)

(define (bigger-than-all? num sent)
  (cond ((empty? sent) #t)
        ((> num (first sent))
         (bigger-than-all? num (bf sent))
         (else #f))
        ))

;; HOF
(define (make-decreasing sent)
  (accumulate
   (lambda (left right)
     (if (word? right)
         (if (< right left)
             (se left right)
             right)
         (if (< (first right) left)
             (se left right)
             right)))
   sent))