CS3: Introduction to Symbolic Programming

Lecture 9: More higher-order functions, \texttt{lambda}, tic-tac-toe

Spring 2007  Nate Titterton  nate@berkeley.edu
<table>
<thead>
<tr>
<th>Week</th>
<th>Date</th>
<th>Lecture:</th>
<th>Lab:</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Mar 12-16</td>
<td>Higher order functions, lambda</td>
<td>Higher order functions,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reading: Simply Scheme, Ch 9, 10</td>
<td>tic-tac-toe</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;DbD&quot; HOF version</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lab: Higher order functions, tic-tac-toe</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Mar 19-23</td>
<td>Higher-order function review</td>
<td>More Higher order functions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lab:</td>
<td>Miniproject #3 assigned</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Mar 26-30 Spring Break</td>
</tr>
<tr>
<td>11</td>
<td>April 2</td>
<td>Midterm review, tree recursion</td>
<td>Lists, tree-recursion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lab:</td>
<td>Miniproject #3 due Tuesday</td>
</tr>
<tr>
<td>12</td>
<td>April 9</td>
<td>Midterm #2</td>
<td></td>
</tr>
</tbody>
</table>
Tic Tac Toe
The board

"X _ _"

"O O X"

"_ _ _"

"X _ _ O O X _ _ _"
Triples (another representation of a board)

"X__O O X__"

( x23 oox 789 xo7 2o8 3x9 xo9 3o7 )
Tic-tac-toe hints

• Read the chapter!
• You will need to be familiar with vocabulary
  - positions, triples, "forks", "pivots", and so on
• This chapter in the book comes before recursion.
  - You would solve things differently if you used recursion
• The code (at the end of the chapter) has no comments.
Higher-order functions: review
Higher order function (HOFs)

• A HOF is a procedure that takes a procedure as an argument.
• There are three main ones that work with words and sentences:

  - **every**
    - take a one-argument procedure that returns a word
    - do something to each element
  - **keep**
    - takes a one-argument predicate
    - return only certain elements
  - **accumulate**
    - takes a two-argument procedure
    - combine the elements
A definition of every

\[
\text{(define (my-every proc ws)}
  \text{(if (empty? ws)}
    \text{'}()
    \text{(se (proc (first ws))}
    \text{(my-every (bf ws))}
  \text{))}
\]

- HOFs do a lot of work for you:
  - Checking the conditional
  - Returning the proper base case
  - Combing the various recursive steps
  - Invoking itself recursively on a smaller problem
• The *direction* matters: right to left
  - `(accumulate / '(4 2 2))
    does not equal 1, but 4.

• Think about expanding an accumulate

  `(accumulate + '(1 2 3 4))
  ➞ (+ 1 (+ 2 (+ 3 4)))

  `(accumulate / '(4 2 2))
  ➞ (/ 4 (/ 2 2))
accumulate can return a sentence...

(accumulate ?? '(a b c d))

⇒ (ab bc cd)

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

- in *later* calls, it uses the return value of its procedure (which is a sentence) as one of its arguments
Any questions from last week?

• You wrote and played with every, keep, and accumulate
• You used them in combination:

\[(\text{gpa ' (A A F C B))} \rightarrow 2.6 \quad \text{(average of 4, 4, 0, 2, 3)}\]

\[(\text{gpa-with-p/no '(A A F NP P C B))} \rightarrow 2.6 \quad \text{(average of 4, 4, 0, 2, 3)}\]

\[(\text{true-for-all? even? '(2 4 6 8))} \rightarrow \#t\]
Which HOFs would you use? (1/2)

1) **capitalize-proper-names**
   
   `(c-p-n '(mr. smith goes to washington))`  
   => `(mr. Smith goes to Washington)`

3) **count-if**
   
   `(count-if odd? '(1 2 3 4 5))`  
   => 3

5) **longest-word**
   
   `(longest-word '(I had fun on spring break))`  
   => spring

7) **count-vowels-in-each**
   
   `(c-e-l '(I have forgotten everything))`  
   => (1 2 3 3)
1) squares-greater-than-100
   \((s\text{-}g\text{-}t\text{-}100\ (2\ 9\ 13\ 16\ 9\ 45))\)
   \(\Rightarrow\ (169\ 256\ 2025)\)

3) root of the sum-of-squares
   \((sos\ (1\ 2\ 3\ 4\ 5\ 6\ 7))\)
   \(\Rightarrow\ (sqrt\ (+\ (*\ 1\ 1)\ (*\ 2\ 2)\ ...))\)
   \(\Rightarrow\ 30\)

5) successive-concatenation
   \((sc\ (a\ b\ c\ d\ e))\)
   \(\Rightarrow\ (a\ ab\ abc\ abcd\ abcde)\)
defining variables, let, and lambda
Three ways to define a variable

1. In a procedure call (e.g., the variable proc):
   
   (define (doit proc value)
       ;; proc is a procedure here...
       (proc value))

3. As a global variable
   
   (define *alphabet* '(a b c d e ... ))
   (define *month-name* '(january ... ))

5. With let
Using `let` to define temporary variables

- `let` lets you define variables **within** a procedure:

  ```scheme
  (define (scramble-523 wd)
      (let ((second (first (bf wd)))
                (third (first (bf (bf wd))))
                (fifth (item 5 wd))
           )
       (word fifth second third) ) )
  
  (scramble-523 'meaty) \rightarrow yea
  ```
Any differences?

```
(define pi 3.14159265)
(define (alpha beta pi zeta)
    ... lots of code here ...
    (* pi radius)
    ... more code here ...
)
```

YES!

```
(define (alpha beta pi zeta)
    (let ((pi 3.14159265))
        ... lots of code here ...
        (* pi radius)
        ... more code here ...
    )
)```
In Scheme, procedures are *first-class* objects

- You can assign them a name
- You can pass them as arguments to procedures
- You can return them as the result of procedures
- You can include them in data structures

1. Well, you don't know how to do all of these yet.

3. What else in scheme is a *first-class* object?
The "hard" one is #3: returning procedures

;; this returns a procedure
(define (make-add-to number)
    (lambda (x) (+ number x)))

;; this also returns a procedure
(define add-to-5 (make-add-to 5))

;; hey, where is the 5 kept!?
(add-to-5 8) ➞ 13

((make-add-to 3) 20) ➞ 23
"lambda" is a special form that returns a function:

\[
\text{(lambda (arg1 arg2 \ldots) }
\text{statements)}
\]

\[
\text{(lambda (x) (* x x))}
\]

\[
\Rightarrow \Rightarrow \Rightarrow \Rightarrow \Rightarrow
\]

a procedure that takes one argument and multiplies it by itself.
Using lambda with define

• These are the same:

```plaintext
(define (square x)
    (* x x))

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

- These are VERY DIFFERENT:

```scheme
(define (adder-1 y)
  (lambda (x) (+ x 1)))
```

```scheme
(define adder-2
  (lambda (x) (+ x 1)))
```
Can a lambda-defined function be recursive?

(lambda (sent)
  (if (empty? sent)
   '()
   (se (square (first sent))
     (???? (bf sent))))))
When do you NEED lambda?

1. When you need the context (inside a two-parameter procedure)

```
(add-suffix '-is-great '(haddad sam mary))
⇒ (haddad-is-great sam-is-great mary-is-great)
```

3. When you need to make a function on the fly
Problems
(hangman-status 'joebob 'abcde)
→ __eb_b

(define (hangman-status secret-wd ltrs)
  ???
)

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))
## Schedule

<table>
<thead>
<tr>
<th>Week</th>
<th>Dates</th>
<th>Lecture</th>
<th>Lab</th>
<th>Miniproject #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Mar 12-16</td>
<td>Higher order functions, lambda</td>
<td>Higher order functions, tic-tac-toe</td>
<td>Assigned</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reading: Simply Scheme, Ch 9, 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;DbD&quot; HOF version</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Mar 19-23</td>
<td>Higher-order function review</td>
<td>More Higher order functions, tic-tac-toe</td>
<td>Assigned</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reading: Simply Scheme, Ch 9, 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;DbD&quot; HOF version</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lab: Higher order functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mar 26-30</td>
<td>Spring Break</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>April 2</td>
<td>Midterm review, tree recursion</td>
<td>Lists, tree-recursion</td>
<td>Due Tuesday</td>
</tr>
<tr>
<td>12</td>
<td>April 9</td>
<td>Midterm #2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tic Tac Toe

Click to add text
The board
**Triples** (another representation of a board)

```
X |   |
---+---+---
O | O | X
---+---+---
|   |
```

"X__ O O X__"

```
(find-triples 'x__oox___) ➜ (x23 oox 789 xo7 2o8 3x9 xo9 3o7)
```

```
(find-triples 'x__oox___) ➜ (x23 oox 789 xo7 "2o8" "3x9" xo9 "3o7")
```
• Read the chapter!
• You will need to be familiar with vocabulary
  - positions, triples, "forks", "pivots", and so on
• This chapter in the book comes *before* recursion.
  - You would solve things differently if you used recursion
• The code (at the end of the chapter) has no comments.
Higher-order functions: review

Click to add text
Higher order function (HOFs)

- A HOF is a procedure that takes a procedure as an argument.
- There are three main ones that work with words and sentences:
  - **every**
    - take a one-argument procedure that returns a word
    - do something to each element
  - **keep**
    - takes a one-argument predicate
    - return only certain elements
  - **accumulate**
    - takes a two-argument procedure
    - combine the elements
A definition of every

(define (my-every proc ws)
  (if (empty? ws)
      '()
      (se (proc (first ws))
          (my-every (bf ws))
      )))

• HOFs do a lot of work for you:
  - Checking the conditional
  - Returning the proper base case
  • Combing the various recursive steps
  • Invoking itself recursively on a smaller problem
Accumulate (1/2)

• The *direction* matters: right to left
  - `(accumulate / '(4 2 2))`
    does not equal 1, but 4.

• Think about expanding an accumulate

```
(accumulate + '(1 2 3 4))
  ➞ (+ 1 (+ 2 (+ 3 4)))

(accumulate / '(4 2 2))
  ➞ (/ 4 (/ 2 2))
```
accumulate (2/2)

*accumulate can return a sentence...*

\[
(\text{accumulate } ?? ' (a b c d))
\rightarrow (ab \ bc \ cd)
\]

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

- in later calls, it uses the return value of its procedure (which is a sentence) as one of its arguments

\[
(\text{define (concat-pairs sent)})
\]

\[
(\text{accumulate (lambda (wd so-far))})
\]

\[
(\text{if (word? so-far})
\]

\[
(\text{se (word wd so-far))
\]

\[
(\text{se (word wd (first (first so-far)))) so-far))
\]

\[
)
\]

sent))
Any questions from last week?

• You wrote and played with every, keep, and accumulate
• You used them in combination:

\[(\text{gpa} \ (A\ A\ F\ C\ B))\]
\[\Rightarrow\ 2.6\ \ (\text{average\ of\ } 4,\ 4,\ 0,\ 2,\ 3)\]

\[(\text{gpa-with-p/np} \ (A\ A\ F\ NP\ P\ C\ B))\]
\[\Rightarrow\ 2.6\ \ (\text{average\ of\ } 4,\ 4,\ 0,\ 2,\ 3)\]

\[(\text{true-for-all?\ even?} \ (2\ 4\ 6\ 8))\]
\[\Rightarrow\ \#t\]
Which HOFs would you use? (1/2)

1) capitalize-proper-names
   (c-p-n '(mr. smith goes to washington)) ➞ (mr. Smith goes to Washington)

3) count-if
   (count-if odd? '(1 2 3 4 5)) ➞ 3

5) longest-word
   (longest-word '(I had fun on spring break)) ➞ spring

7) count-vowels-in-each
   (c-e-l '(I have forgotten everything)) ➞ (1 2 3 3)

1) Every
2) Keep
3) Accumulate (longest-word needs to compare elements of the sentence; it can't consider each element in isolation)
4) Every containing a keep (count-if)
Which HOFs would you use? (2/2)

1) squares-greater-than-100
   
   \( (s-g-t-100 \ (2 \ 9 \ 13 \ 16 \ 9 \ 45)) \)
   \[ \rightarrow (169 \ 256 \ 2025) \]

3) root of the sum-of-squares
   
   \( (sos \ (1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7)) \)
   \[ \rightarrow (sqrt \ (+ \ (* \ 1 \ 1) \ (* \ 2 \ 2) \ ...)) \]
   \[ \rightarrow 30 \]

5) successive-concatenation
   
   \( (sc \ (a \ b \ c \ d \ e)) \)
   \[ \rightarrow (a \ ab \ abc \ abcd \ abcde) \]

1) Keep containing an every
2) Accumulate containing an every
3) Just accumulate. This isn't an every, although it looks like it at first glance, because you can't process the non-first elements without determining the elements that came before!
defining variables, \texttt{let}, and \texttt{lambda}

Click to add text
Three ways to define a variable

1. In a procedure call (e.g., the variable proc):
   (define (doit proc value)
       ;; proc is a procedure here...
       (proc value))

3. As a global variable
   (define *alphabet* '(a b c d e ... ))
   (define *month-name* '(january ... ))

5. With let
Using `let` to define temporary variables

- `let` lets you define variables within a procedure:

```
(define (scramble-523 wd)
  (let ((second (first (bf wd)))
      (third (first (bf (bf wd))))
      (fifth (item 5 wd))
    )
  (word fifth second third) ) )
```

```
(scramble-523 'meaty) ➜ yea
```
Any differences?

(define pi 3.14159265)
(define (alpha beta pi zeta)
    ... lots of code here ...
    (* pi radius)
    ... more code here ...

YES!

(define (alpha beta pi zeta)
(let ((pi 3.14159265)) )
    ... lots of code here ...
    (* pi radius)
    ... more code here ...
In Scheme, procedures are *first-class objects*

- You can assign them a name
- You can pass them as arguments to procedures
- You can return them as the result of procedures
- You can include them in data structures

1. Well, you don't know how to do all of these yet.

3. What else in scheme is a *first-class object*?

First-class objects (in scheme) can:
- Be named
- Be an parameter to functions
- Be returned from functions
- Be stored in other data structures
The "hard" one is #3: returning procedures

;; this returns a procedure
(define (make-add-to number)
  (lambda (x) (+ number x)))

;; this also returns a procedure
(define add-to-5 (make-add-to 5))

;; hey, where is the 5 kept!?
(add-to-5 8) ➔ 13

((make-add-to 3) 20) ➔ 23
the lambda form

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

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

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

⇒  ⇒  ⇒  ⇒  ⇒

a procedure that takes one argument and multiplies it by itself
Using lambda with define

• These are the same:

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

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

The top form is just a shortcut, really, for the bottom form. We would get tired having to type l-a-m-b-d-a all the time, so the above form is quicker.
Using lambda with define

• These are VERY DIFFERENT:

(define (adder-1 y)
    (lambda (x) (+ x 1)))

(define adder-2
    (lambda (x) (+ x 1)))

adder1 takes a single argument and returns a procedure (that takes a single argument
and returns 1 more than it)

adder2 takes a single argument and returns one more than it.
Can a lambda-defined function be recursive?

(lambda (sent)
  (if (empty? sent)
      '()
      (se (square (first sent))
          (???? (bf sent)))))

In cs3, nope.

But, you will find a way to make recursive lambda (non-named) functions if you continue in CS. (You might google for "anonymous recursion" in scheme' or something like that).
When do you NEED lambda?

1. When you need the context (inside a two-parameter procedure)

   \[
   \text{(add-suffix 'is-great '(haddad sam mary))} \\
   \Rightarrow \text{(haddad-is-great sam-is-great mary-is-great)}
   \]

3. When you need to make a function on the fly

   (define (add-suffix suf sent)
     (every
       (lambda (wd)
         (word wd suf)
       )
     )
   )
Problems

Click to add text
(hangman-status 'joebob 'abcde)
  \rightarrow __eb_b

(define (hangman-status secret-wd ltrs)
  ???
)

(define (hangman-status secret-wd ltrs)
  (accumulate
   word
   (every (lambda (ltr)
                (if (member? ltr ltrs)
                    ltr
                    '_))
          secret-wd)
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))

Email me for the solution if you want it before next lecture!