CS3:
Introduction to Symbolic Programming

Lecture 10:
Tic-tac-toe
Lambda

Fall 2006

Nate Titterton
nate@berkeley.edu

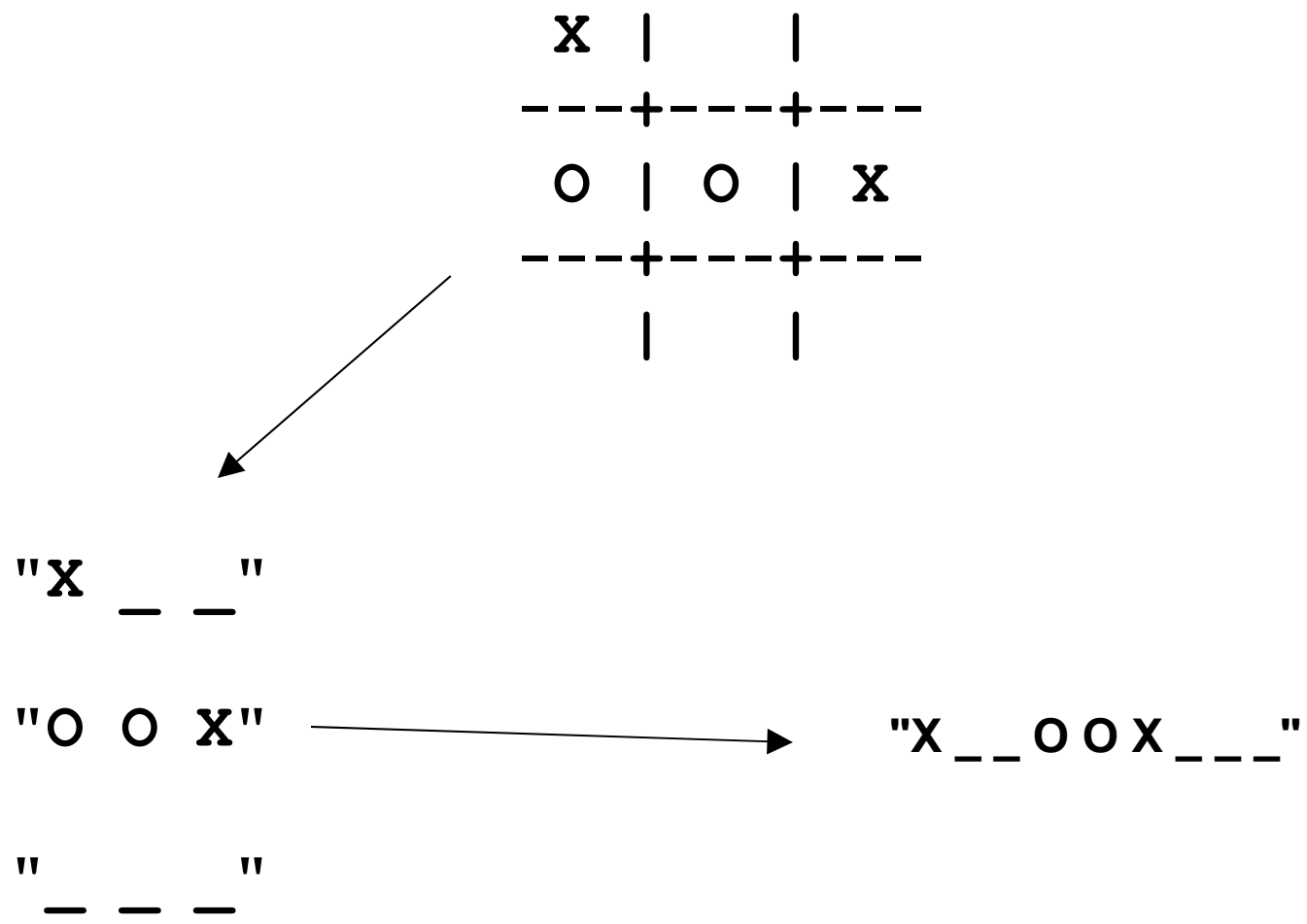
Schedule

9	Oct 23-27	Introduction to Higher Order Procedures Reading: Simply Scheme, Ch. 7-9 "Difference btw Dates" (HOF soln)
10	Oct 30 -Nov 3	More HOF, Tic-tac-toe Lab: Tic-tac-toe Starting Miniproject #3 Reading: Simply Scheme, Ch. 10
11	Nov 6-10	Tree-recursion, Review, Exam problems Lab: Tree recursions, Miniproject #3 Reading: "Change Making" case study Simply Scheme, Ch. 15
12	Nov 13-17	Lecture: *Midterm #2* Lab: Start on "Lists"

When should the MT2 review session be?

Tic Tac Toe

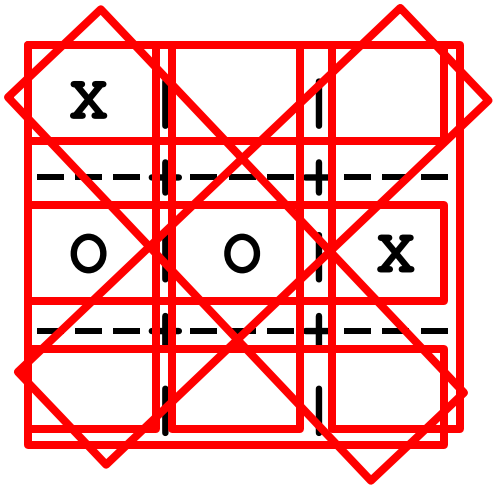
The board



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.**

Triples (another representation of a board)



"X__OOX___"

(x23 oox 789 xo7 2o8 3x9 xo9 3o7)

procedures and

`lambda`

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
```

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)))
```

Using `lambda` with `define`

- **These are VERY DIFFERENT:**

```
(define (adder1 y)
   (lambda (x) (+ x 1)))


(define adder2
   (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 '(nate sam mary))
    ➔ (nate-is-great sam-is-great
       mary-is-great)
```

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

Review

Higher order procedures

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 predictate
    - 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

- 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…
  - Here, the argument the *first* time accumulate is run (when it reads the last two words of the sentence) will be different from additional calls (when it uses the return value of its procedure, which is a sentence)

Which HOFs would you use to write these?

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)
```

9) **squares-greater-than-100**

```
(s-g-t-100 '(2 9 13 16 9 45) ➔ (169 256 2025)
```

11) **root of the sum-of-squares**

```
(sos '(1 2 3 4 5 6 7) ➔ 30
```

13) **successive-concatenation**

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

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))
```

Hangman-status

```
(hangman-status 'joebob 'abcde)
  → __eb_b
```

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

CS3:
Introduction to Symbolic Programming

Lecture 10:
Tic-tac-toe
Lambda

Fall 2006

Nate Titterton
nate@berkeley.edu

Schedule

9	Oct 23-27	Introduction to Higher Order Procedures Reading: Simply Scheme, Ch. 7-9 "Difference btw Dates" (HOF soln)
10	Oct 30 -Nov 3	More HOF, Tic-tac-toe Lab: Tic-tac-toe Starting Miniproject #3 Reading: Simply Scheme, Ch. 10
11	Nov 6-10	Tree-recursion, Review, Exam problems Lab: Tree recursions, Miniproject #3 Reading: "Change Making" case study Simply Scheme, Ch. 15
12	Nov 13-17	Lecture: *Midterm #2* Lab: Start on "Lists"

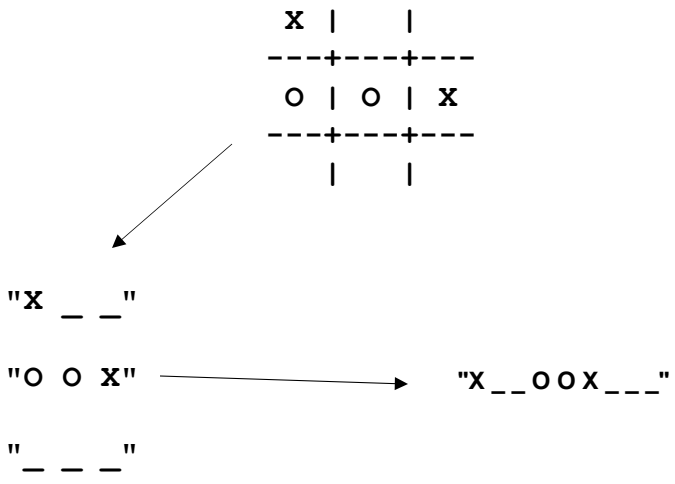
When should the MT2 review session be?

Click to add text

Tic Tac Toe

Click to add text

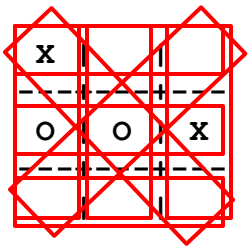
The board



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.

Triples (another representation of a board)



"X__OOX___"

(x23 oox 789 xo7 2o8 3x9 xo9 3o7)

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

procedures and

`lambda`

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 (adder1 y)
   (lambda (x) (+ x 1)))


(define adder2
   (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)**

```
(add-suffix '-is-great '(nate sam mary))
    ➔ (nate-is-great sam-is-great
       mary-is-great)
```

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

Review

Higher order procedures

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 predictate
    - 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

- 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…
  - Here, the argument the *first* time accumulate is run (when it reads the last two words of the sentence) will be different from additional calls (when it uses the return value of its procedure, which is a sentence)

Which HOFs would you use to write these?

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)
```

9) squares-greater-than-100

```
(s-g-t-100 '(2 9 13 16 9 45) ➔ (169 256 2025)
```

11) root of the sum-of-squares

```
(sos '(1 2 3 4 5 6 7) ➔ 30
```

13) successive-concatenation

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


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)
5) Keep containing an every
6) Accumulate containing an every
7) 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!

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!

Hangman-status

```
(hangman-status 'joebob 'abcde)
  → __eb_b
```

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


Some more practice with HOF…