CS3:
Introduction to Symbolic Programming

Lecture 9:
Higher Order Procedures

Fall 2006

Nate Titterton
nate@berkeley.edu

Schedule

8	Oct 16-20	Finishing recursion Miniproject #2: Number names
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, Tree Recursion *Reading*: Simply Scheme ch. 10, 15 "Change Making" case study
11	Nov 6-10	Finish HOF, Review, Exam problems Miniproject #3: Election processing *Note: Thursday is a "catch-up" day, and Friday a holiday.*
12	Nov 13-17	Lecture: *Midterm #2* Lab: Start on "Lists"

Announcements

- Surveys *really* coming this week and next
  - Take the time to do these, they are required.

What is a procedure?

(or, a *function*).

Treating functions as things

- "define" associates a name with a value
  - The usual form associates a name with a object that is a function

    ```
    (define (square x) (* x x))
    (define (pi) 3.1415926535)
    ```

  - You can define other objects, though:

    ```
    (define *pi* 3.1415926535)
    (define *month-names*
       `(january february march april may
         june july august september
         october november december))
    ```

"Global variables"

- Functions are "global", in that they can be used anywhere:

```
(define (pi) 3.1415926535)
(circle-area (radius)
      (* (pi) radius radius))
```

- A "global" variable, similarly, can be used anywhere:

```
(define *pi* 3.1415926535)
(circle-area (radius)
      (*  *pi*  radius radius))
```

Are these the same?

Consider two forms of “month-name”:

```
(define (month-name1 date)
    (first date))
```

```
(define month-name2 first)
```

Why have procedures as objects?

Other programming languages don’t (often)

Procedures can be taken as arguments…

```
(define (math-function? func)
  (or (equal? func +)
      (equal? func -)
      (equal? func *)
      (equal? func /)))
```

…and procedures can be returned from procedures

```
(define (choose-func name)
   (cond ((equal? name 'plus) +)
         ((equal? name 'minus) -)
         ((equal? name 'divide) /)
         (else 'sorry)))


(define (make-add-to number)
   (lambda (x) (+ number x)))

(define add-to-5 (make-add-to 5))
```

Higher order function (HOFs)

- A HOF is a function that takes a function as an argument.

```
(define (do-math f arg1 arg2)
    (if (and (equal? arg2 0)
             (equal? f /))
       '(uh oh - divide by zero)
       (f arg1 arg2)))
```

The three we will focus on

- There are three main ones that work with words and sentences:

`every`	do something to each element
`keep`	return only certain elements
`accumulate`	combine the elements

Patterns for simple recursions

- **Most recursive functions that operate on a sentence fall into:**

Mapping: `square-all` `<- every`

Counting: `count-vowels`, `count-evens`

Finding: `member`, `first-even`

Filtering: `keep-evens` `<- keep`

Testing: `all-even?`

Combining: `sum-evens` `<- accumulate`

Using every...

```
(define (square-all sent)
   (if (empty? sent)
       '()
       (se (square (first sent))
           (square-all (bf sent))
       ))

(square-all '(1 2 3 4 5))




(every square '(1 2 3 4 5))
```

Write "my-every"

```
(my-every factorial '(1 2 3 4 5))
➔ (1 2 6 24 120)
```

Write "my-keep"

```
(my-keep odd? '(1 2 3 4 5))
➔ (1 3 5)
```

lambda

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

```
(lambda (param1 param2 …)
   statement1
   statement2
       )

(lambda (x) (* x x)) ➔ [a function]
(every (lambda (x) (* x x)) '(1 2 3 4))
    ➔ (1 4 9 16)
```

Using `lambda` with `define`

- Is there a difference between:

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

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

How about between…

```
(define (special? wd)
   (member? wd (member wd '(a b c x y z))))

(define (big-proc ...)
  ... lots of code ...
  (keep special? a-sentence)
  ... more code ... )
```

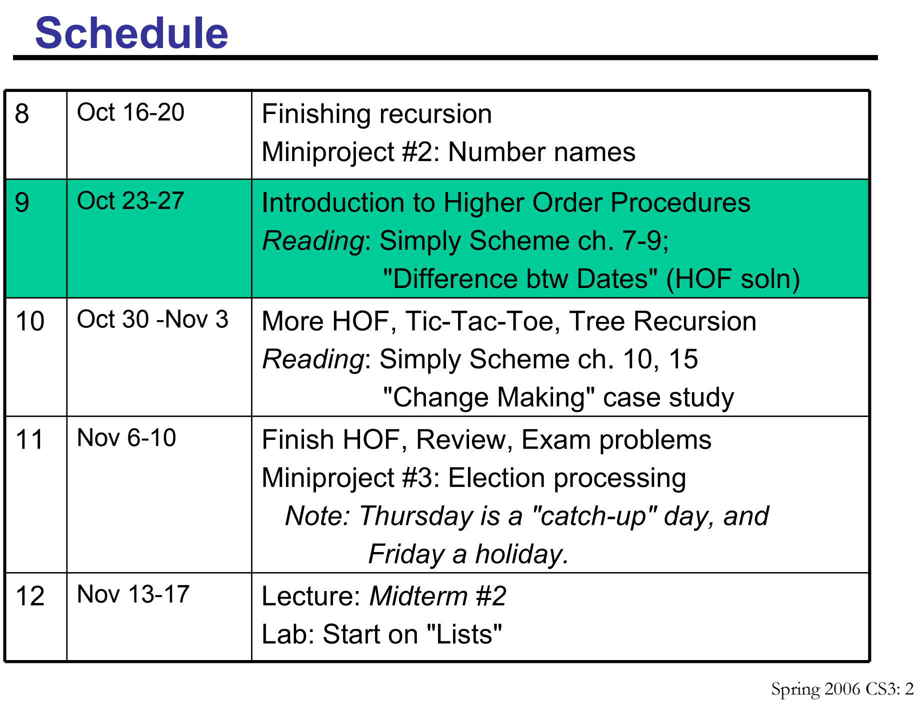
```
(define (big-proc ...)
  ... lots of code ...
  (keep (lambda (wd)
          (member wd '(a b c x y z)))
        a-sentence)
  ... more code ... )
```

CS3:
Introduction to Symbolic Programming

Lecture 9:
Higher Order Procedures

Fall 2006

Nate Titterton
nate@berkeley.edu



Schedule

8	Oct 16-20	Finishing recursion Miniproject #2: Number names
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, Tree Recursion *Reading*: Simply Scheme ch. 10, 15 "Change Making" case study
11	Nov 6-10	Finish HOF, Review, Exam problems Miniproject #3: Election processing *Note: Thursday is a "catch-up" day, and* *Friday a holiday.*
12	Nov 13-17	Lecture: *Midterm #2* Lab: Start on "Lists"

Announcements

- Surveys *really* coming this week and next
  - Take the time to do these, they are required.

What is a procedure?

(or, a *function*).

Treating functions as things

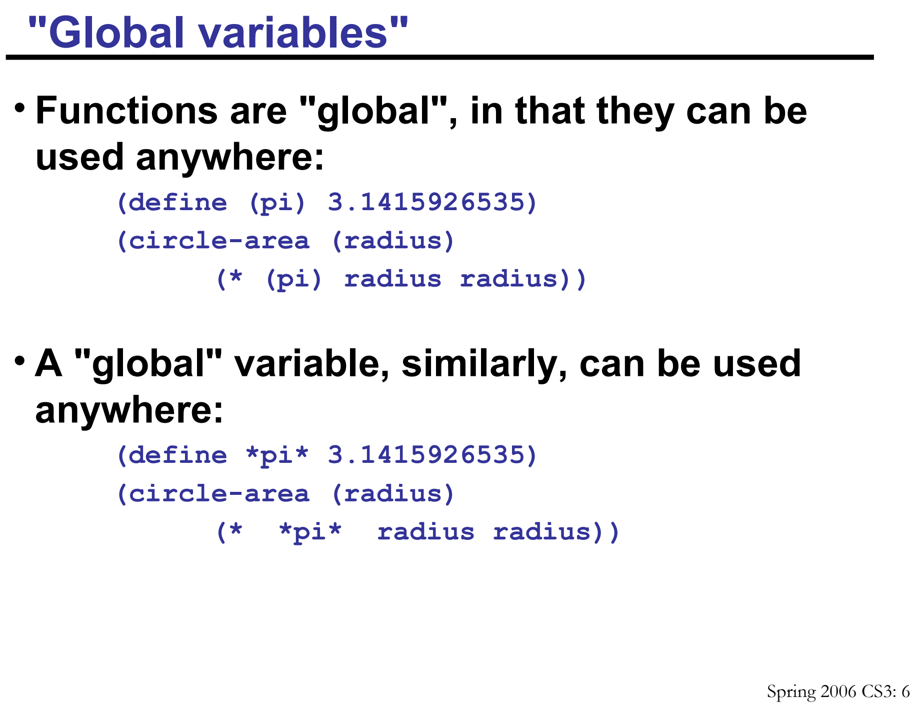
- "define" associates a name with a value

  - The usual form associates a name with a object that is a function

```
 (define (square x) (* x x))
(define (pi) 3.1415926535)
```

  - You can define other objects, though:

```
(define *pi* 3.1415926535)
(define *month-names*
   `(january february march april may
     june july august september
     october november december))
```



"Global variables"

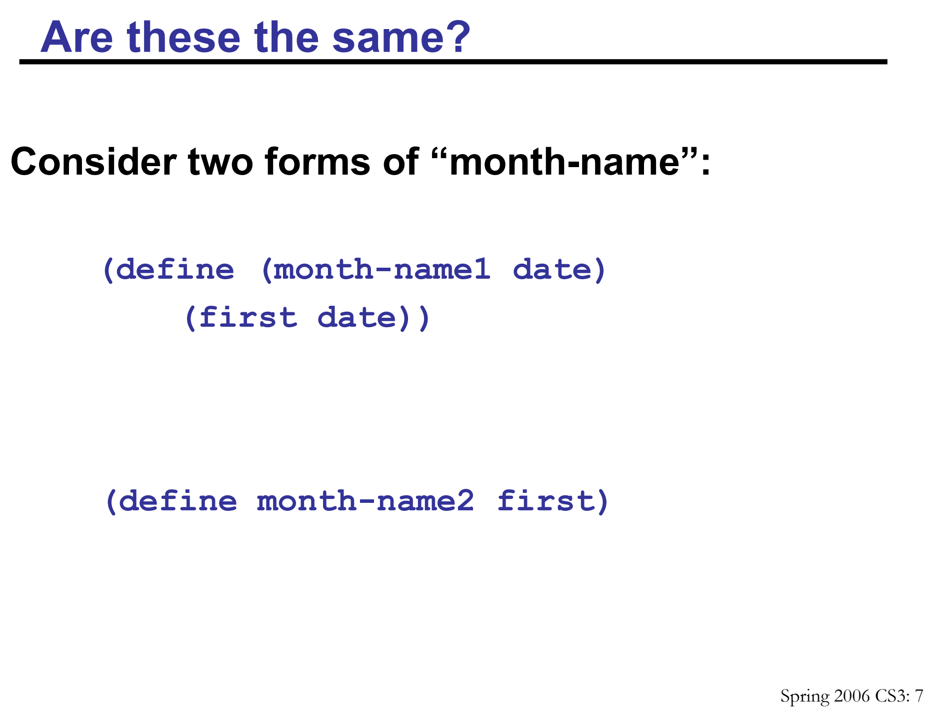
- Functions are "global", in that they can be used anywhere:

```
(define (pi) 3.1415926535)
(circle-area (radius)
      (* (pi) radius radius))
```

- A "global" variable, similarly, can be used anywhere:

```
(define *pi* 3.1415926535)
(circle-area (radius)
      (*  *pi*  radius radius))
```

The asterisks are convention, not required by scheme. Generally, when you surround a global variable with asterisks, you differentiate it from other variables you might be using inside functions (which, right now, are passed as parameters). So, also by convention, don't surround parameter names with asterisks!



Are these the same?

Consider two forms of “month-name”:

```
(define (month-name1 date)
    (first date))
```

```
(define month-name2 first)
```

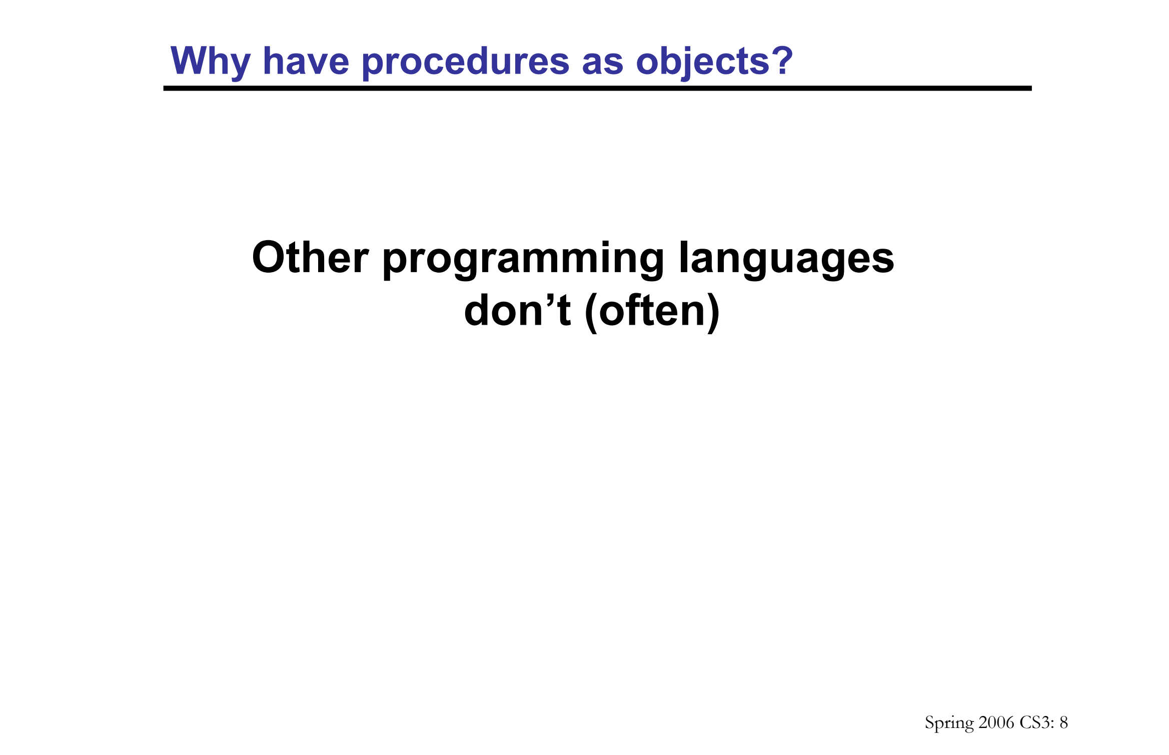
Spring 2006 CS3: 7

Yep, these are pretty much the same in practice.

In lecture, we also showed:

```
(define (joe1 num1 num2)
    (+ num1 num2))

(define jo2 +)
```

in this case, “joe1” and “joe2” are different in the number of arguments that they can take (“joe2” can take any number of numeric arguments, “joe1” can only take 2).



Why have procedures as objects?

Other programming languages don’t (often)

First-class objects (in scheme) can:

-Be named

-Be a parameter to functions

-Be returned from functions

-Be stored in other data structures

Note that functions are first class objects, but, because they are not words, they can't be stored inside sentences. (There are other data structures we will be looking at in a few weeks that can store functions).

Procedures can be taken as arguments…

```
(define (math-function? func)
  (or (equal? func +)
      (equal? func -)
      (equal? func *)
      (equal? func /)))
```

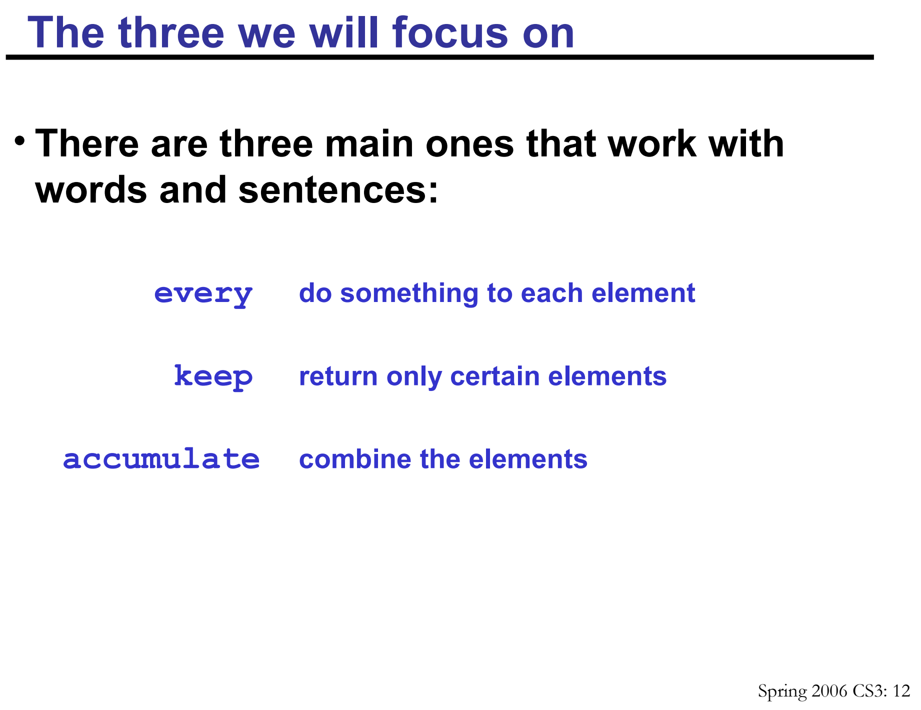
…and procedures can be returned from procedures

```
(define (choose-func name)
   (cond ((equal? name 'plus) +)
         ((equal? name 'minus) -)
         ((equal? name 'divide) /)
         (else 'sorry)))


(define (make-add-to number)
   (lambda (x) (+ number x)))

(define add-to-5 (make-add-to 5))
```

Higher order function (HOFs)

- A HOF is a function that takes a function as an argument.

```
(define (do-math f arg1 arg2)
    (if (and (equal? arg2 0)
             (equal? f /))
      '(uh oh - divide by zero)
      (f arg1 arg2)))
```



The three we will focus on

- **There are three main ones that work with words and sentences:**

`every`	do something to each element
`keep`	return only certain elements
`accumulate`	combine the elements

Every takes two arguments: a function and a sentence (or word). The function takes one argument, and is called on every element of the sentence (or word)

```
(define (factorial n)
  (if (< n 1)   1  (* n (factorial (- n 1)))))

(every factorial '(1 2 3 4 5)) --> (1 2 6 24 120)
```

Keep takes two arguments: a predicate (function) and a sentence (or word). The predicate takes one argument, and is called on each element of the sentence or word.

```
(keep odd? '(1 2 3 4 5 6 7)) --> (1 3 5 7)

(define (vowel? ltr) (member? ltr '(a e i o u)))
(keep vowel? 'mississippi) --> iiii
```

Accumulate takes two parameters: a function and a sentence (sometimes a word). The function here, however, takes two arguments.

```
(accumulate + '(1 2 3 4 5)) --> 15
```

Patterns for simple recursions

- Most recursive functions that operate on a sentence fall into:

Mapping:	`square-all`	`<- every`
Counting:	`count-vowels`, `count-evens`	
Finding:	`member`, `first-even`	
Filtering:	`keep-evens`	`<- keep`
Testing:	`all-even?`	
Combining:	`sum-evens`	`<- accumulate`

Using every…

```
(define (square-all sent)
   (if (empty? sent)
       '()
       (se (square (first sent))
           (square-all (bf sent))
       ))

(square-all '(1 2 3 4 5))



(every square '(1 2 3 4 5))
```

Write "my-every"

```
(my-every factorial '(1 2 3 4 5))
➔ (1 2 6 24 120)
```

```
(define (my-every proc sent)
  (if (empty? sent)
    '()
    (se (proc (first sent))
        (my-keep (bf sent))
        )))
```

(This version uses the “sentence” base case).

Note that the regular "every" takes care of everything but that call to proc.
That is, it takes care of

- doing the condition (identifying the base case condition)
- returning the proper base case value (although, every isn't so good at this)
- doing the combination in the recursive step
- invoking the function recursively on the smaller problem

Write "my-keep"

```
(my-keep odd? '(1 2 3 4 5))
➔ (1 3 5)
```

```
(define (my-keep pred sent)
  (cond ((empty? Sent) '())
        ((pred (first sent))
         (se (first sent)
             (my-keep pred (bf sent))))
        (else (my-keep pred (bf sent)))
```

Like “every”, the real “keep” takes care of everything but that call to pred.

That is, it takes care of

- doing the condition (identifying the base case condition)
- returning the proper base case value
- doing the combination in the recursive step
- invoking the function recursively on the smaller problem

lambda

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

```
(lambda (param1 param2 …)
   statement1
   statement2
       )

(lambda (x) (* x x)) ➔ [a function]
(every (lambda (x) (* x x)) '(1 2 3 4))
    ➔ (1 4 9 16)
```

Using `lambda` with `define`

- Is there a difference between:

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

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

How about between…

```
(define (special? wd)
   (member? wd (member wd '(a b c x y z))))

(define (big-proc ...)
  ... lots of code ...
  (keep special? a-sentence)
  ... more code ... )
```

```
(define (big-proc ...)
  ... lots of code ...
  (keep (lambda (wd)
          (member wd '(a b c x y z)))
        a-sentence)
  ... more code ... )
```