# Topic #3: Lambda

CSE 413, Autumn 2004

Programming Languages

http://www.cs.washington.edu/education/courses/413/04au/

1

## References

- Section 15.5, *Concepts of Programming Languages*
- Section 4.1.4, *Revised[5] Report on the Algorithmic Language Scheme (R5RS)*
- For more:
  - » Section 1.3, *Structure and Interpretation of Computer Programs*

2

## Scheme procedures are "first class"

- Procedures can be manipulated like the other data types in Scheme
  - » A variable can have a value that is a procedure
  - » A procedure value can be passed as an argument to another procedure
  - » A procedure value can be returned as the result of another procedure
  - » A procedure value can be included in a data structure

3

## define and name

```
(define (area-of-disk r)
  (* pi (* r r)))
```

4

## Special form: `lambda`

- **(lambda (**⟨*formals*⟩**)** ⟨*body*⟩**)**
- A lambda expression evaluates to a procedure
  - » it evaluates to a procedure that will later be applied to some arguments producing a result
- ⟨*formals*⟩
  - » formal argument list that the procedure expects
- ⟨*body*⟩
  - » sequence of one or more expressions
  - » the value of the last expression is the value returned when the procedure is actually called

5

## "Define and name" with lambda

```
(define area-of-disk
  (lambda (r)
    (* pi (* r r))))
```

6

## "Define and use" with lambda

- **((lambda (r) (* pi r r)) 1)**

7

## Separating procedures from names

- We can treat procedures as regular data items, just like numbers
  - » and procedures are more powerful because they express behavior, not just state
- We can write procedures that operate on other procedures - applicative programming

8

## define min-fx-gx

```
(define (min-fx-gx f g x)
  (min (f x) (g x)))
```

9

## apply min-fx-gx

```
(define (identity x) x)

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

(define (cube x)
  (* x x x))

(define (min-fx-gx f g x)
  (min (f x) (g x)))

(min-fx-gx square cube 2)         ; (min 4 8) => 4

(min-fx-gx square cube -2)        ; (min 4 -8) => -8

(min-fx-gx identity cube 2)       ; (min 2 8) => 2

(min-fx-gx identity cube (/ 1 2)) ; (min 1/2 1/8) => 1/8
```

10

## apply s-fx-gx

```
; define a procedure 's-fx-gx' that takes:
; s - a combining function that expects two numeric arguments
; and returns a single numeric value
; f, g - two functions that take a single numeric argument and
; return a single numeric value f(x) or g(x)
; x - the point at which to evaluate f(x) and g(x)
; s-fx-gx returns s(f(x),g(x))



   (s-fx-gx min square cube 2)   ; => (min 4 8) = 4

   (s-fx-gx min square cube -2)  ; => (min 4 -8) = -8

   (s-fx-gx + square cube 2)     ; => (+ 2 8) = 12

   (s-fx-gx - cube square 3)     ; => (- 27 9) = 18
```

11

## Exercises

```
; 1. define a procedure 's-fx-gx' that takes:
; s - a combining function that expects two numeric arguments
; and returns a single numeric value
; f, g - two functions that take a single numeric argument and
; return a single numeric value f(x) or g(x)
; x - the point at which to evaluate f(x) and g(x)
; s-fx-gx returns s(f(x),g(x))

(define (s-fx-gx s f g x)
```

12

## Exercises

```
; 2. define a procedure 'positive-fx-gx?' that takes:
; f, g - two functions that take a single numeric argument and
; return a single numeric value f(x) or g(x)
; x - the point at which to evaluate f(x) and g(x)
; positive-fx-gx? returns true if f(x) and g(x) are both > 0,
;   false otherwise
```

13

## Exercises

```
; 3. Define a procedure 'compose' that takes:
; f, g - two functions that take a single numeric argument and
; return a single numeric value f(x) or g(x)
; 'compose' returns a function that accepts one numeric
; argument and returns f(g(x))
```

14

## Exercises

```
; 4. (CHALLENGE) Define a procedure 'apply-n-times' that takes:
; f - a function that take a single numeric argument and
; return a single numeric value f(x)
; n – the number of times to apply the function f
; 'apply-n-times' returns a function that accepts one numeric
; argument 'x' and the result of applying f() to 'x', 'n' times
; Example:     ((apply-n-times square 2) 3)
                 → 81
```

15

## Example : summation

- We can always define specific functions for specific applications

```
(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a) (sum-cubes (+ a 1) b))))
```

$$\sum_{a}^{b} i^3$$

$$\frac{\pi}{8} \approx \frac{1}{1\cdot 3} + \frac{1}{5\cdot 7} + \frac{1}{9\cdot 11} + \cdots$$

```
(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1.0 (* a (+ a 2))) (pi-sum (+ a 4) b))))
```

16

## Generalize?

- Where can we generalize to perhaps provide broader application?

```
(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a) (sum-cubes (+ a 1) b))))

(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1.0 (* a (+ a 2))) (pi-sum (+ a 4) b))))
```

17

## General purpose sum

- Define the sum function so that it takes functions as arguments that calculate the current term and the next index

```
; a general purpose sum function
; args:
; term - calculate the term in the sum from a single arg x
; a - lower summation limit
; next - calculate next index value given current index value
; b - upper summation limit

(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a) (sum term (next a) next b))))


(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a) (sum-cubes (+ a 1) b))))
```

18

## Redefine sum-cubes using sum

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))



(define (inc i) (+ i 1))

(define (cube x) (* x x x))

(define (sum-cubes a b)
```

19

## Redefine pi-sum using sum

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))



(define (pi-sum a b)
  (define (pi-term i)
    (/ 1.0 (* i (+ i 2))))
  (define (pi-next i)
    (+ i 4))
  (sum pi-term a pi-next b))
```

20

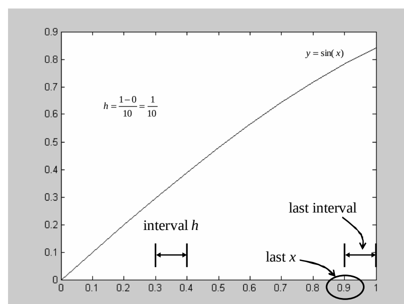
## Redefine pi-sum using sum and lambda

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))


(define (pi-sum2 a b)
  (sum
   (lambda (i) (/ 1.0 (* i (+ i 2))))
   a
   (lambda (i) (+ i 4))
   b))
```

21

## a numeric interval




```
x = [0:.1:1];
y = sin(x);
plot(x,y)
```

22

## calculate-h

```
; define a function to calculate an
; interval size (b-a)/n

(define calculate-h (lambda (a b n) (/ (- b a) n)))





; try it out on [0,1]
(calculate-h 0 1 10)
```

$\frac{1}{10}$

23

## anonymous calculate-h

```
; do the same thing without naming the function

((lambda (a b n) (/ (- b a) n)) 0 1 10)
```

$\frac{1}{10}$

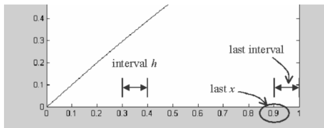
24

## calculate last-x

```
; define a function that figures out what the beginning
; of the last interval is

; calculate a+((n-1)*h) directly

(define (last-x1 a b n)
  (+ a (* (- n 1) (/ (- b a) n))))
```




25

## last-x using a helper function

```
; calculate a+(k*h) using a simple function, and
; pre-calculate k and h to pass to the function


(define (last-x2 a b n)
  (define (use-kh k h)
    (+ a (* k h )))
  (use-kh (- n 1) (/ (- b a) n)))
```

26

## last-x using anonymous helper function

```
; calculate a+(k*h) using an anonymous function

(define (last-x3 a b n)
  ((lambda (k h) (+ a (* k h )))
   (- n 1)
   (/ (- b a) n)))
```

27

## last-x with concealed anonymous function

```
; hide the use of the anonymous function by using let

(define (last-x4 a b n)
  (let ((h (/ (- b a) n))
        (k (- n 1)))
    (+ a (* k h ))))
```

28

## Special form `let`

**(let** (($\langle var_1 \rangle$ $\langle exp_1 \rangle$ )
($\langle var_2 \rangle$ $\langle exp_2 \rangle$ ))
$\langle body \rangle$ **)**

- When the `let` is evaluated, each expression $exp_i$ is evaluated and the resulting value is associated with the related name $var_i$, then the *body* is evaluated.
- There is no order implied in the evaluation of $exp_i$

29

## scope and `let`

```
; an example in scoping with let

(define x 2)

(let ((x 3)
      (y (+ x 2)))
  (* x y))

((lambda (x y)
   (* x y))
 3
 (+ x 2))
```

30

## nesting `lets` lets us get x

```
; nested lets and let*

(define x 2)

(let ((x 3))
  (let ((y (+ x 2)))
    (* x y)))

(let* ((x 3)
       (y (+ x 2)))
  (* x y))
```



## Special form `let*`

**(let*** **(**(⟨$var_1$⟩ ⟨$exp_1$⟩**)**
**(**⟨$var_2$⟩ ⟨$exp_2$⟩**))**
⟨*body*⟩**)**

- When the `let*` is evaluated, each expression $exp_i$ is evaluated in turn and the resulting value is associated with the related name $var_i$, then the *body* is evaluated.
- The $exp_i$ are evaluated in left to right order
  - » each binding indicated by **(**⟨$var_i$⟩ ⟨$exp_i$⟩**)** is part of the environment for **(**⟨$var_{i+1}$⟩ ⟨$exp_{i+1}$⟩**)** and following
  - » This is exactly equivalent to nesting the `let` statements



## an iterator with parameter h

```
; show all the x values on the interval

(define (show-x1 a b n)
  (define (iter h count)
    (if (> count n)
        (newline)
        (begin
          (display (+ a (* h count)))
          (display " ")
          (iter h (+ count 1)))))
  (iter (/ (- b a) n) 0))
```



## h defined in enclosing scope

```
; show all the x values on the interval
; using let

(define (show-x2 a b n)
  (let ((h (/ (- b a) n)))
    (define (iter count)
      (if (> count n)
          (newline)
          (begin
            (display (+ a (* h count)))
            (display " ")
            (iter (+ count 1)))))
    (iter 0)))
```



## Special form `begin`

**(begin** ⟨$exp_1$⟩ ⟨$exp_2$⟩ … ⟨$exp_n$⟩**)**

- Evaluate the $exp_i$ in sequence from left to right
- The value returned by the entire `begin` expression is the value of $exp_n$
- Best used to sequence side effects like I/O
  - » for example displaying each of the x values in show-x
- There is implicit sequencing in the body of a `lambda` procedure or a `let` but we generally don't use it
  - » the procedure returns the value of the last $exp_i$, so the body of most of our procedures consists of one expression only



## show-x

```
Welcome to DrScheme, version 205.
Language: Standard (R5RS).
> (show-x2 0 1 10)
0  1/10  1/5  3/10  2/5  1/2  3/5  7/10  4/5  9/10 1
>
```