More Lambda

CSE 413, Autumn 2002

Programming Languages

http://www.cs.washington.edu/education/courses/413/02au/

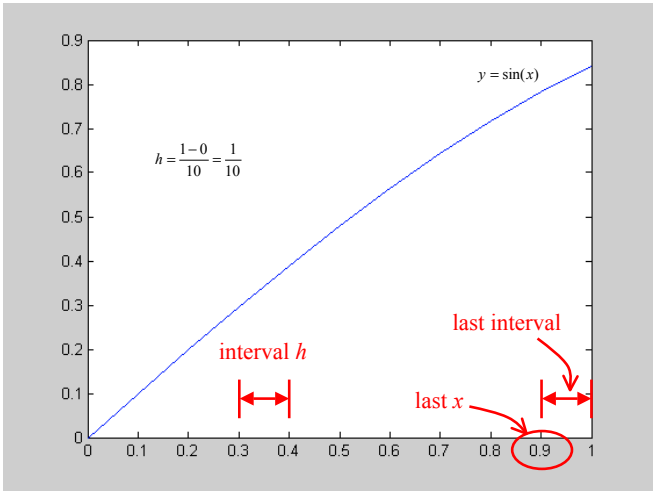
Readings and References

- Reading
  - » Section 1.3, *Structure and Interpretation of Computer Programs*, by Abelson, Sussman, and Sussman, but you've already read this, right?
- Other References

Procedures as unnamed blobs

- With `lambda`, we've separated the body of the procedure from any particular name for the procedure
- Procedures are objects like any other, and can be handed around from procedure to procedure as arguments, return values, etc
- Procedures can be defined and applied without ever getting a name assigned to them

a numeric interval




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

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)
```

```
 1
 —
10
```

define a function body ...

… and bind it to the name calculate-h

apply the function to some arguments

anonymous calculate-h

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

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

```
 1
 —
10
```

define a function body ...

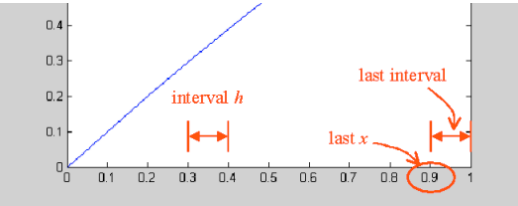
… and apply it to some arguments

calculate last-x

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

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

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



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)))
```

define a function body and bind it to the name `use-kh`

apply `use-kh` to some arguments

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)))
```

k

h

define a function body ...

… and apply it to some arguments

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 ))))
```

bind some values to some names ...

… and use those names in the body of the `let`

Special form `let`

(let ((⟨*var*$_1$⟩ ⟨*exp*$_1$⟩**)**
(⟨*var*$_2$⟩ ⟨*exp*$_2$⟩**))**
⟨*body*⟩**)**

- 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*$_i$
- This is exactly the same as parameter evaluation before a procedure call
  - » This *is* parameter evaluation before a procedure call

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))
```

this `let` and this `lambda` are equivalent

scope of the local x and y

the parameter values are calculated *outside* the scope of the parameter variables in the procedure

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))
```

is this x referenced anywhere?

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 $\langle exp_1 \rangle \langle exp_2 \rangle \ldots \langle exp_n \rangle$**)**

- 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

sequencing with `begin`

```
; 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: `if`

```
(if exp
    tx
    fx)
```


show-x

```
Welcome to DrScheme, version 201.
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
>
```