Lambda

CSE 413, Autumn 2005

Programming Languages

http://www.cs.washington.edu/education/courses/413/05au/

References

- Section 1.3, *Structure and Interpretation of Computer Programs*
- Section 4.1.4, *Revised^5 Report on the Algorithmic Language Scheme (R5RS)*

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

Recall: Define and name a procedure

- **`(define (`**⟨*name*⟩ ⟨*formal params*⟩**`)`** ⟨*body*⟩**`)`**
  - » **`define`** - special form
  - » *name* - the name that the procedure is bound to
  - » *formal params* - names used within the body of procedure
  - » *body* - expression (or sequence of expressions) that will be evaluated when the procedure is called.
  - » The result of the last expression in the body will be returned as the result of the procedure call

define and name

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

- » define a procedure that takes one argument **`r`** and calculates **`(* pi (* r r)))`**
- » bind that procedure to the name **`area-of-disk`**

- The name of the variable that holds the procedure and the actual body of the procedure are separate issues

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

"Define and name" with lambda

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

- » define a procedure that takes one argument **`r`** and calculates **`(* pi (* r r))`**
- » bind that procedure to the name **`area-of-disk`**

- The name of the variable that holds the procedure and the actual body of the procedure are separate issues

"Define and use" with lambda

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

  - » define a procedure that takes one argument **`r`** and calculates **`(* pi r r)`**
  - » apply that procedure to the argument value 1
  - » return the result => pi

- The body of the procedure is applied directly to the argument and is never named at all

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
  - » higher order functions
  - » functions that take functions as arguments and do standard things with them

---

define min-fx-gx

```
; define a procedure that takes two functions
; and a numeric value, and returns the min of
; f(x) and g(x)

(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)))
```

---

apply min-fx-gx

```
(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
```

---

define s-fx-gx

```
; define a procedure 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 identity
  (lambda (x) x))

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

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

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

apply s-fx-gx

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

---

apply s-fx-gx

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

```
(s-fx-gx
 (lambda (x y) (expt x y))
 identity
 identity
 2)                    ; => (expt 2 2) = 4

(s-fx-gx
 (lambda (x y) (/ x y))
 cube
 square
 14)                   ; => (/ x³ x²) = x = 14

(s-fx-gx
 (lambda (x y) (/ (+ x y) 2))
 (lambda (x) (+ x 1))
 (lambda (e) (- e 1))
 128)                  ; => (avg (+ x 1) (- x 1)) => x = 128
```

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))))
```

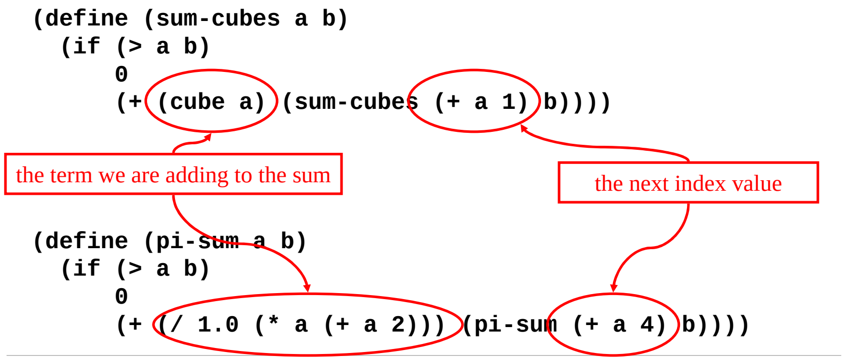
$$\sum_{a}^{b} i^3$$

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

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

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))))
```

the term we are adding to the sum

the next index value

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

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))))
```

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)
  (sum cube a inc b))
```

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))
```

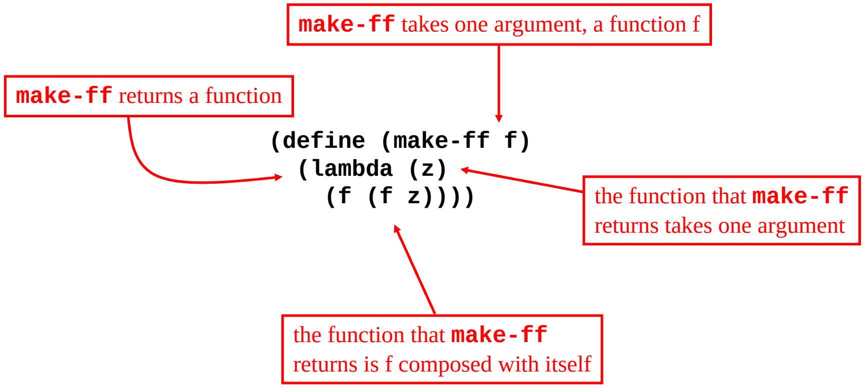
Redefine pi-sum using sum and lambda

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


; define pi-sum without explicit auxiliary functions

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

Define “make-ff”

- Define a procedure **`make-ff`**
  - » takes a procedure of one argument as its argument
    - ie, a procedure f that can be applied (f x)
  - » returns a procedure that applies the original procedure twice
    - ie, a procedure g that is (f (f x))
- For example
  - » **`(make-ff inc)`** returns a procedure **`(inc (inc x))`**

The parts of `make-ff`



`make-ff` takes one argument, a function f

`make-ff` returns a function

```
(define (make-ff f)
  (lambda (z)
    (f (f z))))
```

the function that **`make-ff`** returns takes one argument

the function that **`make-ff`** returns is f composed with itself

Evaluate expressions with `make-ff`

```
(define (make-ff f)
  (lambda (z)
    (f (f z))))

(define (inc x) (+ x 1))

((make-ff inc) 3)    ; 2 + 3 = 5

(define (plus4 x)
  ((make-ff (make-ff inc)) x))

(plus4 10)          ; 10+4 = 14
```