

FP Foundations, Scheme

In Text: Chapter 14

1

Mathematical Functions

- Def: A **mathematical function** is a mapping of members of one set to another set
- The "input" set is called the **domain**
- The "output" set is called the **range**
- Each input maps to exactly one output

■ Chapter 14: FP Foundations, Scheme ■

2

Scheme Syntax Basics

- Case-insensitive
- Data Types:
 - **Atoms**: identifiers, symbols, numbers
 - **Lists** (S-expressions)
 - List form: parenthesized collections of sublists and/or atoms
 - (a b c d)
 - (a (b c) d e)
- All lists are internally represented by singly-linked chains where each node has 2 pointers (think "data" and "next")

■ Chapter 14: FP Foundations, Scheme ■

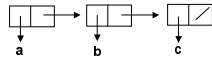
3

Internal List Representation

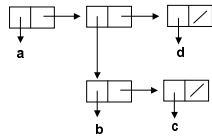
- Single atom:



- List of atoms: (a b c)



- List containing list:
(a (b c) d)



Chapter 14: FP Foundations, Scheme

4

Primitive Functions

- Arithmetic:** +, -, *, /, abs, sqrt

(+ 5 2)

(* 47 (+ (- 5 3) 2))

- QUOTE** takes one parameter; returns the parameter without evaluation

- Parameters to a function are evaluated before applying the function; use QUOTE to prevent it when inappropriate

- QUOTE can be abbreviated with the apostrophe prefix operator

'(a b)

(quote (a b))

Chapter 14: FP Foundations, Scheme

5

Primitive Functions (cont.)

- CAR** takes a list parameter; returns the first element of that list

(car '(a b c)) yields a

(car '((a b) c d)) yields (a b)

- CDR** takes a list parameter; returns the list after removing its first element

(cdr '(a b c)) yields (b c)

(cdr '((a b) c d)) yields (c d)

Chapter 14: FP Foundations, Scheme

6

Primitive Functions (cont.)

5. **CONS** takes two parameters; returns a new list that includes the first parameter as its first element and the second parameter as the remainder

```
(cons 'a '(b c)) returns (a b c)
```

6. **LIST** takes any number of parameters; returns a list with the parameters as elements

```
(list 'a '(b c)) returns (a (b c))
```

Predicate Functions

- A **predicate** is a function returning a boolean value
- #T is true and (), the empty list, is false
- By convention, Scheme predicates have names that end in a "?"

1. **EQUAL?** takes two parameters; it returns #T if both parameters are "the same"; works on about everything

Predicate Functions (cont.)

2. **EQV?** (atom equality) takes two symbolic parameters; it returns #T if both parameters are atoms and the two are the same

```
(eqv? 'a 'a) yields #t
```

```
(eqv? 'a '(a b)) yields ()
```

- EQV? is **unreliable** if used on lists
3. **EQ?** (pointer equality) is like EQV?, but is also **unreliable** on numbers, characters, and a few other special cases

Predicate Functions (cont.)

4. **LIST?** takes one parameter; it returns #T if the parameter is a list; otherwise ()
5. **NULL?** takes one parameter; it returns #T if the parameter is the empty list; otherwise ()
6. Numeric Predicate Functions: =, >, <, >=, <=, even?, odd?, zero?, positive?, negative?

■ Chapter 14: FP Foundations, Scheme ■

10

Other Useful Functions

- (write expression)
- (write-string expression)
- (newline)
- (read-string stop-character-set)

■ Chapter 14: FP Foundations, Scheme ■

11

Lambda Expressions

- A lambda expression describes a "nameless" function
- Specifies both the parameter(s) and the mapping
- Consider this function $\text{cube}(x) = x * x * x$
- Corresponding lambda expr: $\lambda(x) x * x * x$
- Can be "applied" to parameter(s) by placing the parameter(s) after the expression
e.g. $(\lambda(x) x * x * x)(3)$
- The above application evaluates to 27

■ Chapter 14: FP Foundations, Scheme ■

12

Lambda Expressions in Scheme

- Based on λ notation
`(lambda (l) (car (car l)))`
- L is called a "bound variable"; think of it as a formal parameter name
- Lambda expressions can be applied
`((lambda (l) (car (car l))) '(a b c d))`

■ Chapter 14: FP Foundations, Scheme ■

13

A Function for Constructing Functions

DEFINE has two forms:

1. To bind a symbol to an expression:
`(define pi 3.141593)`
`(define two_pi (* 2 pi))`
2. To bind names to lambda expressions
`(define (cube x)`
`(* x x x)`
`)`

Example use: `(cube 4)`

■ Chapter 14: FP Foundations, Scheme ■

14

Functional Forms

- Def: A **higher-order function**, or **functional form**, is one that:
 - Takes function(s) as parameter(s), or
 - Yields a function as its result, or
 - Both

■ Chapter 14: FP Foundations, Scheme ■

15

Function Composition

- A functional form that takes two functions as parameters and yields a function whose result is a function whose value is the first actual parameter function applied to the result of the application of the second
- Form: $h \equiv f \circ g$
- Which means: $h(x) \equiv f(g(x))$

■ Chapter 14: FP Foundations, Scheme ■

16

Construction

- A functional form that takes a list of functions as parameters and yields a list of the results of applying each of its parameter functions to a given parameter
- Form: $[f, g]$
- For $f(x) \equiv x * x * x$ and $g(x) \equiv x + 3$,
 $[f, g](4)$ yields $(64, 7)$

■ Chapter 14: FP Foundations, Scheme ■

17

Apply-to-All

- A functional form that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters
- Form: α
- For $h(x) \equiv x * x * x$
- $\alpha(h, (3, 2, 4))$ yields $(27, 8, 64)$
- "map" in Scheme is a very flexible version of this

■ Chapter 14: FP Foundations, Scheme ■

18

Scheme Evaluation Process

1. Parameters are evaluated, in no particular order
2. The values of the parameters are substituted into the function body
3. The function body is evaluated
4. The value of the last expression in the body is the value of the function

- (Special forms use a different evaluation process)

■ Chapter 14: FP Foundations, Scheme ■

19

Control Flow: Selection

1. **Selection:** the special form, **IF**
(IF *predicate* *then_exp* *else_exp*)

```
(if (not (zero? count))
    (/ sum count)
    0
)
```

■ Chapter 14: FP Foundations, Scheme ■

20

Control Flow: Multi-Way Selection

2. **Multiple selection:** the special form, **COND**
(COND

```
(predicate_1 expr {expr})
(predicate_1 expr {expr})
...
(predicate_1 expr {expr})
(ELSE expr {expr})
)
```

- Returns the value of the last *expr* in the first pair whose *predicate* evaluates to true

■ Chapter 14: FP Foundations, Scheme ■

21

Introducing Locals: Let

3. **Internal definitions:** the special form, **LET**

```
(let ( (x '(a b c))
      (y '(d e f)) )
  (cons x y)
)
```

- Introduces a list of local names (use `define` for top-level entities, but use `let` for internal definitions)
- Each name is given a value
- Use `let*` if later values depend on earlier ones

■ Chapter 14: FP Foundations, Scheme ■

22

Control Flow: Named Let

2. **Recursive-style looping:** the special form, **named LET**

```
(let loop-name ( definitions )
  ( expr )
  ...
  (cond ((continue condition) (loop-name ...))
        (else exit-value))
)
```

- The *loop-name* is then used like a function call to recursively repeat the body

■ Chapter 14: FP Foundations, Scheme ■

23

Example Scheme Functions: member

- **member** takes an atom and a list; returns #T if the atom is in the list; () otherwise

```
(define (member atm lis)
  (cond
    ((null? lis) '())
    ((equal? atm (car lis)) #t)
    (else (member atm (cdr lis))))
)
```

■ Chapter 14: FP Foundations, Scheme ■

24

Example Scheme Fns: flat-equal

- **flat-equal** takes two simple lists as parameters; returns #T if the two simple lists are equal; () otherwise

```
(define (flat-equal lis1 lis2)
  (cond
    ((null? lis1) (null? lis2))
    ((null? lis2) '())
    ((eqv? (car lis1) (car lis2))
     (flat-equal (cdr lis1) (cdr lis2)))
    (else '())))
```

■ Chapter 14: FP Foundations, Scheme ■

25

Example Scheme Fns: equal

- **equal** takes two lists as parameters; returns #T if the two general lists are equal; () otherwise

```
(define (equal lis1 lis2)
  (cond
    ((not (list? lis1)) (eqv? lis1 lis2))
    ((not (list? lis2)) '())
    ((null? lis1) (null? lis2))
    ((null? lis2) '())
    ((equal (car lis1) (car lis2))
     (equal (cdr lis1) (cdr lis2)))
    (else '())))
```

■ Chapter 14: FP Foundations, Scheme ■

26

Example Scheme Fns: append

- **append** takes two lists as parameters; returns the first parameter list with the elements of the second parameter list appended at the end

```
(define (append lis1 lis2)
  (cond
    ((null? lis1) lis2)
    (else
     (cons
      (car lis1)
      (append (cdr lis1) lis2))))
```

■ Chapter 14: FP Foundations, Scheme ■

27

Creating Functional Forms

- Composition: the previous examples have used it
- Apply to All: one form in Scheme is mapcar
- Applies the given function to all elements of the given list; result is a list of the results

```
(define (mapcar fun lis)
  (cond
    ((null? lis) '())
    (else (cons (fun (car lis))
                 (mapcar fun (cdr lis))))))
```

■ Chapter 14: FP Foundations, Scheme ■

28

Interpretive features

- One can define a function that builds Scheme code and requests its interpretation
- The interpreter is a user-available function, **EVAL**
- Suppose we have a list of numbers that must be added together

```
(define (adder lis)
  (cond
    ((null? lis) 0)
    (else (eval (cons '+ lis))))))
```

- The parameter is a list of numbers to be added; adder inserts a + operator and interprets the resulting list

■ Chapter 14: FP Foundations, Scheme ■

29

Imperative Features in Scheme

- "Functions" that modify/change a data structure are called **mutators**
- By convention, mutator names end in "!"
- SET! binds or rebinds a value to a name
- SET-CAR! replaces the car of a list
- SET-CDR! replaces the cdr part of a list

■ Chapter 14: FP Foundations, Scheme ■

30
