

Functional Programming - Illustrated in Scheme

Original Lisp slides © Diane Horton 2000.
Adapted for Scheme by Eric Joanis.

References:

- Dybvig
- Sebesta, chapter 14,

1

Jumping right in

A Scheme function:

```
(define (increment n)
  (+ n 1)
)
```

A call to the function:

```
(increment 21)
```

2

The Spirit of Lisp-like Languages

We shall first define a class of **symbolic expressions** in terms of ordered pairs and lists. Then we shall define five elementary **functions and predicates**, and build from them by **composition, conditional expressions** and **recursive definitions** an extensive class of functions of which we shall give a number of examples. We shall then show how these **functions can themselves be expressed as symbolic expressions**, and we shall give a **universal function *apply*** that allows us to compute from the expressions for a given function its value for given arguments. Finally, we shall define some **functions with functions as arguments** and give some useful examples.

McCarthy, J, [1960]. Recursive functions of symbolic expressions and their computation by machine, Part I. *Comm. ACM* 3:4; quoted in Sethi.

3

Pure Functional Programming

- A program consists of function definitions and calls. There is no other structure.
- No loops. Just recursion.
- No side effects – a function may not change its parameters, or do input or output.
- No variables. Just parameters.
- Implicit memory mgmt: no new or free.
- One data structure: the list. Can represent anything, including code.
- Functions can be:
 - passed as an argument;
 - returned from a function;
 - represented by a data structure;
 - and such a data structure can then be evaluated.

4

The MIT Scheme Interface

```
eddie 1% scheme
Scheme Microcode Version ...

1 ]=> (+ 8 3 5 16 9)
;Value: 41

1 ]=> (define (increment n)
      (+ n 1)
      )
;Value: increment

1 ]=> (increment 21)
;Value: 22

1 ]=> (load "egs")

;Loading "egs" -- done
;Value: increment_list

1 ]=> (increment_list (1 32 7))
;The object 1 is not applicable.
;To continue, call RESTART with an option number:
; (RESTART 2) => Specify a procedure to use in its place.
; (RESTART 1) => Return to read-eval-print level 1.

2 error> (restart 1)
;Abort!

1 ]=> (increment_list '(1 32 7))
;Value 1: (2 33 8)
```

5

How Scheme Evaluates

To evaluate an expression, Scheme always does the following:

- Evaluate the operands.
- Apply the operation to the results.

Well, not always. Examples:

These are called **special forms** because they violate the rule.

Why did *(increment 21)* work?

7

```
1 ]=> (trace increment_list)
;No value

1 ]=> (increment_list '(1 32 7))

[Entering #[compound-procedure 2 increment_list]
  Args: (1 32 7)]
[Entering #[compound-procedure 2 increment_list]
  Args: (32 7)]
[Entering #[compound-procedure 2 increment_list]
  Args: (7)]
[Entering #[compound-procedure 2 increment_list]
  Args: ()]
[()
  <=#[compound-procedure 2 increment_list]
  Args: ()]
[(8)
  <=#[compound-procedure 2 increment_list]
  Args: (7)]
[(33 8)
  <=#[compound-procedure 2 increment_list]
  Args: (32 7)]
[(2 33 8)
  <=#[compound-procedure 2 increment_list]
  Args: (1 32 7)]
;Value 3: (2 33 8)

1 ]=> (exit)

Kill Scheme (y or n)? Yes
Happy Happy Joy Joy.
eddie 2%
```

6

Basic Scheme

Pre-defined Functions

- *(+ -numbers-)*: Returns the sum of the numbers.
- Other arithmetic operators: *-*, ***, */*
- *(eq? a b)*: Returns *#t* iff *a* and *b* are the same Scheme object. (Use *=* for numbers.)
- *(null? object)*: Returns *#t* iff *object* is the empty list (or *#f*, depending on the version).
- *(car list)*: Returns the first element of *list*. Pre: *list* must be a list with 1 or more elements.
- *(cdr list)*: Returns *list* with the first element removed. Pre: *list* must be a list with 1 or more elements.
- *(caddr list)*: Returns the *car* of the *cdr* of the *cdr* ...

8

- (*cons a b*): Returns a list whose *car* is *a* and whose *cdr* is *b*. Pre: *b* must be a list.
- (*append a b*): Returns the list formed by joining the elements of *a* and *b* together. Pre: *a* and *b* must be lists.
- (*list -objects-*): Returns a list of the *objects*.
- (**quote** object): Returns *object* without evaluating it. Same as 'object.

Other things

- #t: Means "true".
- #f: Means "false". Same as '() in some implementations.

```
(cond (test1 exp11 exp12 ...)
      (test2 exp21 exp22 ...)
      ...
      (else expe1 expe2 ...))
```

9

More Interesting Examples

Write functions to:

1. Return the sum of the first *n* integers.
2. Increment every integer in a list.
3. Reverse the elements of a list.
4. Build and return a list of the first *n* odd numbers.
5. Return the last element of a list
6. Take a list of lists, and return a new list containing that last element of each.
7. Count the number of atoms, at all levels, in a list.
8. Return a list of all the atoms, at all levels, in a list.

10

Allatoms: version 1

```
(define (a1 lst)
  (cond ((null? lst) '())
        ((= (length lst) 1) lst)
        (else (cons (a1 (car lst))
                     (a1 (cdr lst))))))
```

```
1 ]=> (a1 '((b c)))
;Value 1: ((b c))
```

```
1 ]=> (a1 '(a (b c) d))
;The object b, passed as the first argument
to length, is not the correct type.
```

11

Allatoms: version 2

```
(define (a2 lst)
  (cond ((null? lst) '())
        ((= (length lst) 1) lst)
        (else (append (if (pair? (car lst))
                           (a2 (car lst))
                           (list (car lst)))
                       (a2 (cdr lst))))))
```

```
1 ]=> (a2 '(a (b c) d))
;Value 4: (a b c d)
```

```
1 ]=> (a2 '((a b c) ((d)) (e (f (g)) h)))
;Value 5: (a b c (d) (e (f (g)) h))
```

```
1 ]=> (a2 '((b c)))
;Value 6: ((b c))
```

12

Allatoms: version 3

```
(define (a3 lst)
  (cond ( (null? lst) '() )
        ( else (append (if (pair? (car lst))
                           (a3 (car lst))
                           (list (car lst)))
                        (a3 (cdr lst))) )
  )
)
```

```
1 ]=> (a3 '((b c)) )
;Value 7: (b c)
```

```
1 ]=> (a3 '(a (b c) d) )
;Value 8: (a b c d)
```

```
1 ]=> (a3 '((a b c) ((d)) (e (f (g)) h)))
;Value 9: (a b c d e f g h)
```

13

Allatoms: version 4

```
(define (a4 lst)
  (cond ( (null? lst) '() )
        ( (pair? lst) (append (a4 (car lst))
                               (a4 (cdr lst))) )
        ( else (list lst) )
  )
)
```

This is simpler and more general, but changes the specification of the function:

```
1 ]=> (a4 '((a b c) ((d)) (e (f (g)) h)))
;Value 10: (a b c d e f g h)
```

```
1 ]=> (a4 '(a . b))
;Value 11: (a b)
```

```
1 ]=> (a4 'a)
;Value 12: (a)
```

14

Countatoms

```
(define (countatoms lst)
  (cond ( (null? lst) 0 )
        ( else (+ (if (pair? (car lst))
                     (countatoms (car lst))
                     1 )
                 (countatoms (cdr lst))) )
  )
)
```

```
(define (countall lst)
  (cond ( (null? lst) 0 )
        ( (pair? lst) (+ (countall (car lst))
                          (countall (cdr lst))) )
        ( else 1 )
  )
)
```

15

Efficiency Issues

Pitfall 1: **Building a list and traversing it again.**

Example:

```
(define (rev lst)
  (cond ( (null? lst) '() )
        ( else (append (rev (cdr lst))
                        (list (car lst))) )
  )
)
```

16

Efficient version:

```
(define (rev2 rest sofar)
  (cond ( (null? rest) sofar )
        ( else (rev2 (cdr rest)
                      (cons (car rest) sofar))
        )
  )
)
```

A nicer interface.

```
(define (my-reverse lst)
  (rev2 lst '())
)
```

17

Pitfall 2:

Evaluating the same expression twice.

Example:

```
(define (mostatoms lst)
  (cond ( (null? lst) 0 )
        ( (> (countatoms (car lst))
              (mostatoms (cdr lst)))
          (countatoms (car lst))
        )
        ( else (mostatoms (cdr lst)) )
  )
)
```

18

Efficient version:

```
(define (most2 lst)
  (cond ( (null? lst) 0 )
        ( (biggest (countatoms (car lst))
                   (most2 (cdr lst)))
        )
  )
)
(define (biggest x y)
  (if (> x y) x y)
)
```

In fact, Scheme has a built-in *max* function, so in this case we don't need to write our own helper.

19

Let

When a helper function is not a natural solution to Pitfall 2, use **let**.

```
(let ( ( var1 value1 )
      ( var2 value2 )
      ...
      ( varn valuen )
    )
  ; Can now use var1 through varn
)
; var1 through varn now revert to having
; no value, or to their previous values if
; they had any.
```

20

More Practise Writing Functions

; Returns the sum of all elements of *lst*.
; Precondition:

```
(define (sumlist lst)
  (cond ( (null? lst) 0 )
        ( else (+ (car lst)
                   (sumlist (cdr lst))) )
  )
)
```

21

; Returns a list containing the sum for each
; list in *ll*.
; Precondition:

```
(define (sums ll)
  (cond ( (null? ll) '() )
        ( else (cons (sumlist (car ll))
                      (sums (cdr ll))
                      )
        )
  )
)
```

22

; Returns a list containing the length of each
; list in *ll*.
; Precondition:

```
(define (lengths ll)
  (cond ( (null? ll) '() )
        ( else (cons (length (car ll))
                      (lengths (cdr ll))
                      )
        )
  )
)
```

23

; Returns a list containing the cdr of each
; list in *ll*.
; Precondition:

```
(define (cdrs ll)
  (cond ( (null? ll) '() )
        ( else (cons (cdr (car ll))
                      (cdrs (cdr ll))
                      )
        )
  )
)
```

24

Passing Functions as Arguments

Example 1: *consall*

We could have different functions *sums*, *lengths*, and *cdrs*. Each one does a certain thing to every element of a list.

The code is very repetitive. And if we want to do something else to every element of a list, we need to write yet another function.

Instead, we can write one general function.

```
; Returns a list containing the result of applying fn
; to each element of lst.
; Precondition:
```

```
(define (consall lst fn)
  (cond ((null? lst) '())
        (else (cons (fn (car lst))
                     (consall (cdr lst) fn)
                     )
        )
  )
)
```

25

Sample run

```
eddie 1% scheme -load fcnpars.scm
Scheme Microcode Version ...

;Loading "fcnpars.scm" -- done

1 ]=> (sums '((3 1) (6 29 4) (5) ()))
;Value 1: (4 39 5 0)

1 ]=> (lengths '((3 1) (6 29 4) (5) ()))
;Value 2: (2 3 1 0)

1 ]=> (cdrs '((3 1) (6 29 4) (5)))
;Value 3: ((1) (29 4) ())

1 ]=> (consall '((3 1) (6 29 4) (5) ()) sumlist)
;Value 4: (4 39 5 0)

1 ]=> (consall '((3 1) (6 29 4) (5) ()) length)
;Value 5: (2 3 1 0)

1 ]=> (consall '((3 1) (6 29 4) (5)) cdr)
;Value 6: ((1) (29 4) ())
```

26

Aside: *map*

Scheme has a built-in function that is a more general version of our *consall*; it is called *map*.

Example:

```
1 ]=> (map length '((1 2 3) (5) () (1 2 3 4)))
;Value 7: (3 1 0 4)
```

; equivalent to:

```
1 ]=> (consall '((1 2 3) (5) () (1 2 3 4)) length)
;Value 8: (3 1 0 4)
```

map can even apply a function that takes more than one argument; you just supply a list of values for the 1st argument, then a list of values for the second, etc.

```
1 ]=> (map > '(99 2 5 -1) '(88 4 5 3))
;Value 9: (#t () () ())
```

27

Example 2 (of passing functions): *prune*

Suppose we want a function that will test every element of a list and return a list containing only those that pass the test.

We want it to be very general: it should be able to use any test we might give it. How will we tell it what test to apply?

What should a function call look like?

Example: Prune out the elements of *myList* that are not atoms.

Now let's write the function.

28

```
; Return a new list containing only the elements of list
; that pass the test.
; Precondition:
```

```
(define (prune lst test)
  (cond ((null? lst) '())
        ((test (car lst))
         (cons (car lst)
               (prune (cdr lst) test)))
        (else (prune (cdr lst) test))))
)
```

Sample run

```
1 ]=> (define (atom? x) (not (pair? x)))
;Value: atom?

1 ]=> (prune '((3 1) 4 (x y z) (x) y ()) atom?)
;Value 12: (4 y ())

1 ]=> (prune '(() (a b c) (1 2) () (()) (x (y w) z)) null?)
;Value 13: (() ())
```

29

Write calls to *prune* that will prune *myList* in these ways:

- Prune out elements that are null.
- (Assume *myList* contains lists of integers.) Prune out elements whose minimum is not at least 50. Hint: there is a built-in *min* function.
- (Assume *myList* contains lists.) Prune out elements that themselves have more than 2 elements.

This is becoming tedious. We need to declare a function for each possible test we might dream up.

30

Unnamed Functions

Fortunately, we can define functions without giving them a name. These are called "Lambda Expressions".

Example definition:

```
(lambda (lst) (< (length lst) 2))
```

To call an unnamed function, just put the lambda-expression where you would normally put the function name. Example call:

```
( (lambda (lst) (< (length lst) 2))
  '(1 2 3 4) )
```

Lambda expressions can have multiple parameters (like regular named functions) and any sort of function body that a named function could have.

31

Exercise: What is the value of each of these Scheme expressions?

```
( (lambda (x) (cons x ())) 'y )
```

```
( (lambda (x y) (> (length x) (length y)))
  'a b c 'd )
```

```
( (lambda (x) (list? x)) '(lambda (x) (list? x)) )
```

```
( (lambda (x y) (append x y)) '(1 2) '(3 4 5) )
```

32

Using unnamed functions when calling prune

```
1 ]=> (define myList
      '(() (a b c) (1 2) () (()) (x (y w) z)))
;Value: myList
```

```
1 ]=> (prune myList (lambda (x) (not (null? x))))
;Value 4: ((a b c) (1 2) (()) (x (y w) z))
```

```
1 ]=> (define myList '((59 72 40) (85 70 88 56)))
;Value: myList
```

```
1 ]=> (prune myList (lambda (x) (> (apply min x) 50)))
;Value 5: ((85 70 88 56))
```

```
1 ]=> (define myList '((59 72 40) (85 70 88 56)))
;Value: myList
```

```
1 ]=> (prune myList (lambda (x) (> (apply min x) 50)))
;Value 6: ((85 70 88 56))
```

33

Example 3 (of passing functions):

bubblesort

; Precondition: *smaller?* is a function that can be
; applied to any two elements of *lst*. It should return
; #t iff the first argument is "smaller" than the second.

```
(define (bubblesort lst smaller?)
  (helper lst smaller? (- (length lst) 1))
)
```

; Bubblesorts the first *n* elements of *lst*. Returns a
; new list with the first *n* elements of *lst* sorted,
; followed by the rest of *lst* unchanged.
; Precondition: $n < (\text{length } \text{list})$.

```
(define (helper lst smaller? n)
  (if (= n 0)
      lst
      (helper (bubbleFirstN lst smaller? n)
              smaller?
              (- n 1)))
)
```

35

Why bother with lambda-expressions??

Example: Suppose we have tables of data (represented using Scheme lists), and functions that can do things like select out the rows of a given table that pass some test.

Suppose we want the *user* to be able to specify any criterion they might want. Examples:

- Retrieve students where $\text{gpa} > 3.0$
- Retrieve courses where $\text{classSize} < 100$
- Retrieve profs where $\text{building} = \text{SF}$

It would be tedious to write a named function for every single criterion that the user might specify.

Instead, we can have the *program* construct an appropriate lambda-expression, based on the user's query.

34

```
; Does a single "bubble run".
; Precondition:  $n < (\text{length } \text{lst})$ 
(define (bubbleFirstN lst smaller? n)
  (cond ((= n 0) lst)
        ((smaller? (car lst) (cadr lst))
         (cons (car lst)
               (bubbleFirstN (cdr lst)
                             smaller?
                             (- n 1))))
        (else (cons (cadr lst)
                    (bubbleFirstN (cons (car lst)
                                         (cddr lst))
                                  smaller?
                                  (- n 1))))))
)
```

Is our *bubblesort* function $O(n^2)$, where *n* is the length of the original list, as it should be?

Can you improve the style and/or comments?

36

Sample run of function bubblesort

```
eddie 1% scheme
Scheme Microcode Version ...

1 ]=> (load "sort.scm")

;Loading "sort.scm" -- done
;Value: bubblesort

1 ]=> (bubblesort '(3 4 1 5 0 2 3) <)

;Value 1: (0 1 2 3 3 4 5)

1 ]=> (bubblesort
      '((a b c) (a) (1 2 3 4) () (z z z) (y y))
      (lambda (x y) (< (length x) (length y))))

;Value 2: (() (a) (y y) (z z z) (a b c) (1 2 3 4))

1 ]=> (trace helper)
;No value

1 ]=> (trace bubbleFirstN)
;No value
```

; Note: #[compound-procedure ... fn] has been changed
; to #[fn] and the spacing has been reduced to make
; the slide more readable.

```
1 ]=> (bubblesort '(3 4 1 5 0 2 3) <)

[Entering #[helper]           Args: (3 4 1 5 0 2 3) #[<] 6]
[Entering #[bubblefirstn]    Args: (3 4 1 5 0 2 3) #[<] 6]
...
[(3 1 4 0 2 3 5)
 <== #[bubblefirstn]  Args: (3 4 1 5 0 2 3) #[<] 6]
[Entering #[helper]           Args: (3 1 4 0 2 3 5) #[<] 5]
[Entering #[bubblefirstn]    Args: (3 1 4 0 2 3 5) #[<] 5]
...
[(1 3 0 2 3 4 5)
 <== #[bubblefirstn]  Args: (3 1 4 0 2 3 5) #[<] 5]
[Entering #[helper]           Args: (1 3 0 2 3 4 5) #[<] 4]
[Entering #[bubblefirstn]    Args: (1 3 0 2 3 4 5) #[<] 4]
...
[(1 0 2 3 3 4 5)
 <== #[bubblefirstn]  Args: (1 3 0 2 3 4 5) #[<] 4]
[Entering #[helper]           Args: (1 0 2 3 3 4 5) #[<] 3]
[Entering #[bubblefirstn]    Args: (1 0 2 3 3 4 5) #[<] 3]
...
[(0 1 2 3 3 4 5)
 <== #[bubblefirstn]  Args: (1 0 2 3 3 4 5) #[<] 3]
[Entering #[helper]           Args: (0 1 2 3 3 4 5) #[<] 2]
[Entering #[bubblefirstn]    Args: (0 1 2 3 3 4 5) #[<] 2]
...
```

37

38

```
[(0 1 2 3 3 4 5)
 <== #[bubblefirstn]  Args: (0 1 2 3 3 4 5) #[<] 2]
[Entering #[helper]           Args: (0 1 2 3 3 4 5) #[<] 1]
[Entering #[bubblefirstn]    Args: (0 1 2 3 3 4 5) #[<] 1]
...
[(0 1 2 3 3 4 5)
 <== #[bubblefirstn]  Args: (0 1 2 3 3 4 5) #[<] 1]
[Entering #[helper]           Args: (0 1 2 3 3 4 5) #[<] 0]
[(0 1 2 3 3 4 5)
 <== #[helper]         Args: (0 1 2 3 3 4 5) #[<] 0]
[(0 1 2 3 3 4 5)
 <== #[helper]         Args: (0 1 2 3 3 4 5) #[<] 1]
[(0 1 2 3 3 4 5)
 <== #[helper]         Args: (0 1 2 3 3 4 5) #[<] 2]
[(0 1 2 3 3 4 5)
 <== #[helper]         Args: (1 0 2 3 3 4 5) #[<] 3]
[(0 1 2 3 3 4 5)
 <== #[helper]         Args: (1 3 0 2 3 4 5) #[<] 4]
[(0 1 2 3 3 4 5)
 <== #[helper]         Args: (3 1 4 0 2 3 5) #[<] 5]
[(0 1 2 3 3 4 5)
 <== #[helper]         Args: (3 4 1 5 0 2 3) #[<] 6]
;Value 3: (0 1 2 3 3 4 5)
```

Eval

eval is a built-in function of two arguments:
(*eval expr env*).

eval evaluates *expr* in the environment *env* and returns the result. We can use *'()* or (*the-environment*) to evaluate *expr* in the current environment.

Before *eval* is called, *expr* already gets evaluated once, so in effect *eval* cause *expr* to be evaluated twice.

Note: Dybvig describes a version of *eval* that precedes the current Scheme standard. The examples work if you add *()* as a second argument to *eval*.

39

40

Example: Write an expression, to go inside function *blah*, that will return the sum of the elements of *lst*.

```
(define (blah lst)
```

```
)
```

41

```
(define (myif test then myelse)
  (cond (test then)
        (else myelse))
)
```

```
(define (absolute num)
  (myif (> num 0)
        num
        (* num -1))
)
```

```
(define (len x)
  (myif (list? x)
        (length x)
        -1))
)
```

43

Redefining the if-statement in Scheme

Define function *myif*, without using the **if** syntactic form:

```
(myif (> num 0) ; If num is greater than 0
      num       ; then return num
      (* num -1) ; else return num * -1.
)
```

42

Problem: We shouldn't evaluate the arguments for *then* and *myelse* at the time of the call to *myif*.

Solution: Quote them.

```
(define (absolute num)
  (myif (> num 0)
        'num
        '(* num -1))
)
```

```
(define (len x)
  (myif (list? x)
        '(length x)
        '-1))
)
```

44

Problem: The arguments for *then* and *myelse* are echoed back to us, unevaluated.

Solution: Eval them inside *myif*.

```
(define (myif test then myelse)
  (cond (test (eval then ()))
        (else (eval myelse ())))
)
```

Problem: It may not make sense to evaluate the arguments for *then* and *myelse* in the context of *myif*.

```
(define (myif test then myelse env)
  (cond (test (eval then env))
        (else (eval myelse env)))
)
```

```
(define (len x)
  (myif (list? x)
        '(length x)
        '-1)
  (the-environment))
)
```