

## Part III: Functional Programming

### Pure Functional Languages

#### CSC 324: Principles of Programming Languages

#### Functional Programming

READING: Sebesta 14.1-14.5, 14.7-14.10

©Suzanne Stevenson 2001, revised by  
Diana Inkpen 2001

1

### Pure Functional Languages

2. The concept of assignment is **not** part of functional programming
  - no explicit assignment statements
  - variables bound to values only through the association of actual parameters to formal parameters in function calls
  - function calls have no side effects
  - thus no need to consider global state
3. Control flow is governed by function calls and conditional expressions
  - ⇒ no iteration
  - ⇒ recursion is widely used

3

Fundamental concept: **application** of (mathematical) **functions** to **values**

1. **Referential transparency:** The value of a function application is independent of the context in which it occurs
  - value of  $f(a,b,c)$  depends only on the values of  $f$ ,  $a$ ,  $b$  and  $c$
  - It does not depend on the global state of computation⇒ all vars in function must be parameters

2

### Pure Functional Languages

4. All storage management is implicit
  - needs garbage collection
5. Functions are *First Class Values*
  - Can be returned as the value of an expression
  - Can be passed as an argument
  - Can be put in a data structure as a value
  - Unnamed functions exist as values

4

A program includes:

1. A set of function definitions
2. An expression to be evaluated

E.g. in Scheme:

```
1 ]=> (define (abs-val x)
      (if (>= x 0)
          x
          (- x)))
```

```
;Value: abs-val
```

```
1 ]=> (abs-val (- 3 5))
```

```
;Value: 2
```

5

## COMMON LISP

- Implementations of LISP did not completely adhere to semantics
- Semantics redefined to match implementations
- COMMON LISP has become the standard
- Committee-designed language (1980s) to unify LISP variants
- Many defined functions
- Simple syntax, large language

7

## LISP

- Functional language developed by John McCarthy in the mid 50's
- Semantics based on *Lambda Calculus*
- All functions operate on lists or atomic symbols: (called "S-expressions")
- Only five basic functions: list functions `cons`, `car`, `cdr`, `equal`, `atom` and one conditional construct: `cond`
- Useful for list-processing applications
- Programs and data have the same syntactic form: S-expressions
- Used in Artificial Intelligence

6

## SCHEME

- Developed in 1975 by G. Sussman and G. Steele
- A version of LISP
- Consistent syntax, small language
- Closer to initial semantics of LISP
- Provides basic list processing tools
- Allows functions to be first class objects

8

## Commonalities between LISP and SCHEME

- Expressions are written in prefix, parenthesized form
  - (function arg<sub>1</sub> arg<sub>2</sub> ...arg<sub>n</sub>)
  - (+ 4 5)
  - (+ (\* 3 4 5) (- 5 3))
- In order to evaluate an expression:
  1. evaluate `function` to a function value
  2. evaluate each `argi` in order to obtain its value
  3. apply the function value to these values

9

## Built-In Functions

- `eq?`: identity on atoms
- `null?`: is list empty?
- `car`: selects first element of list
- `cdr`: selects rest of list
- `(cons element list)`: constructs lists by adding element to front of list
- `quote` or `'`: produces constants

11

## S-expressions

`<S-expression>` ::= `<Atom>` |  
                  `(' { <S-expression> }+ )'`  
`<Atom>` ::= `Sym` | `Num` | `#t` | `#f` | `'(')`

`#t`

`()`

`(a b c)`

`(a (b c) d)`

`((a b c) (d e (f)))`

`(1 (b) 2)`

Lists have nested structure.

10

## Built-In Functions

- `'()` is the empty list
- `(car '(a b c)) =`
- `(car '((a) b (c d))) =`
- `(cdr '(a b c)) =`
- `(cdr '((a) b (c d))) =`

12

## More about lists

- `car` and `cdr` can break up any list:
  - `(car (cdr (cdr '((a) b (c d)))))) =`
  - `(caddr '((a) b (c d)))`
- `cons` can construct any list:
  - `(cons 'a '()) =`
  - `(cons 'd '(e)) =`
  - `(cons '(a b) '(c d)) =`
  - `(cons '(a b c) '((a) b)) =`

13

## Other Functions

- `+` `-` `*` `/` numeric operators, e.g.,
  - `(+ 5 3) = 8`, `(- 5 3) = 2`
  - `(* 5 3) = 15`, `(/ 5 3) = 1.6666666`
- `=` `<` `>` `<=` `>=` number comparison ops
- Run-time type checking functions:
  - All return Boolean values: `#t` and `()`
  - `(number? 5)` is `#t`
  - `(zero? 0)` is `#t`
  - `(symbol? 'sam)` is `#t`
  - `(list? '(a b))` is `#t`
  - `(null? '())` is `#t`

15

Proper lists:

`()`, `(a (b (c) d) e)`  
`(cons 'a '(b)) → (a b)`

Dotted pairs (improper lists):

`(cons 'a 'b) → (a . b)`  
`(car '(a . b)) → a`  
`(cdr '(a . b)) → b`  
`(cons a '(b . c)) → (a b . c)`  
`(a b c) → (a . (b . (c . ())))`

14

## Other Functions

- `(number? 'sam)` evaluates to `()`
- `(null? '(a))` evaluates to `()`
- `(zero? (- 3 3))` evaluates to `#t`
- `(zero? '(- 3 3))` ⇒ type error
- `(list? (+ 3 4))` evaluates to `()`
- `(list? '(+ 3 4))` evaluates to `#t`

16

## READ-EVAL-PRINT Loop

**READ:** Read input from user:

a function application

**EVAL:** Evaluate input:

(f arg<sub>1</sub> arg<sub>2</sub> ...arg<sub>n</sub>)

1. evaluate f to obtain a function
2. evaluate each arg<sub>i</sub> to obtain a value
3. apply function to argument values

**PRINT:** Print resulting value:

the result of the function application

17

## READ-EVAL-PRINT Loop Example

```
1 ]=> (cons 'a (cons 'b '(c d)))
```

```
;Value 1: (a b c d)
```

1. Read the function application  
(cons 'a (cons 'b '(c d)))
2. Evaluate cons to obtain a function
3. Evaluate 'a to obtain a itself
4. Evaluate (cons 'b '(c d)):
  - (a) Evaluate cons to obtain a function
  - (b) Evaluate 'b to obtain b itself
  - (c) Evaluate '(c d) to obtain (c d) itself
  - (d) Apply the cons function to b and (c d) to obtain (b c d)
5. Apply the cons function to a and (b c d) to obtain (a b c d)
6. Print the result of the application:  
(a b c d)

18

## Quotes Inhibit Evaluation

;;Same as before:

```
1 ]=> (cons 'a (cons 'b '(c d)))
```

```
;Value 2: (a b c d)
```

;;Now quote the second argument:

```
1 ]=> (cons 'a '(cons 'b '(c d)))
```

```
;Value 3: (a cons (quote b) (quote (c d)))
```

;;Instead, un-quote the first argument:

```
1 ]=> (cons a (cons 'b '(c d)))
```

```
;Unbound variable: a
```

```
;To continue, call RESTART...
```

```
2 error> ^C^C
```

```
1 ]=>
```

19

## Quotes Inhibit Evaluation

;;Some things evaluate to themselves:

```
1 ]=> (list 1 42 #t #f ())
```

```
;Value 4: (1 2 #t () ())
```

;;They can also be quoted:

```
1 ]=> (list '1 '42 '#t '#f '())
```

```
;Value 5: (1 2 #t () ())
```

20

## READ-EVAL-PRINT Loop

Can also be used to define functions.

**READ:** Read input from user:

a symbol definition

**EVAL:** Evaluate input:

store function definition

**PRINT:** Print resulting value:

the symbol defined

Example:

```
1 ]=> (define (square x) (* x x))
```

```
;Value: square
```

21

## Function Definition

Two syntaxes for definition:

```
1. (define (<fcn-name> <fcn-params>)
    <expression>)
```

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

```
(define (mean x y)
  (/ (+ x y) 2))
```

22

## Function Definition

```
2. (define <fcn-name> <fcn-value>)
```

```
(define square
  (lambda (n) (* n n)))
```

```
(define mean
  (lambda (x y) (/ (+ x y) 2)))
```

Lambda calculus:

- A formal system for defining functions and their properties
- Equivalent to Turing machines

23

## Conditional Execution: if

```
(if <condition> <result1> <result2>)
```

1. Evaluate <condition>
2. If the result is a “true value” (i.e., anything but () or #f), then evaluate and return <result1>
3. Otherwise, evaluate and return <result2>

```
(define (abs-val x)
  (if (>= x 0) x (- x)))
```

```
(define (rest-if-first e lst)
  (if (eq? e (car lst)) (cdr lst) '()))
```

24

## Conditional Execution: cond

```
(cond (<condition1> <result1>)
      (<condition2> <result2>)
      ...
      (<conditionN> <resultN>)
      (else <else-result>) ;optional else
) ;clause
```

1. Evaluate conditions in order until obtaining one that returns a true value
2. Evaluate and return the corresponding result
3. If none of the conditions returns a true value, evaluate and return <else-result>

25

## Conditional versus Boolean Expressions

Write a function that takes a parameter *x* and returns `#t` if *x* is an atom, and `false` otherwise. Using `cond`:

```
(define (atom? x)
  (cond ((symbol? x) '#t)
        ((number? x) '#t)
        ((char? x) '#t)
        ((string? x) '#t)
        ((null? x) '#t)
        (else ()))
)
```

27

## Conditional Execution: cond

```
(define (abs-val x)
  (cond ((>= x 0) x)
        (else (- x))
  )
)

(define (rest-if-first e lst)
  (cond ((null? lst) '())
        ((eq? e (car lst)) (cdr lst))
        (else '())
  )
)
```

26

## Conditional versus Boolean Expressions

Now write `atom?` without using `cond`:

```
(define (atom? x)
  (if (symbol? x) '#t
      (if (number? x) '#t
          (if (char? x) '#t
              (if (string? x) '#t
                  (if (null? x) '#t ()))
              )
          )
      )
)
```

28

## Better atom? function

Any list is a pair (dotted pair with CAR and CDR), except the empty list (which is both list and atom).

```
(define (atom? x)
  (if (pair? x) () '#t)
)
```

```
(define (atom? x)
  (cond ((pair? x) ())
        (else '#t))
)
```

29

## Recursive Scheme Functions: Length

```
(define (length x)
  (if (null? x)
      0
      (+ 1 (length (cdr x))))
)
```

This is called “cdr-recursion.”

Note: There is a built-in `length` function.

31

## Recursive Scheme Functions: Sum-N

Parameter: integer  $n \geq 0$ .

Result: sum of integers from 0 to  $n$ .

```
(define (sum-n n)
  (cond ((= n 0) 0)
        (else (+ n (sum-n (- n 1))))
)
```

30

## Recursive Scheme Functions: Length

```
1 ]=> (trace length)
;No value
1 ]=> (length '(a b c))
[Entering #[compound-procedure 5 length]
  Args: (a b c)]
[Entering #[compound-procedure 5 length]
  Args: (b c)]
[Entering #[compound-procedure 5 length]
  Args: (c)]
[Entering #[compound-procedure 5 length]
  Args: ()]
[0
  <= #[compound-procedure 5 length]
  Args: ()]
[1
  <= #[compound-procedure 5 length]
  Args: (c)]
[2
  <= #[compound-procedure 5 length]
  Args: (b c)]
[3
  <= #[compound-procedure 5 length]
  Args: (a b c)]
;Value: 3
```

32

## Recursive Scheme Functions: Abs-List

- `(abs-list '(1 -2 -3 4 0)) ⇒ (1 2 3 4 0)`
- `(abs-list '()) ⇒ ()`

```
(define (abs-list lst)
```

```
)
```

33

## Recursive Scheme Functions: Counting

```
(define (atomcount x)
  (cond ((null? x) 0)
        ((atom? x) 1)
        (else (+ (atomcount (car x))
                  (atomcount (cdr x))))))
```

- `(atomcount '(1 2)) ⇒ 2`
- `(atomcount '(1 (2 (3)) (5))) ⇒ 4:`

```
(at '(1 (2 (3)) (5)))
(+ (at 1) (at ((2 (3)) (5))))
(+ 1 (+ (at (2 (3))) (at ((5))))))
(+ 1 (+ (+ (at 2) (at ((3)))) (+ (at 5) (at ())))))
(+ 1 (+ (+ 1 (+ (at 3) (at ()))) (+ (+ (at 5) (at ())) 0)))
(+ 1 (+ (+ 1 (+ (+ (at 3) (at ())) 0)) (+ (+ 1 0) 0)))
(+ 1 (+ (+ 1 (+ (+ 1 0) 0)) (+ 1 0)))
(+ 1 (+ (+ 1 (+ 1 0)) 1))
(+ 1 (+ (+ 1 1) 1))
(+ 1 (+ 2 1))
(+ 1 3)
```

4

This is called “car-cdr-recursion.”

35

## Recursive Scheme Functions: Append

```
(append '(1 2) '(3 4 5)) ⇒ (1 2 3 4 5)
(append '(1 2) '(3 (4) 5)) ⇒ (1 2 3 (4) 5)
(append '() '(1 4 5)) ⇒ (1 4 5)
(append '(1 4 5) '()) ⇒ (1 4 5)
(append '() '()) ⇒ ()
```

```
(define (append x y)
```

```
)
```

Note: There is a built-in `append` function.

34

## Efficiency Issues

**Problem:** Evaluating the same expression twice.

Example:

```
(define (longest-nonzero x y)
  (cond ((and (null? x) (null? y)) -1)
        ((> (length x) (length y))
         (length x))
        (else (length y))))
```

What can you do if there is no assignment statement?

36

## Efficiency Issues

**Solution 1:** Bind values to parameters in a helper function.

```
(define (maximum x y)
  (cond ((> x y) x)
        (else y)
  ))

(define (longest-nonzero x y)
  (cond ((and (null? x) (null? y)) -1)
        (else
         (maximum (length x) (length y))))
  ))
```

Note: There is a built-in `max` function.

37

## Polymorphic and Monomorphic Functions

- *Polymorphic* functions can be applied to arguments of many forms
- The function `length` is polymorphic: it works on lists of numbers, lists of symbols, lists of lists, lists of anything
- The function `square` is monomorphic: it only works on numbers

39

## Efficiency Issues

**Solution 2:** Use a `let` or `let*` construct, that binds variables to expression results.

```
(let ((var1 expr1)
      ...
      (varn exprn))
  <vars are defined and can be used here>)

(let* ((var1 expr1)
       ...
       (varn exprn))
  <vars are defined and can be used here>)
```

38

## Equality Checking

The `eq?` predicate doesn't work for lists.

Why not?

1. `(cons 'a '())` makes a new list
2. `(cons 'a '())` makes a(nother) new list
3. `eq?` checks if its two arguments are *the same*
4. `(eq? (cons 'a '()) (cons 'a '()))` evaluates to `()` (ie, `#f`)

Lists are stored as pointers to the first element (`car`) and the rest of the list (`cdr`).

Symbols are stored uniquely, so `eq?` works on them.

40

## Equality Checking for Lists

For lists, need a comparison function to check for the same **structure** in two lists

```
(define (equal? x y)
  (or (and (atom? x) (atom? y) (eq? x y))
      (and (not (atom? x)) (not (atom? y))
            (equal? (car x) (car y))
            (equal? (cdr x) (cdr y)))))
```

- (equal? 'a 'a) evaluates to #t
- (equal? 'a 'b) evaluates to ()
- (equal? '(a) '(a)) evaluates to #t
- (equal? '((a)) '(a)) evaluates to ()

41

## Higher-Order Functions

Functions as returned values:

```
(define (plus-list x)
  (cond ((number? x)
        (lambda (y) (+ (sum-n x) y)))
        ((list? x)
        (lambda (y) (+ (sum-list x) y)))
        (else (lambda (x) x))))
```

```
1 ]=> ((plus-list 3) 4)
;Value: 10
```

```
1 ]=> ((plus-list '(1 3 5)) 5)
;Value: 14
```

43

## Higher-Order Functions

Functions as input values:

```
(define (all-num lst)
  (or (null? lst)
      (and (number? (car lst))
            (all-num (cdr lst)))))
```

```
(define (all-num-f f lst)
  (cond ((all-num lst) (f lst))
        (else 'error)))
```

```
1 ]=> (all-num-f abs-list '(1 -2 3))
;Value 1: (1 2 3)
```

```
1 ]=> (all-num-f abs-list '(1 a))
;Value: error
```

42

### Built-In Higher-Order Functions: map

```
(define (map f l)
  (cond ((null? l) '())
        (else (cons (f (car l))
                      (map f (cdr l)))))
```

- map takes two arguments: a function and a list
- map builds a new list whose elements are the result of applying the function to each element of the (old) list

44

## What's Wrong Here??

### Higher-order Functions: map

- Example:

```
(map abs '(-1 2 -3 4)) ⇒  
(1 2 3 4)
```

```
(map (lambda (x) (+ 1 x)) '(-1 2 -3)) ⇒  
(0 3 -2)
```

- Actually, the built-in map can take more than two arguments:

```
(map cons '(a b c) '((1) (2) (3))) ⇒  
((a 1) (b 2) (c 3))
```

45

```
1 ]=>  
(define (atomcount s)  
  (cond ((null? s) 0)  
        ((atom? s) 1)  
        (else (+ (map atomcount s))))  
  )  
;Value: atomcount  
1 ]=> (atomcount '(a b))
```

```
;The object (1 1), passed as an argument  
;to +, is not the correct type.
```

```
...  
2 error>
```

### Why doesn't this work?

46

### Using eval to Correct the Problem

```
(define (atomcount s)  
  (cond ((null? s) 0)  
        ((atom? s) 1)  
        (else  
         (eval  
          (cons '+ (map atomcount s)) '())))  
  )  
1 ]=> (atomcount '(a b))  
;Value: 2  
  
1 ]=> (atomcount '((1) (2 3 (4)) (((5))))))  
;Value: 5
```

47

### Limitations of Using eval

**BUT:** eval only works in the current definition of atomcount because numbers evaluate to themselves.

```
1 ]=> (+ 1 2 3)  
;Value: 6  
  
1 ]=> (cons '+ '(1 2 3))  
;Value 12: (+ 1 2 3)  
  
1 ]=> (eval (cons '+ '(1 2 3)) '())  
;Value: 6
```

48

## Using eval to Evaluate Expressions

```
1 ]=> (append '(a) '(b))
;Value 13: (a b)
1 ]=> (cons 'append '((a) (b)))
;Value 14: (append (a) (b))

1 ]=> (eval (cons 'append '((a) (b))) '())
;Unbound variable: b
...
1 ]=> (cons 'append '( '(a) '(b) ))
;Value 15: (append (quote (a)) (quote (b)))

1 ]=> (eval
      (cons 'append '( '(a) '(b))) '())
;Value 16: (a b)
```

**Too complicated!!**

49

## Applying Functions with apply

```
1 ]=> (apply + '(1 2 3))
;Value: 6
1 ]=> (apply append '((a) (b)))
;Value 5: (a b)

1 ]=>
(define (atomcount s)
  (cond ((null? s) 0)
        ((atom? s) 1)
        (else
         (apply + (map atomcount s)))))

;Value: atomcount
1 ]=> (atomcount '(a (b) c))
;Value: 3
```

50

## Higher-order Functions: reduce

```
(define (reduce op l id)
  (if (null? l)
      id
      (op (car l)
          (reduce op (cdr l) id)))
))
```

A binary  $\mapsto$  n-ary function.

The `reduce` function takes a binary operation and applies it right-associatively to a list of an arbitrary number of arguments.

**NOTE:** `reduce` is not equivalent to `apply`.

51

## Higher-order Functions: reduce

```
(reduce + '(1 2 3) 0)  $\Rightarrow$  6:
(reduce + '(1 2 3) 0)
(+ 1 (reduce + '(2 3) 0))
(+ 1 (+ 2 (reduce + '(3) 0)))
(+ 1 (+ 2 (+ 3 (reduce + '() 0))))
(+ 1 (+ 2 (+ 3 0)))
6
```

**Note:**  $(+ 1 2 3) \Rightarrow 6$

```
(reduce / '(24 6 2) 1)  $\Rightarrow$  8:
(reduce / '(24 6 2) 1)
(/ 24 (reduce / '(6 2) 1))
(/ 24 (/ 6 (reduce / '(2) 1)))
(/ 24 (/ 6 (/ 2 (reduce / '() 1))))
(/ 24 (/ 6 (/ 2 1)))
8
```

**Note:**  $(/ 24 6 2) \Rightarrow 2$

52

## Higher-order Functions: reduce

Given `union`, which takes two lists representing sets and returns their union:

```
1 ]=> (apply union '((1 3)(2 3 4)))
;Value 21: (1 2 3 4)
```

```
1 ]=> (apply union '((1 3)(2 3)(4 5)))
;The procedure #[compound-procedure union]
;has been called with 3 arguments;
;it requires exactly 2 arguments.
```

```
1 ]=> (reduce union '((1 3)(2 3)(4 5)) '())
;Value 22: (1 2 3 4 5)
```

**Question:** How would you have to change `reduce` to be able to take `intersection` as its function argument?

53

## More on Efficiency

```
(define (reverse lst)
```

```
)
```

55

## Example Functions

- `cdrLists`: given a list of lists, form new list giving all elements of the `cdr`'s of the sublists.  
 $((1\ 2)\ (3\ 4\ 5)\ (6)) \Rightarrow (2\ 4\ 5)$
- `swapFirstTwo`: given a list, swap the first two elements of the list.  
 $(1\ 2\ 3\ 4) \Rightarrow (2\ 1\ 3\ 4)$
- `swapTwoInLists`: given a list of lists, form new list of all elements in all lists, with first two of each swapped.  
 $((1\ 2\ 3)(4)(5\ 6)) \Rightarrow (2\ 1\ 3\ 4\ 6\ 5)$
- `addSums`: given a list of numbers, sum the total of all sums from 0 to each number.  
 $(1\ 3\ 5) \Rightarrow 22$

54

## More on Efficiency

Using an **accumulator**.

```
(define (reverse lst new)
```

```
)
```

56

## Example Functions

- `addToEnd`: add an element to the end of a list.

`(addToEnd 'a '(a b c)) ⇒ (a b c a)`

- `revLists`: given a list of lists, form new list consisting of all elements of the sublists in reverse order.

`((1 2) (3 4 5) (6)) ⇒ (6 5 4 3 2 1)`

- `revListsAll`: given a list of lists, form new list from reversal of elements of each list.

`((1 2) (3 4 5) (6)) ⇒ (2 1 5 4 3 6)`

57

## Lambda Expressions: Environments and Local Variables

Lambda variables get values by the process of **lambda reduction**: when a lambda expression is followed by a sequence of expressions, the values of those expressions are substituted for the lambda variables.

`((lambda (x y) (* x y)) 3 (4 + 5) )`

⇒ [by lambda reduction]

`( * 3 9 )`

⇒ [by simplification]

27

59

## Lambda Expressions: Environments and Local Variables

Recall that functions are **lambda expressions**:

`(define (mult x y) (* x y)) ≡`

`(define mult (lambda (x y) (* x y)))`

Lambda expressions are a formal notation for establishing an **environment** (a local context) in which the lambda variables (the parameters to the function) are defined.

Compare logic expressions:

$\forall x (P(x) \Rightarrow Q(x))$

58

## Function Calls and Lambda Reduction

A function call in Scheme is **lambda reduction**:

`> (mult 3 (+ 4 5))`

⇒ [by evaluation]

`((lambda (x y) (* x y)) 3 9 )`

⇒ [by **lambda reduction**]

`( * 3 9 )`

⇒ [by simplification]

> 27

60

## let and let\*

The special forms `let` and `let*` are used to define local variables in a new environment.

General Form:

```
(let ((v1 e1) (v2 e2) ... (vn en)) expr)
(let* ((v1 e1) (v2 e2) ... (vn en)) expr)
```

Both establish the variables  $v_1, \dots, v_n$  to have values  $e_1, \dots, e_n$  in the expression `expr`

- `let` does the binding in parallel
- `let*` does the binding in order

61

## let and let\* are not primitive

```
(let ((v1 e1)...(vn en)) expr)
```

⇕

```
((lambda (v1...vn) expr) e1...en)
```

AND

```
(let* ((v1 e1) (v2 e2)) expr)
```

⇕

```
((lambda (v1) ((lambda (v2) expr) e2)) e1)
```

All binding of values to variables is by parameter passing ( $\equiv$  lambda reduction):

$\Rightarrow$  **NO ASSIGNMENT!**

63

## let and let\*

```
(let ((x 2)) (* x x))
```

$\Rightarrow$  4

```
(let ((x 4)) (let ((y (+ x 2))) (* x y)))
```

$\Rightarrow$  24

```
(let ((x 4) (y (+ x 2))) (* x y))
```

is an error: unbound variable x

```
(let* ((x 4) (y (+ x 2))) (* x y))
```

$\Rightarrow$  24

62

## Summary of Part III: Functional Programming Languages

- Pure functional languages:
  - Referential transparency
  - No assignment
  - No iteration, only recursion
  - Implicit storage management (garbage collection)
  - Functions are values
- LISP, Common LISP, Scheme
- S-expressions
- Read-eval-print loop
- Inhibiting evaluation
- Function definition and lambda expressions
- Conditionals
- Recursive functions
- Polymorphism
- Atomic and structural equality
- Higher-order functions
- Lambda reduction and local environments

64