Functional Programming Languages

BBM 301 – Programming Languages

Introduction

- The design of the imperative languages is based directly on the *von Neumann architecture*
  - Efficiency is the primary concern, rather than the suitability of the language for software development
- The design of the functional languages is based on *mathematical functions*
  - A solid theoretical basis that is also closer to the user, but relatively unconcerned with the architecture of the machines on which programs will run

Mathematical Functions

- A mathematical function is a *mapping* of members of one set, called the *domain set*, to another set, called the *range set*
- In math functions, the evaluation order is controlled by recursion
- They don’t have side effects: same value given the same arguments

Lambda Expressions

- Lambda expressions describe nameless functions
- A *lambda expression* specifies the parameter(s) and the mapping of a function in the following form

  `λ(x) x * x * x`

  for the function `cube (x) = x * x * x`
- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression

  e.g., `(λ(x) x * x * x)(2)`

  which evaluates to 8

Functional Forms

- A higher-order function, or *functional form*, is one that either takes functions as parameters or yields a function as its result, or both

Function Composition

- A functional form that takes two functions as parameters and yields a function whose value is the first actual parameter function applied to the application of the second

  Form: `h ≡ f ° g`

  which means `h(x) ≡ f(g(x))`

  For `f(x) ≡ x + 2` and `g(x) ≡ 3 * x`,

  `h ≡ f ° g` yields `(3 * x)+ 2`

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) ≡ x * x`

  α`( h, (2, 3, 4))` yields `(4, 9, 16)`

Fundamentals of Functional Programming Languages

- The objective of the design of a FPL is to mimic mathematical functions to the greatest extent possible
- The basic process of computation is fundamentally different in a FPL than in an imperative language
  - In an imperative language, operations are done and the results are stored in variables for later use
  - Management of variables is a constant concern and source of complexity for imperative programming
- In an FPL, variables are not necessary, as is the case in mathematics

Fundamentals of Functional Programming Languages (cont'd.)

- *Referential Transparency* - In an FPL, the evaluation of a function always produces the same result given the same parameters
- *Tail Recursion* – Writing recursive functions that can be automatically converted to iteration

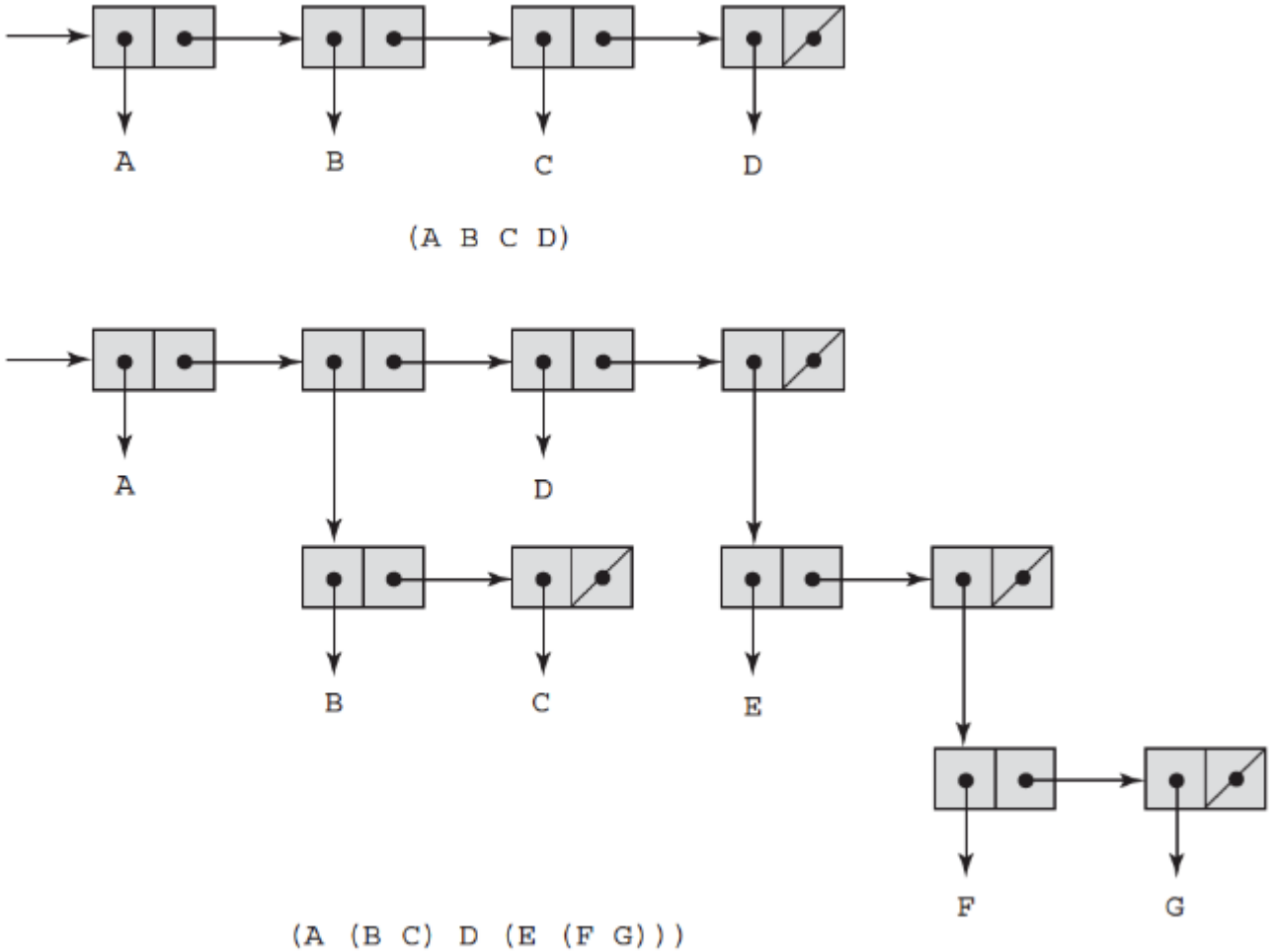
LISP Data Types and Structures

- LISP was developed by John McCarthy at MIT in 1959
- *Data object types*: originally only atoms and lists
- *List form*: parenthesized collections of sublists and/or atoms

  e.g., `(A B (C D) E)`
- Originally, LISP was a typeless language
- LISP lists are stored internally as single-linked lists

Figure 15.1

Internal representation of two LISP lists



LISP Interpretation

- Lambda notation is used to specify functions and function definitions.
- **Function applications and data have the same form**.

  e.g., If the list `(A B C)` is interpreted as data it is

  a simple list of three atoms, `A`, `B`, and `C`
- *If it is interpreted as a function application, it means that the function named `A` is applied to the two parameters, `B` and `C`*

Origins of Scheme

- A mid-1970s dialect of LISP, designed to be a cleaner, more modern, and simpler version than the contemporary dialects of LISP
- Uses only static scoping
- Functions are first-class entities
  - They can be the values of expressions and elements of lists
  - They can be assigned to variables, passed as parameters, and returned from functions

The Scheme Interpreter

- In interactive mode, the Scheme interpreter is an infinite read-evaluate-print loop (REPL)
  - This form of interpreter is also used by Python and Ruby
- Expressions are interpreted by the function `EVAL`
- Literals evaluate to themselves

Primitive Function Evaluation

- Parameters are evaluated, in no particular order
- The values of the parameters are substituted into the function body
- The function body is evaluated
- The value of the last expression in the body is the value of the function

Primitive Functions

- Primitive Arithmetic Functions: `+, -, *, /, ABS, SQRT, REMAINDER, MIN, MAX`

  e.g., `(+ 5 2) yields 7`

Examples

Expression	Value
`42`	42
`(* 3 7)`	21
`(+ 5 7 8)`	20
`(- 5 6)`	-1
`(- 15 7 2)`	6
`(- 24 (* 4 3))`	12

Function Definition: `LAMBDA`

- Lambda Expressions
  - Form is based on λ notation

  e.g., `(LAMBDA (x) (* x x))`

  x is called a bound variable
- Lambda expressions can be applied to parameters

  e.g., `((LAMBDA (x) (* x x)) 7)`
- `LAMBDA` expressions can have any number of parameters

```
(LAMBDA (a b x) (+ (* a x x) (* b x)))

(LAMBDA (a b c x) (+ (* a x x) (* b x) c))
```

Special Form Function: `DEFINE`

- A Function for constructing functions: `DEFINE` - Two forms:

1. To bind a symbol to an expression

   e.g., `(DEFINE pi 3.141593)`

   Example use: `(DEFINE two_pi (* 2 pi))`

- The evaluation process for `DEFINE` is different! The first parameter is never evaluated. The second parameter is evaluated and bound to the first parameter.

Special Form Function: DEFINE

- 2. The second use of the DEFINE function is to bind a lambda expression to a name.
- To bind a name to a lambda expression, DEFINE takes two lists as parameters. The first parameter is the prototype of a function call, with the function name followed by the formal parameters, together in a list. The second list contains an expression to which the name is to be bound.
- The general form of such a DEFINE is

Special Form Function: DEFINE

```
(DEFINE (function_name parameters)
(expression)
)
```

e.g., `(DEFINE (square x) (* x x))`

Example use: `(square 5)`

```
(DEFINE (hypotenuse side1 side2)
  (SQRT(+(square side1)(square side2)))
  )
```

Output Functions

- `(DISPLAY expression)`
- `(NEWLINE)`

Numeric Predicate Functions

- `#T` (or `#t`) is true and `#F` (or `#f`) is false (sometimes `()` is used for false)
- `=, <>, >, <, >=, <=`
- `EVEN?, ODD?, ZERO?, NEGATIVE?`
- The `NOT` function inverts the logic of a Boolean expression

Control Flow: `IF`

- Selection- the special form, `IF`

  `(IF predicate then_exp else_exp)`

  e.g.,

```
(IF (not (= count 0))
    (/ sum count)
    0)
```

Control Flow: COND

- Multiple Selection - the special form, COND
  General form:

```
(COND
    (predicate_1  expr {expr})
    (predicate_2  expr {expr})
    ...
    (predicate_n  expr {expr})
    (ELSE expr {expr})
)
```

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

Example of COND

```
(DEFINE (compare x y)
   (COND
     ((> x y) “x is greater than y”)
     ((< x y) “y is greater than x”)
     (ELSE “x and y are equal”)
   )
 )
```

Example of COND

Ex: Leap year: Every **year** that is exactly divisible by four is a **leap year**, except for years that are exactly divisible by 100, but these centurial years are **leap** years if they are exactly divisible by 400.

```
(DEFINE (leap? year)
(COND
((ZERO? (MODULO year 400)) #T)
((ZERO? (MODULO year 100)) #F)
(ELSE (ZERO? (MODULO year 4)))
)
)
```

Example

```
(DEFINE (factorial n)
(IF (<= n 1)
  1
  (* n (factorial (- n 1)))
)
)
```

Function QUOTE

- `QUOTE` - takes one parameter; returns the parameter without evaluation
  - `QUOTE` is required because the Scheme interpreter, named `EVAL`, always evaluates parameters to function applications before applying the function. `QUOTE` is used to avoid parameter evaluation when it is not appropriate
  - `QUOTE` can be abbreviated with the apostrophe prefix operator

    `'(A B) is equivalent to (QUOTE (A B))`

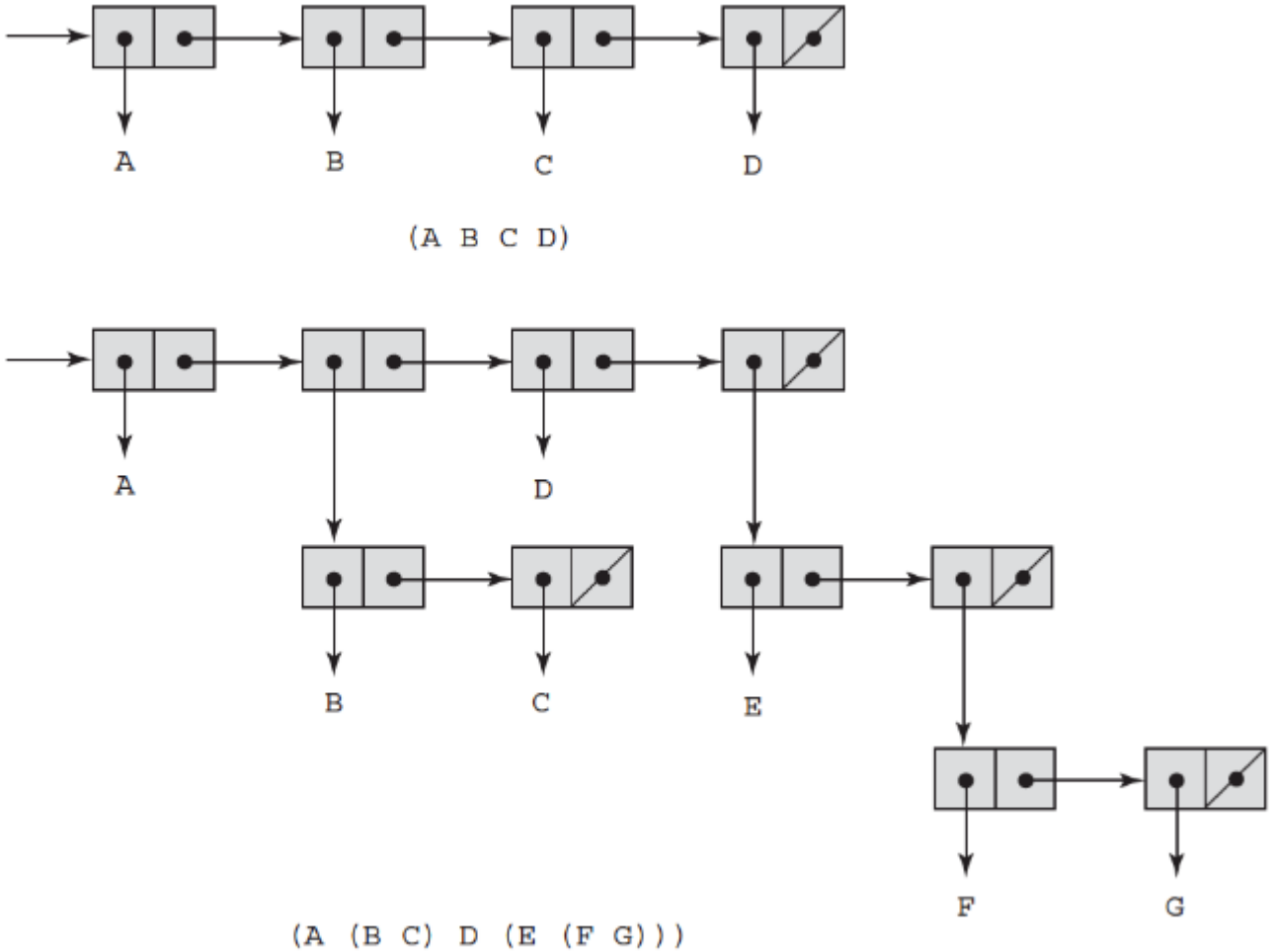
List Functions: CAR and CDR

- `CAR` takes a list parameter; returns the first element of that list
  e.g., `(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
  e.g., `(CDR '(A B C))` yields `(B C)`
  `(CDR '((A B) C D))` yields `(C D)`

Figure 15.1

Internal representation of two LISP lists



List Functions: CAR and CDR

- `(DEFINE (second a_list) (CAR (CDR a_list)))`

Once this function is evaluated, it can be used, as in

`(second '(A B C))  = returns B`

- Some of the most commonly used functional compositions in Scheme are built in as single functions.

```
(CAAR x)  = (CAR(CAR x))
(CADR x) = (CAR (CDR x))
(CADDAR x)   = (CAR (CDR (CDR (CAR x)))).

(CADDAR '((A B (C) D) E)) = (C)
```

- (CADDAR '((A B (C) D) E)) =
- (CAR (CDR (CDR (CAR '((A B (C) D) E))))) =
- (CAR (CDR (CDR '(A B (C) D)))) =
- (CAR (CDR '(B (C) D))) =
- (CAR '((C) D)) =
- (C)

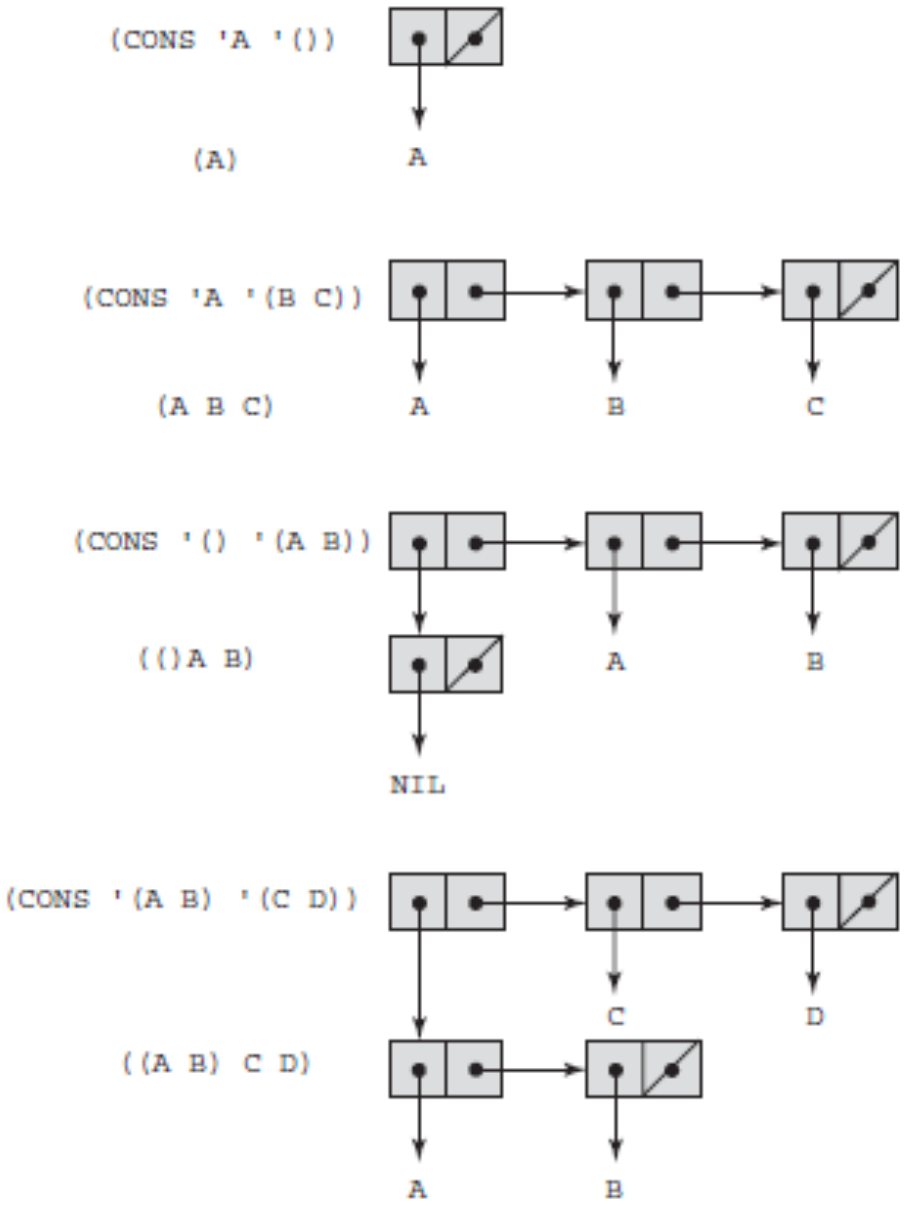
List Functions: CONS and LIST

- CONS takes two parameters, the first of which can be either an atom or a list and the second of which is a list; returns a new list that includes the first parameter as its first element and the second parameter as the remainder of its result

  e.g., `(CONS 'A '(B C))` returns `(A B C)`

Figure 15.2

The result of several CONS operations



- (CONS 'A '()) returns (A)
- (CONS 'A '(B C)) returns (A B C)
- (CONS '() '(A B)) returns (() A B)
- (CONS '(A B) '(C D)) returns ((A B) C D)

- (CONS (CAR a_list) (CDR a_list))

- BE CAREFUL (CONS 'A 'B) -> (A . B)

List Functions: CONS and LIST

- `LIST` takes any number of parameters; returns a list with the parameters as elements

  e.g. `(LIST 'apple 'orange 'grape)` returns

  `(apple orange grape)`

  (CONS 'apple (CONS 'orange (CONS 'grape '())))

(list 'A '(B C)) => (A (B C))

(list 'A '()) => (A ())

(list '() '(A B)) => (() (A B))

(list '(A B) '(C D)) => ((A B) (C D))

Predicate Function: EQ?

- `EQ?` takes two symbolic parameters; it returns `#T` if both parameters are atoms and the two are the same; otherwise `#F`

  e.g., `(EQ? 'A 'A)` yields `#T`

  `(EQ? 'A 'B)` yields `#F`

  - Note that if `EQ?` is called with list parameters, the result is not reliable
  - Also `EQ?` does not work for numeric atoms

- (EQ? 'A 'A) returns #T
- (EQ? 'A 'B) returns #F
- (EQ? 'A '(A B)) returns #F
- (EQ? '(A B) '(A B)) returns #F or #T
- (EQ? 3.4 (+ 3 0.4)) returns #F or #T

Predicate Function: EQV?

- `EQV?` is like `EQ?`, except that it works for both symbolic and numeric atoms; it is a value comparison, not a pointer comparison

  `(EQV? 3 3)` yields `#T`

  `(EQV? 'A 3)` yields `#F`

  `(EQV? 3.4 (+ 3 0.4))` yields `#T`

  `(EQV? 3.0 3)` yields `#F` (floats and integers are different)

Predicate Functions: `LIST?` and `NULL?`

- `LIST?` takes one parameter; it returns `#T` if the parameter is a list; otherwise `#F`

`(LIST? '())` yields `#T`

(LIST? '(X Y)) returns #T

(LIST? 'X) returns #F

- `NULL?` takes one parameter; it returns `#T` if the parameter is the empty list; otherwise `#F`
  - Note that `NULL?` returns `#T` if the parameter is `()`
  - `(NULL? '(()))` yields `#F`
  - (NULL? '(A B)) returns #F
  - (NULL? '()) returns #T
  - (NULL? 'A) returns #F

Example Scheme Function: `member`

- `member` takes an atom and a simple list; returns `#T` if the atom is in the list; `#F` otherwise

```
(DEFINE (member atm lis)
(COND
    ((NULL? lis) #F)
    ((EQ? atm (CAR lis)) #T)
    (ELSE (member atm (CDR lis))
)
)
```

(member 'B '(A B C)) returns #T
(member 'B '(A C D E)) returns #F

```
(define (member? atm lis)
  (cond
    ((null? lis) #f)
    ( (eq? atm (car lis)) #t)
    ( else (member? atm (cdr lis))
   )
 )
```

(member? 'a '()) => #f

(member? 'a '(a b)) => #t

(member? 'b '(a b)) => #t → **(member? ‘b (cdr ‘(a b)) ->**
(member? ‘b ‘(b))

(member? 'a '(b c)) => #f

Example Scheme Function: `equalsimp`

- `equalsimp` takes two simple lists as parameters; returns `#T` if the two simple lists are equal; `#F` otherwise

```
(DEFINE (equalsimp lis1 lis2)
(COND
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) #F)
    ((EQ? (CAR lis1) (CAR lis2))
         (equalsimp(CDR lis1)(CDR lis2)))
    (ELSE #F)
))
```

```
(define (equalsimp lis1 lis2)
  (cond
    ((null? lis1) (null? lis2))
    ((null? lis2) #f)
    ((eq? (car lis1) (car lis2))
      (equalsimp(cdr lis1)(cdr lis2)))
    (else #f)
  ))
```

(equalsimp '() '()) => #t
(equalsimp '() '(a)) => #f
(equalsimp '(a) '()) => #f
(equalsimp '(a) '(a)) => #t
(equalsimp '(a) '(b)) => #f
equalsimp '(a) '(a b)) => #f
(equalsimp '(a b) '(a b)) => #t

(equalsimp '(a) '(a)) -> (equalsimp '() '())

(equalsimp ‘(a b) ‘(a c))
-> (equalsimp ‘(b) ‘(c))

(equalsimp '(a b) '(a c)) => #f

Example Scheme Function: `equal`

- `equal` takes two general lists as parameters; returns `#T` if the two lists are equal; `#F` otherwise

```
(DEFINE (equal lis1 lis2)
  (COND
   ((NOT (LIST? lis1))(EQ? lis1 lis2))
   ((NOT (LIST? lis2)) #F)
   ((NULL? lis1) (NULL? lis2))
   ((NULL? lis2) #F)
   ((equal (CAR lis1) (CAR lis2))
        (equal (CDR lis1) (CDR lis2)))
   (ELSE #F)
))
```

```
(define (equal lis1 lis2)
  (cond
   ((not (list? lis1))(eq? lis1 lis2))
   ((not (list? lis2)) #f)
   ((null? lis1) (null? lis2))
   ((null? lis2) #f)
   ((equal (car lis1) (car lis2))
     (equal (cdr lis1) (cdr lis2)))
   (else #f)
 ))
```

(equal ' ((a b) c) '((a b) c) => #t

(equal ' ((a b) c) '((a b) c) -> (equal ‘(a b) ‘(a b)) (equal ‘(c) ‘(c)

(equal ‘a ‘a) (equal ‘(b) ‘(b) (equal ‘c ‘c) (equal ‘() ‘()

#t (equal ‘b ‘b) (equal ‘() ‘()) #t #t

#t #t

Example Scheme Function: 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)))
))
```

(append '(A B) '(C D R)) returns (A B C D R)

(append '((A B) C) '(D (E F))) returns ((A B) C D (E F))

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

```
(append ‘( a b c) ‘ (d e))
        (cons ’a (append ‘(b c) ‘(d e)) )
        (cons ‘a (cons ‘b (append ‘(c ) ‘( d e)) ))
        (cons ‘a (cons ‘b (cons ‘c (append ‘() ‘( d e)) )))

        (cons ‘a (cons ‘b (cons ‘c ‘( d e) )))
        (cons ‘a (cons ‘b ’(c d e) ))
        (cons ‘a ’(b c d e) )
        ’(a b c d e)
```

(append '(a b) '(a b)) => (a b a b)

(list '(a b) '(a b)) => ((a b) (a b))

(cons '(a b) '(a b)) => ((a b) a b)

Exercise

```
(define (guess list1 list2)
(cond
((null? list1) '())
((null? list2) '())
((eq? (car list1) (car list2))
  (cons (car list1) (guess (cdr list1) (cdr list2))))
(else (guess (cdr list1) (cdr list2))
)))
```

Example Scheme Function: `LET`

- General form:

```
(LET (
    (name_1 expression_1)
    (name_2 expression_2)
    ...
    (name_n expression_n))
   body
)
```

- Evaluate all expressions, then bind the values to the names; evaluate the body
- Each binding of a <variable> has <body> as its region.

LET examples

```
(let ((x 2) (y 3))
(* x y))
```

⇒6

```
(let ((x 3) (y (+ x 1)))
(+ x y))
```

⇒Error : unbound symbol x

Let example

```
(let ((x 2) (y 3))
     (let ((x 7) (z (+ x y)))
     (* z x))
)
```

=> 35

Let*

- Let* is similar to let, but the bindings are performed sequentially from left to right, and the region of a binding indicated by (<variable> <init>) is that part of the let* expression to the right of the binding. Thus the second binding is done in an environment in which the first binding is visible, and so on.

```
(let ((x 2) (y 3))
    (let* ((x 7) (z (+ x y)))
(* z x)))
=> 70
```

LET Example

```
(DEFINE (quadratic_roots a b c)
  (LET (
    (root_part_over_2a
      (/ (SQRT (- (* b b) (* 4 a c)))(* 2 a)))
    (minus_b_over_2a (/ (- 0 b) (* 2 a)))
  (DISPLAY (+ minus_b_over_2a root_part_over_2a))
  (NEWLINE)
  (DISPLAY (- minus_b_over_2a root_part_over_2a))
))
```

Tail Recursion in Scheme

- Definition: A function is *tail recursive* if its recursive call is the last operation in the function
- A tail recursive function can be automatically converted by a compiler to use iteration, making it faster
- Scheme language definition requires that its language systems convert all tail recursive functions to use iteration

Tail Recursion Example

```
(define (my-last lst)
(if (null? (cdr lst))
     (car lst)
     (my-last (cdr lst))))
```

(my-last '(1 2 3)) ⇒ 3

Tail Recursion in Scheme

the member function is tail recursive

```
(DEFINE (member atm a_list)
(COND
      ((NULL? a_list) #F)
      ((EQ? atm (CAR a_list)) #T)
(ELSE (member atm (CDR a_list)))
))
```

This function can be automatically converted by a compiler to use iteration, resulting in faster execution than in its recursive form

Tail Recursive Example

Is the following function tail recursive ?

```
(define (list-sum lis)
      (if (null? lis)
            0
            (+ (car lis) (list-sum (cdr lis)))
      )
)
```

The problem here is control must always return so that the actual call can be done.

Writing Tail Recursive Functions

- We can write a tail-recursive version of `list-sum` that adds things in front-to-back order instead. The trick is to do the addition before the tail call, and to *pass the sum so far to the recursive call*.
- To do this, we have to keep a running sum, and each recursive call must pass it as an argument to the next. To start it off, we have to have a "running sum" of 0.

Not Tail Recursive:

```
(define (list-sum lis)
      (if (null? lis)
            0
            (+ (car lis) (list-sum (cdr lis)))
      )
)
```

Tail Recursive:

```
(define (loop lis sum-so-far)
      (cond ((null? lis) sum-so-far)
      (else (loop (cdr lis) (+ sum-so-far (car lis))))))

 ;; a wrapper that supplies an initial running sum of 0
(define (list-sum lis)
   (loop lis 0))
```

Tail Recursion in Scheme (cont'd.)

- Example of rewriting a function to make it tail recursive, using helper a function

Original:

```
(DEFINE (factorial n)
    (IF (= n 0)
       1
       (* n (factorial (- n 1)))
 ))
```

you can use an *accumulator* -- an additional parameter to a function that accumulates the answer -- to convert a non-tail recursive function into a tail recursive one.

Tail recursive:

```
(DEFINE (facthelper n factpartial)
    (IF (= n 0)
       factpartial
       (facthelper (- n 1) (* n factpartial)))
  ))
 (DEFINE (factorial n)
    (facthelper n 1))
```

Functional Form - Composition

- If `h` is the composition of `f` and `g`, `h(x) = f(g(x))`

```
(DEFINE (g x) (* 3 x))
(DEFINE (f x) (+ 2 x))
(DEFINE h x) (+ 2 (* 3 x))) (The composition)
```

- In Scheme, the functional composition function `compose` can be written:

```
(DEFINE (compose f g) (LAMBDA (x) (f (g x))))
```

Remember our discussion on CADR, CDDR, etc.

`((compose CAR CDR) '((a b) c d))` yields `c`

Write the alternative of CDAR using compose

```
((compose CDR CAR) '((a b) c d))
```

Write the function to return the third element of a list

```
(DEFINE (third a_list)
        ((compose CAR (compose CDR CDR)) a_list))
```

is equivalent to `CADDR`

Functional Form – Apply-to-All

- Apply to All - one form in Scheme is `map`
  - Applies the given function to all elements of the given list;

```
(DEFINE (map fun lis)
  (COND
    ((NULL? lis) ())
    (ELSE (CONS (fun (CAR lis))
                (map fun (CDR lis))))
))

(map (lambda (num) (* num num num)) '(3 4 2 6))
```

yields `(27 64 8 216)`

Functional Form – Apply-to-All

```
(map cdr '((a b c) (d e) (f g)))
=> ((b c) (e) (g))
(map round '(3 3.3 4.6 5))
⇒(3 3 5 5)
(map null? '(3 () () 5))
⇒(#f #t #t #f)

(map + '(1 2 3) '(10 11 12))
=> (11 13 15)
(map (lambda (x y) (list x y)) '(a b c) '(j k l))
=> ((a j) (b k) (c l))
```

Nested Scopes

- You can have arbitrarily nested scopes (scopes within scopes within scopes ...). Further, since function names are bound like any other variable, function names also obey the scope rules.

```
(define x 100)
(define y 200)
(define (simple y) (+ y 300))

(define (myfunc x)
(let ((simple (lambda (y) (+ y 10))))
(simple x)))
```

(myfunc 18) => 28

(simple 18) => 318

Functions That Build Code

- It is possible in Scheme to define a function that builds Scheme code and requests its interpretation

- This is possible because the interpreter is a user-available function, `EVAL`

Adding a List of Numbers

- Suppose that in a program we have a list of numeric atoms and need the sum. We cannot apply + directly on the list, because + can take only atomic parameters, not a list of numeric atoms.
- Solution 1: Use recursion to go through the list:

```
(DEFINE (adder a_list)
        (COND
        ((NULL? a_list) 0)
        (ELSE (+ (CAR a_list) (adder (CDR a_list))))
))
```

an example call to adder

```
(adder '(3 4 5))
(+ 3 (adder (4 5)))
(+ 3 (+ 4 (adder (5))))
(+ 3 (+ 4 (+ 5 (adder ()))))
(+ 3 (+ 4 (+ 5 0)))
(+ 3 (+ 4 5))
(+ 3 9)
(12)
```

Adding a List of Numbers

```
(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 evaluates the resulting list
  - Use `CONS` to insert the atom `+` into the list of numbers.
  - Be sure that `+` is quoted to prevent evaluation
  - Submit the new list to `EVAL` for evaluation

an example call to adder version 2

```
(adder '(3 4 5))
(EVAL (+ 3 4 5)
(12)
```

Applications of Functional Languages

- LISP is used for artificial intelligence
  - Knowledge representation
  - Machine learning
  - Natural language processing
  - Modeling of speech and vision
- Scheme is used to teach introductory programming at some universities
- Support for functional programming is increasingly creeping into imperative languages

Comparing Functional and Imperative Languages

- Imperative Languages:
  - Efficient execution
  - Complex semantics
  - Complex syntax
  - Concurrency is programmer designed
- Functional Languages:
  - Simple semantics
  - Simple syntax
  - Inefficient execution
  - Programs can automatically be made concurrent

Examples

- Write a Scheme function intersection that takes as arguments two simple lists lst1 and lst2, and returns the intersection of the two lists. For example:

```
(intersection '(7 2 3 5) '(5 2)) => (2 5)
(intersection '(1 2 3 5) '(6 4)) => ()
(intersection '(a c d e g) '(h g a f d)) => (a d g)
```

```
(define (intersect lst1 lst2)
  (cond
  ((null? lst1) '())
  ((null? lst2) '())
   ((member (car lst1) lst2) (cons (car lst1)
(intersect (cdr lst1) lst2)))
  (else (intersect (cdr lst1) lst2))))
```

```
(define (insert x lst)
  (if (null? lst)
      (list x)
      (let ((y (car lst))
            (ys (cdr lst)))
      (if (<= x y)
            (cons x lst)
            (cons y (insert x ys))))))

(define (insertion-sort lst)
  (if (null? lst)
      '()
      (insert (car lst)
            (insertion-sort (cdr lst)))))

(insertion-sort '(6 8 5 9 3 2 1 4 7))
```