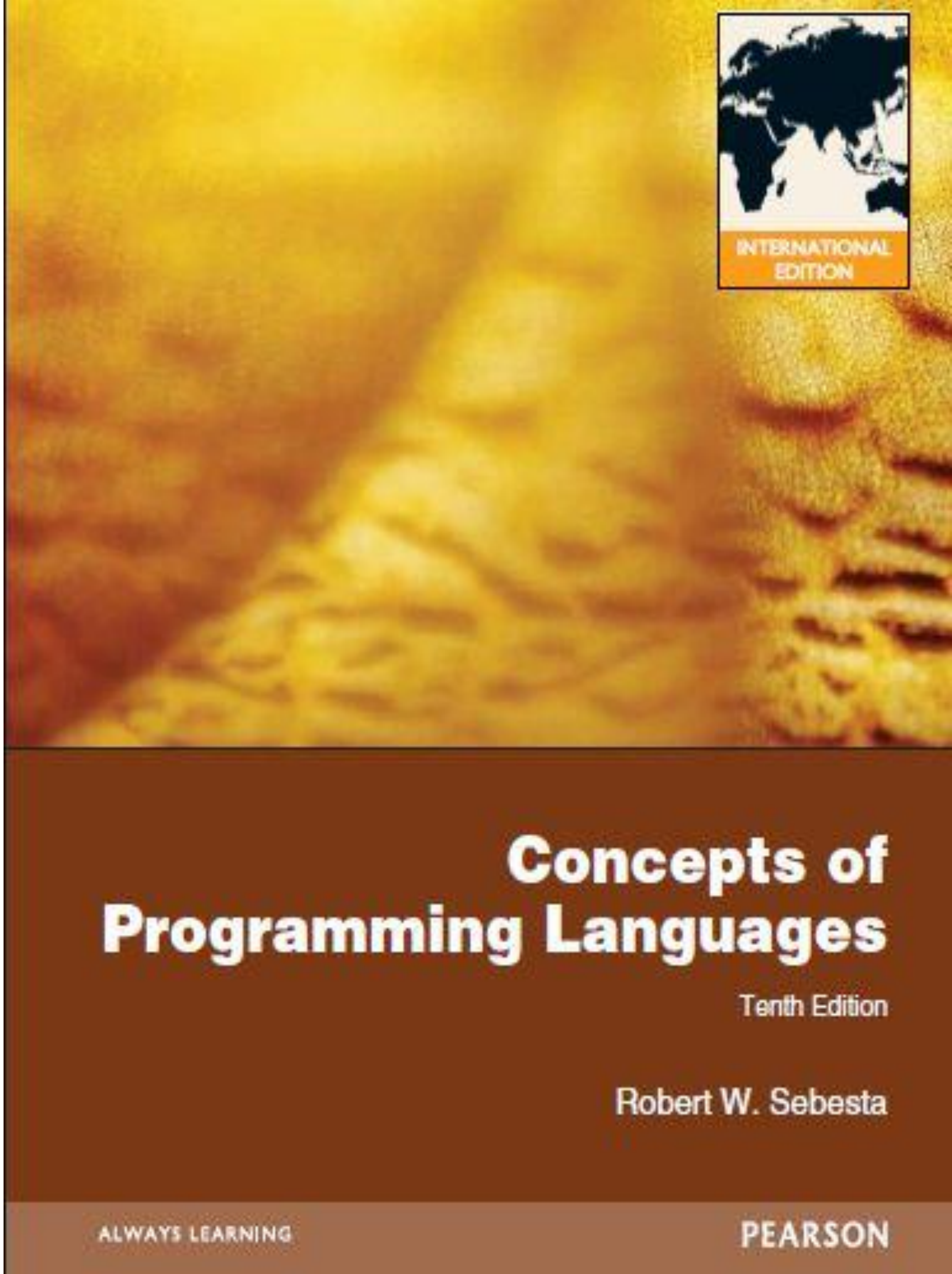
Programming Language

Instructor:
Min-Chun Hu
anita_hu@mail.ncku.edu.tw



Lecture 4
LISP

Functional Programming: LISP

- LISt Processing language
  - Designed at MIT by McCarthy in 1960s.
- AI research needed a language to
  - Process data in lists (rather than arrays)
  - Symbolic computation (rather than numeric)
- Only two data types: atoms and lists

LISP variants

- LISP is a functional language. Variants are
  - Scheme
  - Emacs LISP
  - AutoLISP
  - Common Lisp
  - Racket
- Common Lisp and Scheme are the most popular
  - Scheme is for educational purpose
  - Common Lisp is the most popular functional languages

Objects : Atoms

- Atoms
  - Numbers: 4 3.14 ½ #x16 #o22
  - Constants: pi t nil lambda-list-keywords
  - Characters: #\a #\Q #\space #\tab
  - Strings: "foo" "Hello Hi" "@%!?#"
    - a sequence of characters bounded by double quotes
  - Booleans: T for true, NIL for false
  - Symbols: Dave num123 miles->km !_^_! 2nd-place *foo*
    - Two special symbols: T for true and NIL for false

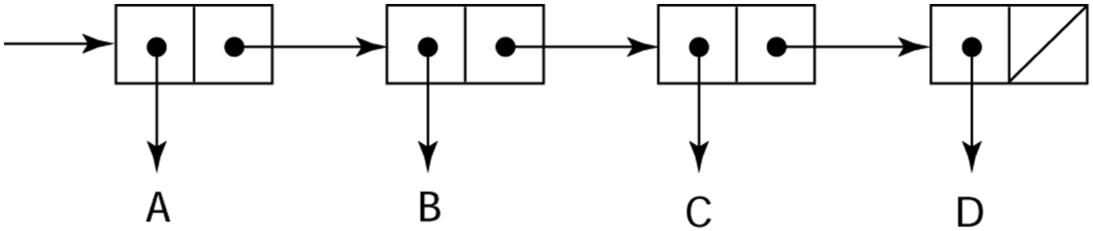
Objects : Lists

- **Lists**: a list of atoms and/or lists, bounded by "(" and ")“,

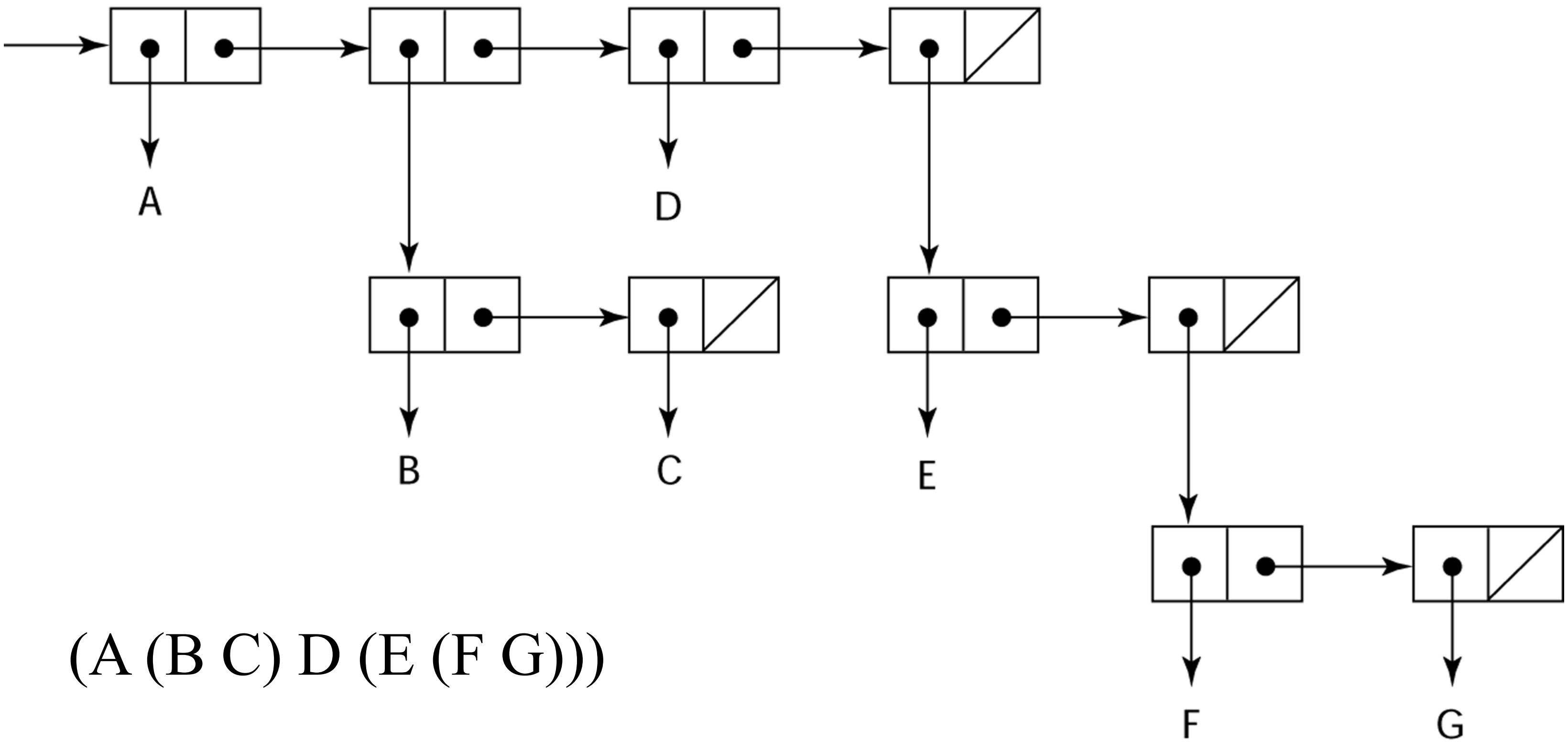
```
e.g.:
(a b c), (a (b c))
()
(a)  is equal to ( a () )
(a b c d)
((a b) c (d e))
(((((a)))))
```

- Top elements of a list
  - example: top elements of list (a b c) are a, b, and c
  - top elements of list (a (b c)) are a and (b c)
  - nil: empty list, same as ().

Representation of Two LISP Lists



(A B C D)



(A (B C) D (E (F G)))

Example of comments

- Examples of comments ;

```
;;; A comment formatted as a block of text
;;; outside of any function definition

(defun fib (n)
  ;; A comment on a line by itself
  (if (< n 3)
      1                   ; A comment on the same line as some code
      (+ (fib (- n 1))
         (fib (- n 2)))))

(setq *global-variable* 10)
(let (local-variable)
  (setq local-variable 15))
```

Substitution Model

- The basic rule:
  - To evaluate a expression:
    1. **Evaluate** its **operands**
    2. **Evaluate** the **operator**
    3. **Apply** the operator to the evaluated operands

```
(operator operand1 operand2 …)
```

```
(+ 3  (* 4  5) )
```

Example expression evaluate

- Example: (+ 3 (* 4 5))
  - evaluate 3
  - evaluate (* 4 5)
    - evaluate 4
    - evaluate 5
    - evaluate *
  - apply * to 4, 5 → 20
  - evaluate +
  - apply + to 3, 20 →23

Syntax (C vs. LISP)

C	LISP
`1 + 2 + 3`	`(+ 1 2 3)`
`3 + 4 * 5`	`(+ 3 (* 4 5))`
`factorial (9)`	`(factorial 9)`
`(a == b) && (c != 0)`	`(and (= a b) (not (= c 0)))`
`(low < x) && (x < high)`	`(< low x high)`
`f (g(2,-1), 7)`	`(f (g 2 -1) 7)`

Example

C

```
if(a == 0)
        return f(x,y);
else
        return g(x,y);
```

Common LISP

```
(if (= a 0)
    (f x y)
    (g x y))
```

Quote

- Use quote or ’ to avoid procedure application

```
>(+ 1 2)     => 3
>(1 2 3)     => error
>'(1 2 3)    => (1 2 3)
>(quote (1 2 3)) => (1 2 3)
```

Primitive Numeric Function

- Primitive Numeric Function:
  - function only deal with numeric atoms
  - + − * / sqrt

```
>42             => 42
>(* 3 7)        => 21
>(+ 5 7 8)       => 20
>(- 24 (* 4 3))   =>12
>(sqrt 4)          => 2
```

Examples

```
> (+)
0
> (+ 2)
2
> (+ 2 3)
5
> (+ 2 3 4)
9
> (+ 2 3 4 5)
14
```

Numerical Calculation Function

- There are some function
  - +, -, *, /
  - (sqrt 16) => 4
  - (expt 10 2) => 100
  - (max 1 2 3 4 5) => 5
  - (min 1 2 3 4 5) => 1
  - (mod (abs -27) 5) => 2

Example

- setq is a special form of function (with two arguments)

```
>(set (quote *foo*) 42)
42
```

```
>(setq *foo* 42)
42
```

```
>(setf (symbol-value '*foo*) 42)
42
```

```
>(setq x 3.0)
3.0
```

```
>x
3.0
```

```
>(setq y x)
3.0
; the value of y is assigned as  the value of x
```

```
>y
3.0
```

```
>(+ x y)
6.0
```

Let evaluation

- Let function:
  (let ((var1 val1) (var2 val2) ...) body)

```
>(let ((x 5) (y 8)) (* x y ))
40
```

```
>x
Error: Unbound variable
```

```
>(setq w 77)
77
>(let ((w 8) (x w)) (+ w x))
85
```

```
>w
77
```

```
>x
Error: Unbound variable
```

Essential Multiple Values

```
(values :this :that)
> :this ;
> :that
```

```
(values 4 (values))
> 4 ;
> nil
```

```
(multiple-value-bind (a b c) (values 2 3 5) (+ a b c))
>10
(multiple-value-bind (a b c) (values 2 3 6 5) (+ a b c))
> 11
```

```
(multiple-value-bind (x) (floor 5 3) (list x))
> (1)
(multiple-value-bind (x y) (floor 5 3) (list x y))
> (1 2)
(multiple-value-bind (x y z) (floor 5 3) (list x y z))
> (1 2 nil)
```

Lambda expression

- Create a function by a lambda expression:
- (lambda (id1 id2 ...) exp1 exp2 ...)
  - id1 id2 ... – formal parameters
  - exp1 exp2 ... – body of the function
  - (lambda (x) (* x x))

No function names
=> nameless function

- Call a function by applying the evaluated lambda expression on its actual parameters

((lambda (x) (* x x)) 3) => 9

the function the actual parameter

Function Definition

- Lambda expression

(lambda (x) (*   x x))

Formal parameter : x
Function body: * x x

- Function application

(lambda (x) (* x x))
> error

>((lambda (x) (* x x))   7)
49

Lambda function

- How can you reuse the function?
  - You can't!   (since it is nameless)
- What if you REALLY want to reuse it?
  - "defun"

Function Definition

- How to define a function?

  (defun <function-name> (<formal-parameters>) <expression>)

- Bind a name to a function:

  ```
  >(defun square (x) (* x x) )
  ```

- Now call the function

  ```
  > (square 5)
  > 25
  ```

Conditions

- Boolean value true is T and false is NIL
- Predicates are expressions that evaluate to true or false
- Numberp
  - test whether argument is a number
    - (numberp 9)

      T

    - (numberp ’a)

      NIL

Some useful predicates

- `(= x y)` returns true if `x` and `y` are identical.
  - Or you can write `(equal x y)` or `(eq x y)`
  - Represents the same number or symbol or the same list.
- For example:
  - `(equal 6 6)      => true`

```
>(numberp x)          ; is x a number?
>(stringp x)          ; is x a string?
>(listp x)            ; is x a list?
>(symbolp x)          ; is x a symbol?
>(atom x)             ; is x a atom?
>(null x)             ; is x nil?
```

```
(eq 'a 'b) is false.
(eq 'a 'a) is true.
(eq 3 3) might be true or false, depending on the implementation.
(eq 3 3.0) is false.
(eq 3.0 3.0) might be true or false, depending on the implementation.
(eq #c(3 -4) #c(3 -4))
  might be true or false, depending on the implementation.
(eq #c(3 -4.0) #c(3 -4)) is false.
(eq (cons 'a 'b) (cons 'a 'c)) is false.
(eq (cons 'a 'b) (cons 'a 'b)) is false.
(eq '(a . b) '(a . b)) might be true or false.
(progn (setq x (cons 'a 'b)) (eq x x)) is true.
(progn (setq x '(a . b)) (eq x x)) is true.
(eq #\A #\A) might be true or false, depending on the implementation.
(eq "Foo" "Foo") might be true or false.
(eq "Foo" (copy-seq "Foo")) is false.
(eq "FOO" "foo") is false.
```

```
(eql 'a 'b) is false.
(eql 'a 'a) is true.
(eql 3 3) is true.
(eql 3 3.0) is false.
(eql 3.0 3.0) is true.
(eql #c(3 -4) #c(3 -4)) is true.
(eql #c(3 -4.0) #c(3 -4)) is false.
(eql (cons 'a 'b) (cons 'a 'c)) is false.
(eql (cons 'a 'b) (cons 'a 'b)) is false.
(eql '(a . b) '(a . b)) might be true or false.
(progn (setq x (cons 'a 'b)) (eql x x)) is true.
(progn (setq x '(a . b)) (eql x x)) is true.
(eql #\A #\A) is true.
(eql "Foo" "Foo") might be true or false.
(eql "Foo" (copy-seq "Foo")) is false.
(eql "FOO" "foo") is false.
```

```
(equal 'a 'b) is false.
(equal 'a 'a) is true.
(equal 3 3) is true.
(equal 3 3.0) is false.
(equal 3.0 3.0) is true.
(equal #c(3 -4) #c(3 -4)) is true.
(equal #c(3 -4.0) #c(3 -4)) is false.
(equal (cons 'a 'b) (cons 'a 'c)) is false.
(equal (cons 'a 'b) (cons 'a 'b)) is true.
(equal '(a . b) '(a . b)) is true.
(progn (setq x (cons 'a 'b)) (equal x x)) is true.
(progn (setq x '(a . b)) (equal x x)) is true.
(equal #\A #\A) is true.
(equal "Foo" "Foo") is true.
(equal "Foo" (copy-seq "Foo")) is true.
(equal "FOO" "foo") is false.
```

```
(equalp 'a 'b) is false.
(equalp 'a 'a) is true.
(equalp 3 3) is true.
(equalp 3 3.0) is true.
(equalp 3.0 3.0) is true.
(equalp #c(3 -4) #c(3 -4)) is true.
(equalp #c(3 -4.0) #c(3 -4)) is true.
(equalp (cons 'a 'b) (cons 'a 'c)) is false.
(equalp (cons 'a 'b) (cons 'a 'b)) is true.
(equalp '(a . b) '(a . b)) is true.
(progn (setq x (cons 'a 'b)) (equalp x x)) is true.
(progn (setq x '(a . b)) (equalp x x)) is true.
(equalp #\A #\A) is true.
(equalp "Foo" "Foo") is true.
(equalp "Foo" (copy-seq "Foo")) is true.
(equalp "FOO" "foo") is true.
```

Conditional Expressions: IF

- Conditional expressions come in two forms:

```
(1)  (if P E1 E2)    ; if P then E1  else E2
(2)  (cond (P1 E1)        ; if P1 then E1
          ......          ; ...
          (Pk Ek)         ; else if Pk then Ek
          (t Ek+1))       ; else Ek+1
```

Conditional Expression: IF

- (if a b c) => b if a is true, else c
  - (if (< 5 6) 1 2) → 1
  - (if (< 4 3) 1 2) → 2
- Anything other than NIL is treated as true:
  - (if 3 4 5) → 4
- if is a special form – evaluates its arguments only when needed:
  - (if (= 3 4) 1 (2)) → error
  - (if (= 3 4) 1 '(2)) → (2)
  - (if (= 3 4) 1 (if 5 2 3)) → 2

Conditional Expression: COND

- Compare x and y

```
(defun compare (x y)
  (cond ((equal x y) 'numbers-are-the-same)
        ((< x y) 'first-is-smaller)
        ((> x y) 'first-is-bigger)))

➢ (compare 3 4)
FIRST-IS-SMALLER
```

Let & defun & cond

- Example: "guess number"

```
(defun secret-number (number)
        (let ((secret 37))
                (cond ((= number secret) 'True)
                      ((< number secret) 'too-low)
                      ((> number secret) 'too-high))))
```

Example

- Design a function to find answer of $aX^2 + bX + c = 0$.

```
[1]> (defun quadratic-roots (a b c)
     "Returns the roots of a quadratic equation aX^2 + bX + c = 0"
     (let ((discriminant (- (* b b) (* 4 a c))))
       (values (/ (+ (- b) (sqrt discriminant)) (* 2 a))
               (/ (- (- b) (sqrt discriminant)) (* 2 a)))))
QUADRATIC-ROOTS
[2]> (quadratic-roots 1 2 1)
-1 ;
-1
```

Example

- Rewrite a function to find aX^2 + bX + c =0 answer.

```
(defun quadratic-roots-2 (a b c)
  (let ((num (- (* b b) (* 4 a c))))   ; zero if one root
   (cond ((zerop num) (/ (+ (- b) (sqrt num)) (* 2 a)))
         (t  (values (/ (+ (- b) (sqrt num)) (* 2 a))
              (/ (- (- b) (sqrt num)) (* 2 a)))))))
```

```
[6]> (quadratic-roots-2 1 -14 49)
7
[7]> (quadratic-roots-2 1 4 -5)
1 ;
-5
```

List manipulation in Lisp

- Three primitives and one constant
  - get head of list: car
  - get rest of list: cdr
  - add an element to a list: cons
  - null list: nil or ()

List Operations

- cons – returns a list built from head and tail
  - (cons 'a '(b c d)) → (a b c d)
  - (cons 'a '()) → (a)
  - (cons '(a b) '(c d)) → ((a b) c d)
  - (cons 'a (cons 'b '())) → (a b)

List Operations

- car – returns first member of a list (head)
  - (car '(a b c d)) → a
  - (car '(a)) → a
  - (car '((a b) c d)) → (a b)
  - (car '(this (is no) more difficult)) → this
- cdr – returns the list without its first member (tail)
  - (cdr '(a b c d)) => (b c d)
  - (cdr '(a b)) => (b)
  - (cdr '(a)) => NIL
  - (cdr '(a (b c))) =>((b c))

List Operations

```
(caar x)       (car (car x))
(cadr x)       (car (cdr x))
(cdar x)       (cdr (car x))
(cddr x)       (cdr (cdr x))
(caaar x)      (car (car (car x)))
(caadr x)      (car (car (cdr x)))
(cadar x)      (car (cdr (car x)))
(caddr x)      (car (cdr (cdr x)))
(cdaar x)      (cdr (car (car x)))
(cdadr x)      (cdr (car (cdr x)))
(cddar x)      (cdr (cdr (car x)))
(cdddr x)      (cdr (cdr (cdr x)))
(caaaar x)     (car (car (car (car x))))
(caaadr x)     (car (car (car (cdr x))))
(caadar x)     (car (car (cdr (car x))))
(caaddr x)     (car (car (cdr (cdr x))))
(cadaar x)     (car (cdr (car (car x))))
(cadadr x)     (car (cdr (car (cdr x))))
(caddar x)     (car (cdr (cdr (car x))))
(cadddr x)     (car (cdr (cdr (cdr x))))
(cdaaar x)     (cdr (car (car (car x))))
(cdaadr x)     (cdr (car (car (cdr x))))
(cdadar x)     (cdr (car (cdr (car x))))
(cdaddr x)     (cdr (car (cdr (cdr x))))
(cddaar x)     (cdr (cdr (car (car x))))
(cddadr x)     (cdr (cdr (car (cdr x))))
(cdddar x)     (cdr (cdr (cdr (car x))))
(cddddr x)     (cdr (cdr (cdr (cdr x))))
```

List Operations

```
(setq llst '(nil))
>(NIL)
 (push 1 (car llst))
>  (1)
 llst
>  ((1))
 (push 1 (car llst))
>  (1 1)
 llst
>  ((1 1))
```

```
 (setq x '(a (b c) d))
>  (A (B C) D)
 (push 5 (cadr x))
>  (5 B C)
 x
>  (A (5 B C) D)
```

List Operations

- null – returns T if the list is empty,
  returns NIL if not empty
  - ➢ (null '())
  T

- list – returns a list built from its arguments
  - (list 'a 'b 'c) → (a b c)
  - (list '(a b c)) → ((a b c))
  - (list '(a b) '(c d)) → ((a b) (c d))

List Operations

- length – returns the length of a list
  - (length '(1 3 5 7)) → 4
  - (length '((a b) c)) → 2
- reverse – returns the list reversed
  - (reverse '(1 3 5 7)) → (7 5 3 1)
  - (reverse '((a b) c)) → (c (a b))

Recursion

- Fibonacci:

$$
fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}
$$

```
(defun fib (n)
"Simple recursive Fibonacci number function"
(if (< n 2)
   n
  (+ (fib (- n 1)) (fib (- n 2)))))
```

Recursion

- How do you THINK recursively?
- Example: define factorial

$$\text{factorial}(n) = \underbrace{1 * 2 * 3 * \ldots(n-1)}_{\text{factorial }(n-1)} * n$$

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n=1 \quad \text{(the base case)} \\ n * \text{factorial}(n-1) & \text{otherwise} \quad \text{(inductive step)} \end{cases}$$

Version 1

- Example: define a factorial function

```
(defun  fact  (n)
    ( if ( = n 0 )
           1
           (* n    (fact(- n 1 )))))
```

Steps of Version 1

Stack (space)

Time

```
(fact 5)
(* 5 (fact 4))
(* 5 (* 4 (fact 3)))
(* 5 (* 4 (* 3 (fact 2))))
(* 5 (* 4 (* 3 (* 2 (fact 1)))))
(* 5 (* 4 (* 3 (* 2 (* 1 (fact 0))))))
.
.
.
.
120
```

```
(defun  fact  (n)
   ( if ( = n 0 )
            1
            (* n    (fact(- n 1 )))))
```

Version 2: Tail Recursion

- a tail-recursive function is one in which the recursive call occurs last

```
(defun tailfact (n result)
  (if (= n 0)
     result
    (tailfact (- n 1 ) (* n result))))

(defun fact (n)
  (tailfact n 1))
```

Steps of Version 2

Stack

```
(fact 5)
(tailfact 5 1)
(tailfact 4 5)
(tailfact 3 20)
(tailfact 2 60)
(tailfact 1 120)
(tailfact 0 120)
120
```

Time

```
(defun tailfact (n result)
  (if (= n 0)
     result
     (tailfact (- n 1 ) (* n result))))
```

Trace in Ver.1 and Ver.2

- Version 1:

```
[4]> (trace fact)
;; Tracing function FACT.
(FACT)
```

```
[6]> (fact 5)
1. Trace: (FACT '5)
2. Trace: (FACT '4)
3. Trace: (FACT '3)
4. Trace: (FACT '2)
5. Trace: (FACT '1)
6. Trace: (FACT '0)
6. Trace: FACT ==> 1
5. Trace: FACT ==> 1
4. Trace: FACT ==> 2
3. Trace: FACT ==> 6
2. Trace: FACT ==> 24
1. Trace: FACT ==> 120
120
```

Version 2:

```
[5]> (trace fact2)
;; Tracing function FACT2.
(FACT2)
```

```
[7]> (fact2 5)
1. Trace: (FACT2 '5)
1. Trace: FACT2 ==> 120
120
```

How does it work?

- Tail recursion requires two elements(more than two)
  - The tail recursive module must terminate with a recursive call that leaves no work on the stack to finish up.
  - Any storage must be done in the parameter list as opposed to the stack
  - The interpreter or compiler must be designed to recognize tail recursion and handle it appropriately

Let's have a look at the three programs

$$\sum_{k=1}^{100} k$$ = (sum-integers 1 100)

```
(define (sum-integers k  n)
   (if  (>  k  n)
        0
        (+  k
            (sum-integers (+ 1 k)  n))))
```

$$\sum_{k=1}^{100} k^2$$ = (sum-squares 1 100)

```
(define (sum-squares  k  n)
   (if  (>  k  n)
        0
        (+  (square K)
            (sum-squares  (+ 1 k)  n))))
```

$$\sum_{k=1,odd}^{101} k^{-2}$$ = (pi-sum 1 101)

```
(define (pi-sum  k  n)
   (if  (>  k  n)
        0
        (+  (/  1  (square  k))
            (pi-sum   (+  k 2)  n))))
```

Loop

```
> (setq a 4)
4
> (loop
    (setq a (+ a 1))
    (when (> a 7) (return a)))
8
> (loop
    (setq a (- a 1))
    (when (< a 3) (return)))
NIL
```

```
> (dolist (x '(a b c)) (print x))
A
B
C
NIL
```

Question

- How to implement a list reverse function?
  - (reverse '(1 3 5 7)) → (7 5 3 1)
  - (reverse '((a b) c)) → (c (a b))

Loop

```
> (do ((x 1 (+ x 1))
       (y 1 (* y 2)))
      ((> x 5) y)
    (print y)
    (print 'programming)
  )
1
PROGRAMMING
2
PROGRAMMING
4
PROGRAMMING
8
PROGRAMMING
16
PROGRAMMING
32
```

Loop

```
(loop for x in '(a b c d e)
      do (print x) )

A
B
C
D
E
NIL
```

```
(loop for x in '(a b c d e)
      for y in '(1 2 3 4 5)
      collect (list x y) )

((A 1) (B 2) (C 3) (D 4) (E 5))
```

```
(loop for x from 1 to 5
      for y = (* x 2)
      collect y)

(2 4 6 8 10)
```

Loop

```
(loop for x in '(a b c d e)
      for y from 1

      when (> y 1)
      do (format t ", ")

      do (format t "~A" x)
)

A, B, C, D, E
NIL
```

```
(loop for x from 1
      for y = (* x 10)
      while (< y 50)

      do (print (* x 5))

      collect y)

5
10
15
20
(10 20 30 40)
```