Pattern Matching in Lisp

Lists can be used to represent sentences, relations, tree structures, etc.

```
(this list represents a sentence)
```

Analysis of such data can often be based on patterns.

Patterns capture regularities, repetitions, ordering constraints.

```
(<determiner> <noun> <verb> <det.> <noun>)
```

Matching Can Be Strict or Loose

Strict equality:

```
(setq pattern '(quit this program))
(setq subject '(quit this program))
(equal pattern subject) ; => T
```

Even equality is relative:

```
(eq 'a 'a)        ; => T   strictest
(eq 5 5)          ; => NIL or T
(eql 5 5)         ; => T
(eql 5 5.0)       ; => NIL
(= 5 5.0)         ; => T
(equal 5 5.0)     ; => NIL
(equalp 5 5.0)    ; => T   most tolerant
```

Structural Similarity in Lisp

Structural similarity:

```
(defun match2 (p s)
   (cond ((atom p)(atom s))
         ((atom s) nil)
         ((/= (length p)(length s)) nil)
         ((eval (cons 'and
            (mapcar #'match2 p s) )) t)
         (t nil) ) )

(match2 '(a (b c) d) '(w (x y) z))    ; => T
(match2 '(a (b c) d) '(w (x (y)) z)) ; => NIL
```

List Equality with WildCards

A wildcard matches any one element:

```
(defun match3 (p  s)
   (cond
      ((null p)(null s))
      ((or (atom p)(atom s)) nil)
      ((equalp (car p) (car s))
       (match3 (cdr p)(cdr s)) )
      ((eq (car p) '?)(match3 (cdr p)(cdr s)))
      (t nil) ) )

(match2 '(a ? c) '(a b c)   ; => T
(match2 '(a ? c) '(a b d c) ; => NIL
```

Association Lists

An association list is a list of dotted pairs in which each left hand side is a symbol:

```
(setq alist '((a . 1)(b . 2)(c . 3)) )

(setq alist (acons 'x 1 nil))
((X . 1))
(setq alist (acons 'y 2 alist))
((Y . 2)(X . 1))
```

Association pairs are retrieved using ASSOC:

```
(assoc 'x alist) ; => (X . 1)
(cdr (assoc 'y alist)) ; => 2
```

Matching with Bindings in an Assoc. List

```
(defun match4 (p  s)
  (cond
    ((and (null p)(null s)) '((:yes . :yes)))
    ((or (atom p)(atom s)) nil)
    ((equalp (car p) (car s))
     (match4 (cdr p)(cdr s)) )
    ((and (= (length (car p)) 2)
          (eql (caar p) '?)
      (let ((rest-match
                (match4 (cdr p)(cdr s))))
        (if rest-match
          (acons (cadar p)
                 (car s) rest-match) ) ) ) )
    (t nil) ) )
```

Using Patterns with Wildcards

```
> (match4 '(a (? x)) '(a y))
((X . Y) (:YES . :YES))


> (match4 '(a (? x) (? y)) '(a (1 2)(3 4)))
((X 1 2) (Y 3 4) (:YES . :YES))
```

Patterns with Wild Sequences and Restricted Wild Cards

```
> (match '(a (* x)) '(a b c d))
((X B C D)) (:YES . :YES))

> (match '(he is (numberp x)(* y))
           '(he is twenty-five years old))
NIL
> (match '(he is (numberp x)(* y))
           '(he is 25 years old))
((X . 25)(Y YEARS OLD)(:YES . :YES))
```

Using Patterns for Dialog Programs

```
(defun dialog ( )
 (print 'type something in parentheses)
 (loop
  (setq user-input (read))
  (cond ((condition1)   (action1))
        ((condition2)   (action2))
          . . .
        ((conditionk)   (actionk))   )

  ) ) )
```

Dialog Patterns

```
((match '(you feel (* x)) s)
 (format t "I SOMETIMES FEEL THE SAME WAY.~%"))
((match '(because (* x)) s)
 (format t "IS THAT REALLY THE REASON.~%"))
((member 'no s)
 (format t "DONT BE SO NEGATIVE.~%"))
```

Dialog Patterns at Work

```
 WELCOME TO MY SOFA!
PLEASE ENCLOSE YOUR INPUT IN PARENTHESES.
(hello there)
TELL ME MORE.
(i am here to talk)
PLEASE TELL ME WHY YOU ARE HERE TO TALK.
(i came to get educated)
YOU CAME TO GET EDUCATED.
(yes)
HOW CAN YOU BE SO SURE?
(because i am)
IS THAT REALLY THE REASON.
(no)
DONT BE SO NEGATIVE.
```

Working with Strings in Lisp

```
(setq s "This is a real live string.")
(elt s 8) ; => #\a
; note that the first position is 0.

; Testing character objects:
(alphanumericp (elt s 7)) ; => NIL
; MUST BE EITHER ALPHA-CHAR-P or
; DIGIT-CHAR-P.

(digit-char-p #\9) ; => 9, i.e., True.

(string-upcase s) ; =>
”THIS IS A REAL LIVE STRING.”
```

Reading Text from the User

```
 ; Read in a line of text from the user.

(defun text-from-user ()
  (format t "Enter some text: ")
  (let ((user-text (read-line t)))
    (format t "You entered: ~A" user-text)
    'bye
  ) )

; T means the input comes from the keyboard.
```

Reading Text from a File

```
; Read in successive lines of test from a file.

(defun text-from-file ()
  (with-open-file (file "input.txt" :direction :input)
    (let (the-line)
      (loop
        (setq the-line (read-line file nil 'done))
        (if (eq the-line 'done) (return))
        (format t "The input line was: ~A~%" the-line)
      )
      'bye
    )
) )
```