Conditionals in LISP

✔ Conditionals use predicates to direct the course of evaluation

✗ (if test-expression then-expression else-expression)
  • test-expression is evaluated
  • if the value is not nil, then-expression is evaluated and its value is the value of the if
  • otherwise, else-expression is evaluated and its value is the value of the if

;;;
;;; count-elements takes a list as argument and
;;; returns the number of elements in the list
;;; (defun count-elements (L)

          (if L (+ 1 (count-elements (rest L)) 0))

✗ (when test-expression then-expression)
  • equivalent to
    (if test-expression then-expression NIL)

✗ (unless test-expression else-expression)
  • equivalent to
    (if test-expression NIL else-expression)

✗ Think about how to write if in terms of and, or, not
Conditionals in LISP

✔ The `cond` construct

```
(cond
  (test₁ expr₁₁ expr₁₂ ...)
  (test₂ expr₂₁ expr₂₂ ...)
  ...
  (testₙ exprₙ₁ exprₙ₂ ...))
```

✗ `cond` is followed by a list of clauses
✗ Each clause is a list
✗ The first element of each clause is a test
✗ The tests are evaluated in order; the first test that evaluates non-NIL triggers its clause
✗ The remaining elements of the triggered clause are evaluated, and the value of the last one is returned as the value of the `cond`
✗ Note: no other expressions in the `cond` are evaluated

```
(defun how-hot (temp)
  (cond
    ((not (numberp temp)) (error))
    ((< temp 0) 'really-cold)
    ((< temp 40) 'cold)
    ((< temp 60) 'cool)
    ((< temp 80) 'nice)
    ((< temp 100) 'hot)
    (t 'really-hot)))
```
Variable Binding in LISP

✓ Function parameters have values bound to them each time the function is called

```
(defun foo (x)
  (cond
    ((endp x) 'zero)
    (t (print (length x)) (foo (rest x)))))
```

USER: (foo '(a b c))

3
2
1
ZERO

✓ Parameters can also have values bound to them using the `let` construct...
Variable Binding in LISP

✔ The **let** construct

\[
(\text{let} \ (\text{param}_1 \ \text{initial-value}_1) \\
(\text{param}_2 \ \text{initial-value}_2) \\
\ldots \\
(\text{param}_n \ \text{initial-value}_n)) \\
expression_1 \\
expression_2 \\
\ldots \\
expression_m)
\]

✔ The **initial-value** expressions are evaluated “in parallel”

✔ The **param** symbols are bound to the corresponding values

✔ The **expressions** in the body of the **let** are evaluated in order, in the context of these *lexical variable bindings*

✔ The value of the last **expression** is the value of the **let**
Variable Binding in LISP

✔ The `let` construct

```lisp
(defun both-ends (list)
  (let ((first (first list))
        (last (first (reverse list))))
    (list first last)))
```

✔ Parameters in a `let` “shadow” parameters in surrounding contexts

```lisp
(defun what-is-it (X Y)
  (let ((x 7)
        (z y))
    (let ((x z)
           (y x))
      (list x y z))))
```

USER: `(what-is-it 3 4)

(4 7 4)
Variable Binding in LISP

✔ setf and setq

(setf symbol expression)

does not evaluate symbol; evaluates expression and binds the symbol to this value

✔ setf and setq permit changing the value of a variable
✔ setf and setq are assignment statement constructs
✔ setf and setq are not part of “pure” LISP
✔ Use setf and setq for convenience at “top level”, not within the body of function definitions

USER: (setq my-list '(now is (the time (to learn lisp))))
(NOW IS (THE TIME (TO LEARN LISP)))

USER: (test-function my-list)
...
(OK)

USER: (defun my-fun (arg) (setq result (first arg) ....

(NOT OK)
What we know about so far:

✔ What expression types LISP has
✔ How LISP evaluates expressions
✔ How to construct lists in LISP
✔ How to manipulate numbers in LISP
✔ Predicates
✔ How to logically combine predicates
✔ How to use conditionals to control evaluation
✔ How to define and use functions in LISP
✔ How to bind local variables with let
Tips on writing LISP code

✔ Progressive envelopment
  × try out pieces of a function by giving them to the interpreter
  × combine the pieces into a function definition
  × example: defining the both-ends function

USER: (setq L '(1 2 3 4 5))
(1 2 3 4 5)

USER: (first L)
1

USER: (last L)
(5)

USER: (first (last L))
5

USER: (list (first L) (first (last L)))
(1 5)

USER: (defun both-ends (list)
  (list (first list) (first (last list))))
BOTH-ENDS

USER: (both-ends L)
(1 5)
Tips on writing LISP code

✔ Comment translation
  ✗ write what needs to be computed in English comments
  ✗ translate the comments into LISP
  ✗ example: defining the both-ends function

(defun both-ends (L)
  ;; extract first element
  ;; extract last element
  ;; combine the first and last elements into a list
 )

(defun both-ends (L)
  (first L) ;; extract first element
  ;; extract last element
  ;; combine the first and last elements into a list
 )

(defun both-ends (L)
  (first L) ;; extract first element
  (first (last L)) ;; extract last element
  ;; combine the first and last elements into a list
 )

(defun both-ends (L)
  ;; combine the first and last elements into a list
  (list
    (first L) ;; extract first element
    (first (last L)) ;; extract last element
  )
 )
Tips on writing LISP code

✔ Comment translation can be combined with further function definition

(defun both-ends (L)
   ;; extract first element
   ;; extract last element
   ;; combine the first and last elements into a list
   )

(defun both-ends (L)
   (extract-first-element L) ;; extract first element
   (extract-last-element L) ;; extract last element
   ;; combine the first and last elements into a list
   )

(defun both-ends (L)
   ;; combine the first and last elements into a list
   (combine-elements-into-list
      (extract-first-element L) ;; extract first element
      (extract-last-element L) ;; extract last element
   )
)

(defun extract-first-element (L) (first L))

(defun extract-last-element (L) (first (last L)))

(defun combine-elements-into-list (L1 L2) (list L1 L2))
Some examples

✔ Define a function \texttt{rev} which works like the builtin function \texttt{reverse}

✔ Use comment translation

\begin{verbatim}
;;; REV
;;; arguments: L, a list
;;; returns: a list with elements in reverse order
;;; (defun rev (L)
;;;   ;;
;;;   ;;
;;;   ;;
;;;   )
\end{verbatim}

✔ An alternative definition of rev, using an “auxiliary function”

\begin{verbatim}
(defun rev (L) (rev-aux L nil))
\end{verbatim}

\begin{verbatim}
;;; REV-AUX
;;; arguments: L1, a list; L2, a list
;;; returns: the result of consing the elements of L1 onto L2 in reverse order
;;; (defun rev-aux (L1 L2)
;;;   (if (endp L1) L2
;;;       (rev-aux (rest L1) (cons (first L1) L2)))))
\end{verbatim}

✗ \texttt{rev-aux} is a “tail-recursive” function...
Some examples

✓ Define the function count-elements using an auxilliary function

(defun count-elements (L) (count-elements-aux L 0))

;;; COUNT-ELEMENTS-AUX
;;; arguments: L, a list; n, an integer
;;; returns: n + the number of elements in L
;;; (defun count-elements-aux (L n)
; (if (endp L) n
; (count-elements-aux (rest L) (+ n 1))))
Some examples

✔ Define a function \texttt{count-atoms} which returns the number of atoms at any level of nesting in a list

\texttt{USER: (count-atoms ' (sqrt (expt x 2) (expt y 2)))}

7

;;; COUNT-ATOMS
;;; arguments: L, a list
;;; returns: the number of atoms at any level in the list
(defun count-atoms (L)
	)