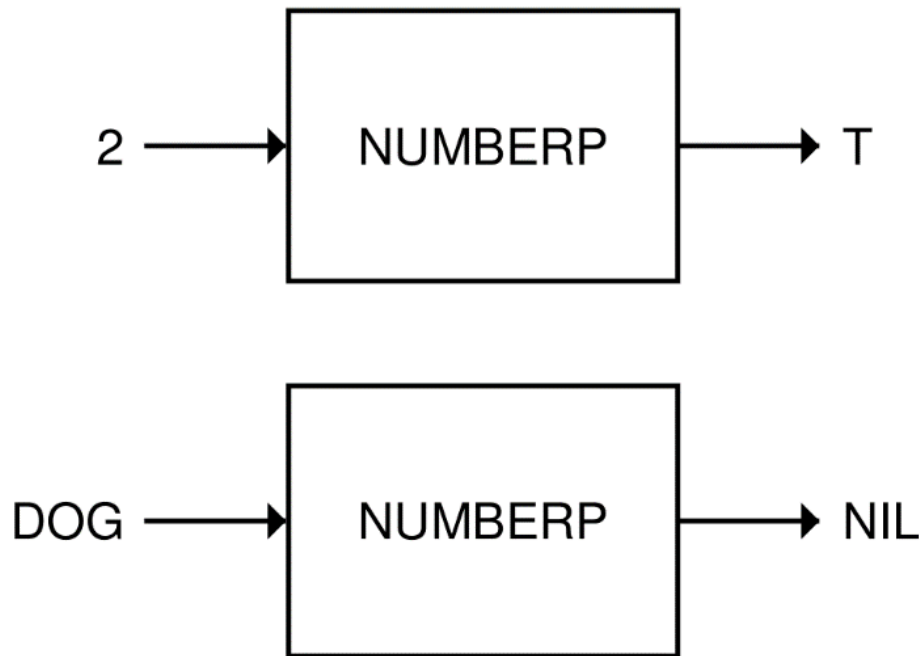


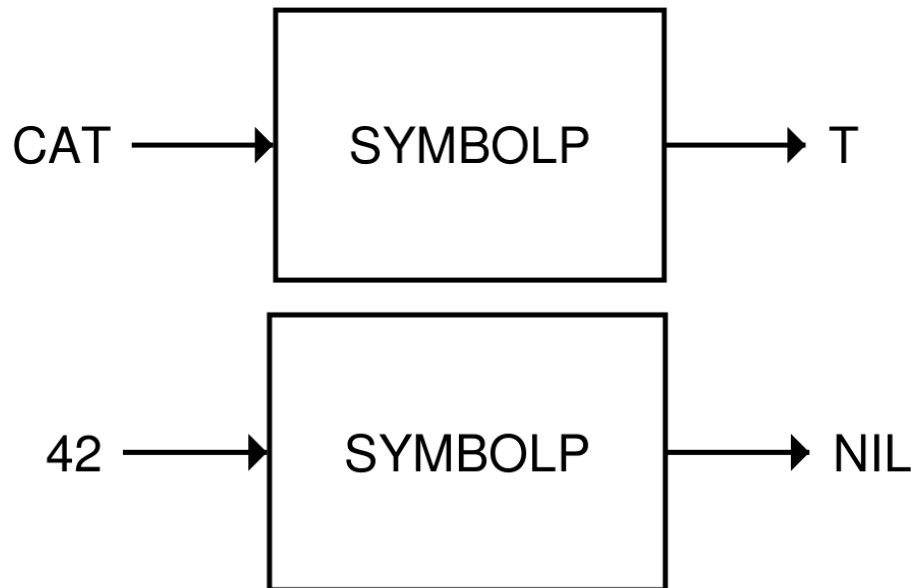
From p. 8 of Touretzky:

A predicate is a question-answering function. Predicates output the symbol *T* when they mean *yes* and the symbol *NIL* when they mean *no*. The first predicate we will study is the one that tests whether its input is a number or not. It is called **NUMBERP** (pronounced “number-pee,” as in “number predicate”), and it looks like this:



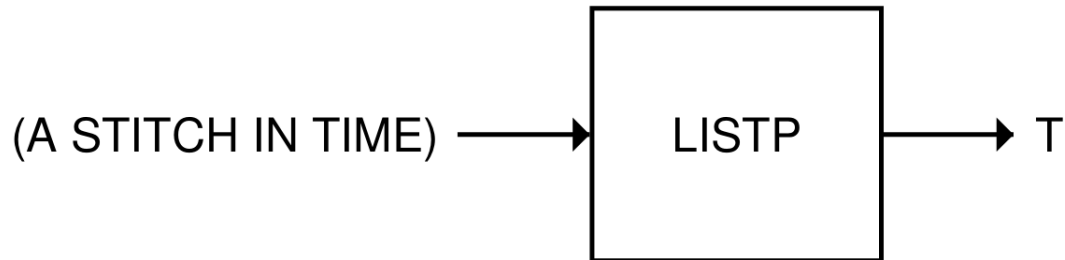
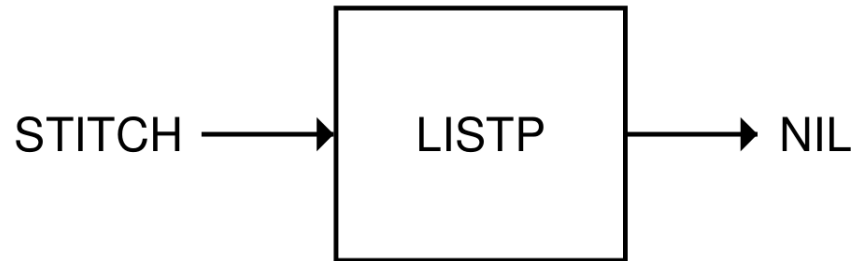
From pp. 8 – 9 of Touretzky:

Similarly, the SYMBOLP predicate tests whether its input is a symbol. SYMBOLP returns T when given an input that is a symbol; it returns NIL for inputs that are not symbols.



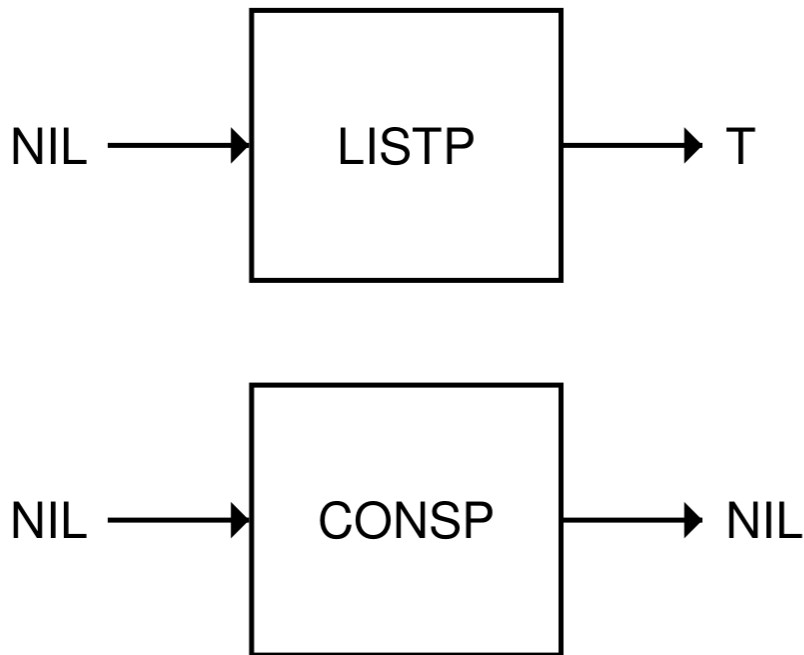
From p. 66 of Touretzky:

The LISTP predicate returns T if its input is a list. LISTP returns NIL for non-lists.



From pp. 66 – 7 of Touretzky:

The CONSP predicate returns T if its input is a cons cell. CONSP is almost the same as LISTP; the difference is in their treatment of NIL. NIL is a list, but it is not a cons cell.

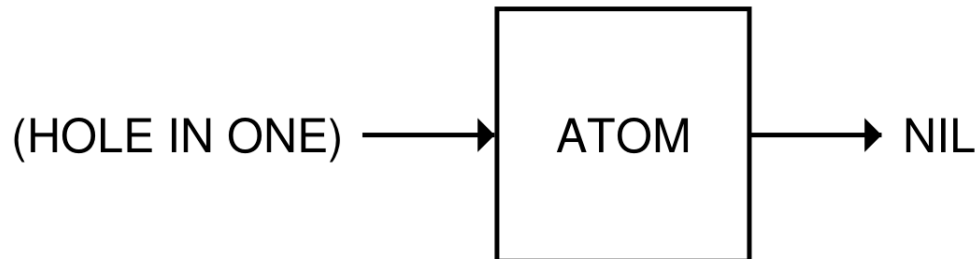
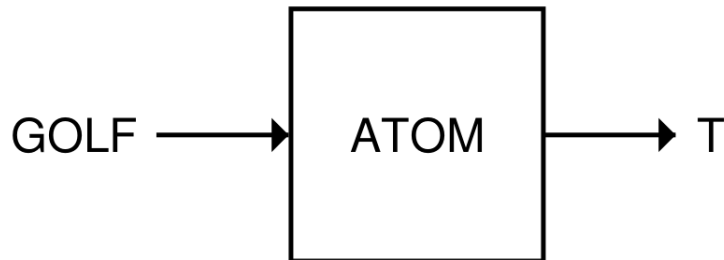
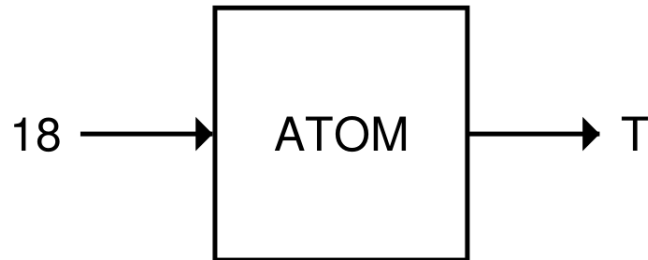


- $(\text{consp } x) = (\text{typep } x \text{ 'cons}) \Rightarrow \text{T}$ if $x \Rightarrow$ a nonempty list.
 $(\text{consp } x) = (\text{typep } x \text{ 'cons}) \Rightarrow \text{NIL}$ if $x \Rightarrow$ an atom.

From p. 67 of Touretzky:

The ATOM predicate returns T if its input is anything other than a cons cell. ATOM and CONSP are opposites; when one returns T, the other always returns NIL.

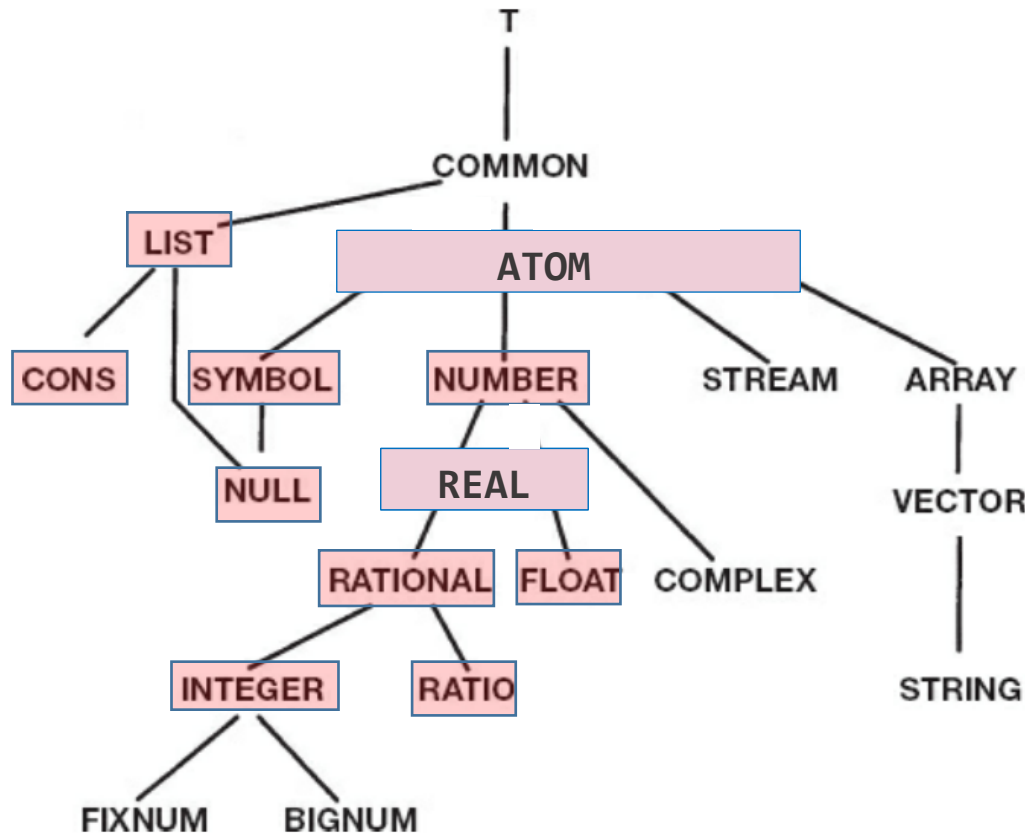
(atom x) = (not (consp x))



`(typep x '<type>)` \Rightarrow T if $x \Rightarrow$ a value of type `<type>`.

`(typep x '<type>)` \Rightarrow NIL if $x \Rightarrow$ a value whose type is not `<type>`.

`<type>` *can be any of the type names shown in the tree on the right* except for COMMON, which is now obsolete.



Reminder: The built-in function names

ATOM and **NULL**

do not end with **P**!

```
(atom x)
= (typep x 'atom)
(null x)
= (typep x 'null)
```

From p. 367 of Touretzky (with ATOM and REAL types added)
Figure 12-1 A portion of the Common Lisp type hierarchy.

$(> x_1 \dots x_n)$, $(\geq x_1 \dots x_n)$, $(< x_1 \dots x_n)$, and $(\leq x_1 \dots x_n)$

1. If any argument value is not a real number, then evaluation of $(> x_1 \dots x_n)$ produces an error!
2. If the argument values are all rational or all floating point, then:
 - $(> x_1 \dots x_n) \Rightarrow \text{True}$ if $(- x_i x_{i+1}) > 0$ for $1 \leq i < n$.
 - $(> x_1 \dots x_n) \Rightarrow \text{NIL}$ otherwise.
3. If there are both rational and floating point argument values, then floating point values are coerced to rational values before being compared with rational values.
E.g., $(> 0.5 \ 1/3) \Rightarrow \text{True}$ because $(> 1/2 \ 1/3) \Rightarrow \text{True}$.

$(\geq \dots)$, $(< \dots)$, and $(\leq \dots)$ are analogous to $(> \dots)$:

- Fact 3 is true for \geq , $<$, and \leq as well as $>$.
- We can substitute \geq , $<$, or \leq for $>$ in facts 1 and 2 to get corresponding facts regarding \geq , $<$, and \leq .

Rounding error may lead to unexpected results. For example, $(> 0.2 \ 1/5) \Rightarrow \text{True}$ since the float 0.2 slightly exceeds $1/5$.

Other Useful Numerical Predicates:

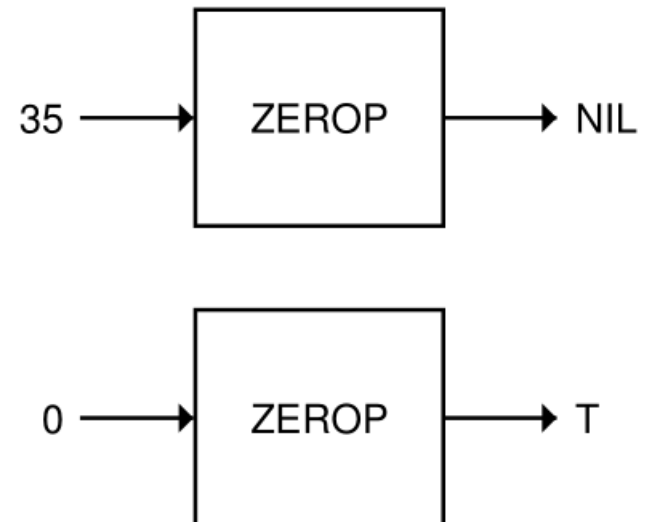
zerop, evenp, oddp, plusp, minusp, /=

If any of these functions is called with an argument whose value is not a number, there will be an **evaluation error**!

- $(\text{zerop } x) = (= x 0)$;
note that $(\text{zerop } 0.0) \Rightarrow \text{T}$!

From p. 9 of Touretzky:

ZEROP returns T if its input is zero.



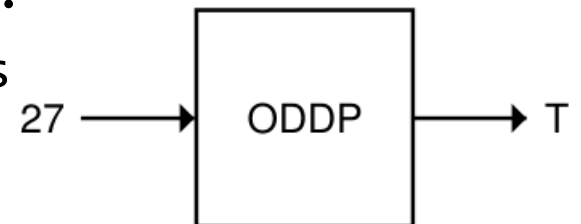
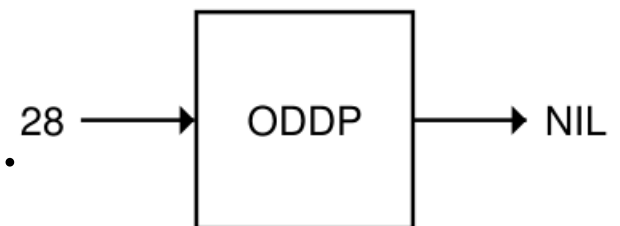
Other Useful Numerical Predicates:

zerop, evenp, oddp, plusp, minusp, /=

If any of these functions is called with an argument whose value is not a number, there will be an **evaluation error**!

- $(\text{zerop } x) = (= x 0)$;
note that $(\text{zerop } 0.0) \Rightarrow \text{T}$!
- $(\text{evenp } n) \Rightarrow \text{T}$ if $n \Rightarrow$ an even integer.
 $(\text{evenp } n) \Rightarrow \text{NIL}$ if $n \Rightarrow$ an odd integer.
- $(\text{oddp } n) \Rightarrow \text{T}$ if $n \Rightarrow$ an odd integer.
 $(\text{oddp } n) \Rightarrow \text{NIL}$ if $n \Rightarrow$ an even integer.
- Calling $(\text{evenp } n)$ or $(\text{oddp } n)$ produces an evaluation error if the argument value is not an integer!

From Touretzky, p. 9:



Other Useful Numerical Predicates:

`zerop`, `evenp`, `oddp`, `plusp`, `minusp`, `/=`

If any of these functions is called with an argument whose value is not a number, there will be an **evaluation error**!

- $(\text{zerop } x) = (= x 0)$;
note that $(\text{zerop } 0.0) \Rightarrow \text{T}$!
- $(\text{evenp } n) \Rightarrow \text{T}$ if $n \Rightarrow$ an even integer.
 $(\text{evenp } n) \Rightarrow \text{NIL}$ if $n \Rightarrow$ an odd integer.
- $(\text{oddp } n) \Rightarrow \text{T}$ if $n \Rightarrow$ an odd integer.
 $(\text{oddp } n) \Rightarrow \text{NIL}$ if $n \Rightarrow$ an even integer.
- Calling $(\text{evenp } n)$ or $(\text{oddp } n)$ produces an evaluation error if the argument value is not an integer!
- $(\text{plusp } x) = (> x 0)$.
- $(\text{minusp } x) = (< x 0)$.
- $(/= x y) = (\text{not } (= x y))$. More generally: If each $z \Rightarrow$ a number then $(/= z_1 \dots z_n) \Rightarrow \text{T}$ just if no two argument values are $=$.

MEMBER: A Built-in Predicate That Never Returns T

Suppose $L \Rightarrow$ a proper list.

Then the value of $(\text{member } x \ L)$ is given by these rules:

- If *no* element of the list is EQL to the value of x , then

$(\text{member } x \ L) \Rightarrow \text{NIL}$

Examples: $(\text{member } 'K \ '(2 \ A \ (9) \ 9 \ A \ B)) \Rightarrow \text{NIL}$

$(\text{member } (\text{list } 9) \ '(2 \ A \ (9) \ 9 \ A \ B)) \Rightarrow \text{NIL}$

because $(\text{eq1 } (\text{list } 9) \ '(9)) \Rightarrow \text{NIL}$.

- If some element of the list is EQL to the value of x , then

$(\text{member } x \ L) \Rightarrow$ the part of the list that begins with the **1st** element that is EQL to the value of x

Examples: $(\text{member } 2 \ '(2 \ A \ (9) \ 9 \ A \ B)) \Rightarrow (2 \ A \ (9) \ 9 \ A \ B)$

$(\text{member } 'A \ '(2 \ A \ (9) \ 9 \ A \ B)) \Rightarrow (A \ (9) \ 9 \ A \ B)$

$(\text{member } 9 \ '(2 \ A \ (9) \ 9 \ A \ B)) \Rightarrow (9 \ A \ B)$

$(\text{member } 'B \ '(2 \ A \ (9) \ 9 \ A \ B)) \Rightarrow (B)$

Note that MEMBER does **not** use T to represent true: It returns *a true value that contains more information!*

From pp. 170 – 1 of Touretzky:

6.6.1 MEMBER

the *first* occurrence of



The MEMBER predicate checks whether an item is a member of a list. If the item is found in the list, the sublist beginning with that item is returned. Otherwise NIL is returned. MEMBER never returns T.

```
> (setf ducks '(huey dewey louie))    Create a set of ducks.  
(HUEY DEWEY LOUIE)
```

```
> (member 'huey ducks)  
(HUEY DEWEY LOUIE)
```

Is Huey a duck?
Non-NIL result: yes.

```
> (member 'dewey ducks)  
(DEWEY LOUIE)
```

Is Dewey a duck?
Non-NIL result: yes.

```
> (member 'louie ducks)  
(LOUIE)
```

Is Louie a duck?
Non-NIL result: yes.

```
> (member 'mickey ducks)  
NIL
```

Is Mickey a duck?
NIL: no.

From p. 171 of Touretzky:

In the very first dialect of Lisp, MEMBER returned just T or NIL. But people decided that having MEMBER return the sublist beginning with the item sought made it a much more useful function. This extension is consistent with MEMBER's being a predicate, because the sublist with *zero* elements is also the only way to say "false."

Here's an example of why it is useful for MEMBER to return a sublist. The BEFOREP predicate returns a true value if *x* appears earlier than *y* in the list *l*.

or (eql x y) and x is in L



```
(defun beforep (x y l)
  "Returns true if X appears before Y in L"
  (member y (member x l)))
```

x and y are assumed to be
symbols, numbers, or characters

```
> (beforep 'not 'whom
      ' (ask not for whom the bell tolls))
(WHOM THE BELL TOLLS)
```

```
> (beforep 'thee 'tolls ' (it tolls for thee))
NIL
```

Recall that: $(\text{member } (\text{list } 9) \text{ '(2 A (9) 9 A B)}) \Rightarrow \text{NIL}$
because $(\text{eql } (\text{list } 9) \text{ '(9)}) \Rightarrow \text{NIL}$.

Can we make MEMBER use EQUAL instead of EQL to test equality?
Yes!

Suppose $L \Rightarrow$ a proper list.

$(\text{member } x \text{ } L \text{ :test #'equal})$ is like $(\text{member } x \text{ } L)$, but looks for an element of the list that is EQUAL to the value of x .

- If *no* element of the list is **EQUAL** to x 's value, then
 $(\text{member } x \text{ } L \text{ :test #'equal}) \Rightarrow \text{NIL}$

Example: $(\text{member 'K '(2 A (9) 9 A B) :test #'equal}) \Rightarrow \text{NIL}$

- If some element of the list is **EQUAL** to x 's value, then
 $(\text{member } x \text{ } L \text{ :test #'equal}) \Rightarrow$ the part of the list that begins with the **1st** element that is **EQUAL** to x 's value

Example: $(\text{member } (\text{list } 9) \text{ '(2 A (9) 9 A B) :test #'equal})$
 $\Rightarrow ((9) \text{ 9 A B})$

From pp. 199 – 200 of Touretzky:

Another function that takes keyword arguments is MEMBER. Normally, MEMBER uses EQL to test whether an item appears in a set. EQL will work correctly for both symbols and numbers. But suppose our set contains lists? In that case we must use EQUAL for the equality test, or else MEMBER won't find the item we're looking for:

```
(setf cards
  '((3 clubs) (5 diamonds) (ace spades)))

(member ' (5 diamonds) cards) ⇒ nil

(second cards) ⇒ (5 diamonds)

(eql (second cards) ' (5 diamonds)) ⇒ nil

(equal (second cards) ' (5 diamonds)) ⇒ t
```

The :TEST keyword can be used with MEMBER to specify a different function for the equality test. We write #'EQUAL to specially quote the function for use as an input to MEMBER.

```
> (member ' (5 diamonds) cards :test #'equal)
((5 DIAMONDS) (ACE SPADES))
```

**IF, COND,
AND, and OR**

Review of the IF Special Operator

From p. 114 of Touretzky:

The IF special function takes three arguments: a **test**, a **true-part**, and a **false-part**. If the test is true, IF returns the value of the true-part. If the test is false, it skips the true-part and instead returns the value of the false-part. Here are some examples.

```
(if (oddp 1) 'odd 'even) ⇒ odd
```

```
(if (oddp 2) 'odd 'even) ⇒ even
```

```
(if t 'test-was-true 'test-was-false) ⇒  
test-was-true
```

```
(if nil 'test-was-true 'test-was-false) ⇒  
test-was-false
```

```
(if (symbolp 'foo) (* 5 5) (+ 5 5)) ⇒ 25
```

```
(if (symbolp 1) (* 5 5) (+ 5 5)) ⇒ 10
```

From p. 114 of Touretzky:

Let's use IF to construct a function that takes the absolute value of a number. Absolute values are always nonnegative. For negative numbers the absolute value is the negation of the number; for positive numbers and zero the absolute value is the number itself. This leads to a simple definition for MY-ABS, our absolute value function. (We call the function MY-ABS rather than ABS because there is already an ABS function built in to Common Lisp; we don't want to interfere with any of Lisp's built-in functions.)

```
(defun my-abs (x)
  (if (< x 0) (- x) x))
```

```
> (my-abs -5)    True-part takes the negation.
5
```

```
> (my-abs 5)     False-part returns the number unchanged.
5
```

- **Recall:** Any value other than NIL represents true!

- (if 0 1 2) \Rightarrow 1
- (if '(D E F) 1 2) \Rightarrow 1
- (if (member 'D '(A B C D E F)) 1 2) \Rightarrow 1

From p. 115 of Touretzky:

Here's another simple decision-making function. SYMBOL-TEST returns a message telling whether or not its input is a symbol.

```
(defun symbol-test (x)
  (if (symbolp x) (list 'yes x 'is 'a 'symbol)
      (list 'no x 'is 'not 'a 'symbol)))
```

When you read this function definition to yourself, you should read the IF part as “If SYMBOLP of X then...else....”

```
> (symbol-test 'rutabaga)           Evaluate true-part.
(YES RUTABAGA IS A SYMBOL)
```

```
> (symbol-test 12345)              Evaluate false-part.
(NO 12345 IS NOT A SYMBOL)
```

IF can be given two inputs instead of three, in which case it behaves as if its third input (the false-part) were the symbol NIL.

```
(if t 'happy) ⇒ happy
```

```
(if nil 'happy) ⇒ nil
```

- (if c e) = (if c e nil)
 - (if t e) = e
 - (if nil e) = nil

An Example of Nested IFs

We'll define a function `KIND-OF-VALUE` with these properties:

- If $e \Rightarrow 0$, then
 $(\text{kind-of-value } e) \Rightarrow (\text{INTEGER-0 } 0)$
- If $e \Rightarrow$ any other number, then
 $(\text{kind-of-value } e) \Rightarrow (\text{NUM-BUT-NOT-0 } \langle e\text{'s value} \rangle)$
- If $e \Rightarrow$ a symbol or any other atom that's not a number, then
 $(\text{kind-of-value } e) \Rightarrow (\text{NON-NUMERIC-ATOM } \langle e\text{'s value} \rangle)$
- If $e \Rightarrow$ a nonempty list, then
 $(\text{kind-of-value } e) \Rightarrow (\text{NONEMPTY-LIST } \langle e\text{'s value} \rangle)$

Examples:

$(\text{kind-of-value } (-\ 3\ 3)) \Rightarrow (\text{INTEGER-0 } 0)$

$(\text{kind-of-value } (-\ 3.0\ 3)) \Rightarrow (\text{NUM-BUT-NOT-0 } 0.0)$

$(\text{kind-of-value } (\text{car } '(-\ 3.0\ 3))) \Rightarrow (\text{NON-NUMERIC-ATOM } -)$

$(\text{kind-of-value } '(-\ 3.0\ 3)) \Rightarrow (\text{NONEMPTY-LIST } (-\ 3.0\ 3))$

An Example of Nested IFs

We'll define a function KIND-OF-VALUE with these properties:

- If $e \Rightarrow 0$, then
 $(\text{kind-of-value } e) \Rightarrow (\text{INTEGER-0 } 0)$
- If $e \Rightarrow$ any other number, then
 $(\text{kind-of-value } e) \Rightarrow (\text{NUM-BUT-NOT-0 } \langle e\text{'s value} \rangle)$
- If $e \Rightarrow$ a symbol or any other atom that's not a number, then
 $(\text{kind-of-value } e) \Rightarrow (\text{NON-NUMERIC-ATOM } \langle e\text{'s value} \rangle)$
- If $e \Rightarrow$ a nonempty list, then
 $(\text{kind-of-value } e) \Rightarrow (\text{NONEMPTY-LIST } \langle e\text{'s value} \rangle)$

```
(defun kind-of-value (e)
  (if (eql 0 e)
      '(integer-0 0)
      (if (numberp e)
          (list 'num-but-not-0 e)
          (if (atom e)
              (list 'non-numeric-atom e)
              (list 'nonempty-list e))))))
```

The COND Macro Operator

The macro form

$$\begin{array}{l} (\text{cond } (g_1 \ e_1) \\ \quad (g_2 \ e_2) \\ \quad \vdots \\ \quad (g_n \ e_n)) \end{array}$$

is equivalent to n nested ifs:

$$\begin{array}{l} (\text{if } g_1 \\ \quad e_1 \\ \quad (\text{if } g_2 \\ \quad \quad e_2 \\ \quad \quad \dots \\ \quad \quad (\text{if } g_n \\ \quad \quad \quad e_n) \dots)) \end{array}$$

- Each $(g_i \ e_i)$ is called a **clause** of the COND.
- In any clause $(g \ e)$:
 - g is called the **test**.
 - e is called the **consequent**.
- If every test \Rightarrow NIL, then the COND's value is **NIL**.

From p. 116 of Touretzky:

Here is how COND works: It goes through the clauses sequentially. If the test part of a clause evaluates to *true*, COND evaluates the consequent part and returns its value; it does not consider any more clauses. If the test evaluates to *false*, COND skips the consequent part and examines the next clause. If it goes through all the clauses without finding any whose test is true, COND returns NIL.

Let's use COND to write a function COMPARE that compares two numbers.

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

- (compare 4 8) ⇒ FIRST-IS-SMALLER
- (compare 9 7.3) ⇒ FIRST-IS-BIGGER
- (compare 7.3 7.3) ⇒ NUMBERS-ARE-THE-SAME
- (compare 7 7.0) ⇒ NIL

Often, the last test is t .

Then we have:

```
(cond (g1 e1)
      (g2 e2)
      ⋮
      (gn-1 en-1)
      (t   en))
```

This is equivalent to

```
(if g1
    e1
    (if g2
        e2
        ...
        (if gn-1
            en-1
            (if t
                en)) ... ))
```

which can be simplified because $(\text{if } t \ e_n) = e_n$.

Often, the last test is t .

Then we have:

```
(cond (g1 e1)
      (g2 e2)
      ⋮
      (gn-1 en-1)
      (t   en))
```

This is equivalent to

```
(if g1
    e1
    (if g2
        e2
        ⋮
        (if gn-1
            en-1
            en) ⋯ )) because (if t en) = en.
```

- So, if the last test is t and all earlier tests have value NIL , then the $COND$'s value is the value of e_n .

The above function definition

```
(defun kind-of-value (e)
  (if (eql 0 e)
      '(integer-0 0)
      (if (numberp e)
          (list 'num-but-not-0 e)
          (if (atom e)
              (list 'non-numeric-atom e)
              (list 'nonempty-list e))))))
```

can be rewritten using COND as follows:

```
(defun kind-of-value (e)
  (cond ((eql 0 e) '(integer-0 0))
        ((numberp e) (list 'num-but-not-0 e))
        ((atom e) (list 'non-numeric-atom e))
        (t (list 'nonempty-list e))))
```