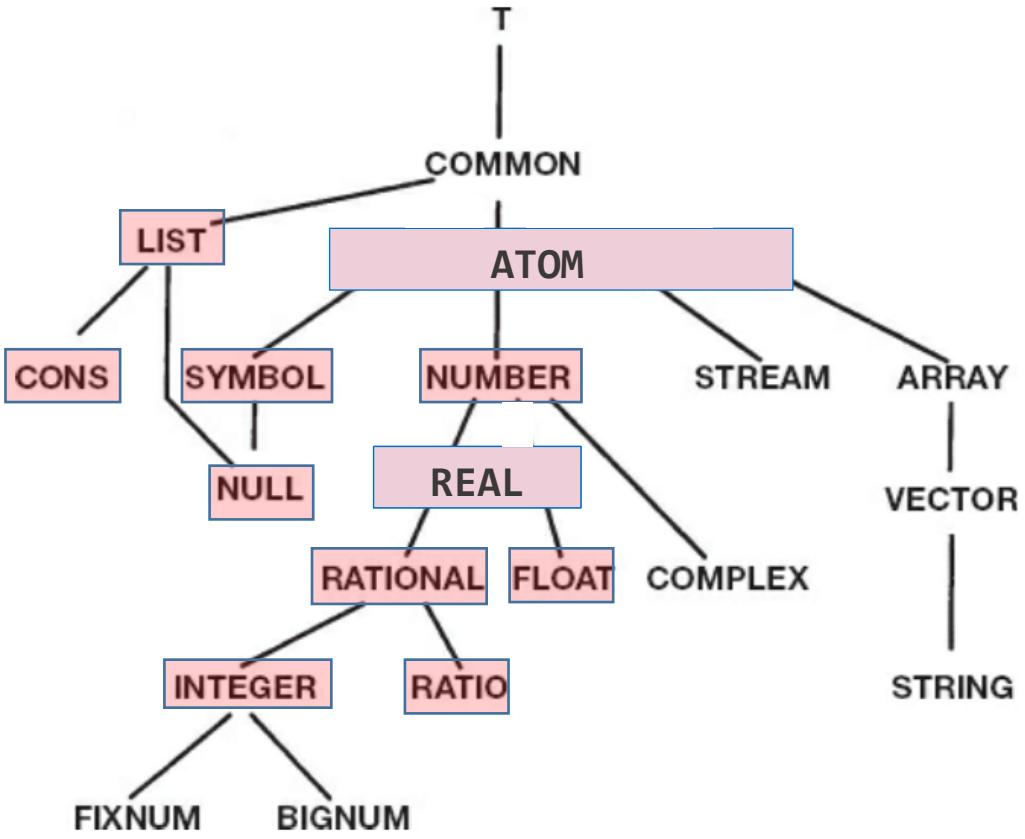
(typep *x* '<type>) ⇒ T if *x* ⇒ a value of type <type>.
(typep *x* '<type>) ⇒ NIL if *x* ⇒ a value whose type is not <type>.



<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 … x_n), (>= x_1 … x_n), (< x_1 … x_n), and (<= x_1 … x_n)

1. If any argument value is not a real number, then evaluation of (> x_1 … x_n) produces an error!
2. If the argument values are all rational or all floating point, then:
   - (> x_1 … x_n) ⇒ T **if** (- x_i x_{i+1}) > 0 for $1 \le i < n$.
   - (> x_1 … x_n) ⇒ 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) ⇒ T because (> 1/2 1/3) ⇒ T.

(>= …), (< …), and (<= …) are analogous to (> …):

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

Rounding error may lead to unexpected results. For example, (> 0.2 1/5) ⇒ T 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**!

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

MEMBER: A Built-in Predicate That Never Returns T

Suppose *L* ⇒ a proper list.
Then the value of (member *x* *L*) is given by these rules:

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

  (member *x* *L*) ⇒ NIL

  Examples: (member 'K '(2 A (9) 9 A B)) ⇒ NIL

  (member (list 9) '(2 A (9) 9 A B)) ⇒ NIL
  because (eql (list 9) '(9)) ⇒ NIL.

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

  (member *x* *L*) ⇒ the part of the list that begins with the ***1st*** element that is **EQL** to the value of *x*

  Examples:
  ```
  (member  2 '(2 A (9) 9 A B)) ⇒ (2 A (9) 9 A B)
  (member 'A '(2 A (9) 9 A B)) ⇒   (A (9) 9 A B)
  (member  9 '(2 A (9) 9 A B)) ⇒         (9 A B)
  (member 'B '(2 A (9) 9 A B)) ⇒           (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))
(HUEY DEWEY LOUIE)
```
Create a set of ducks.

```
> (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: (member (list 9) '(2 A (9) 9 A B)) ⇒ NIL
because (eql (list 9) '(9)) ⇒ NIL.

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

Yes!

Suppose *L* ⇒ a proper list.

(member *x L* **:test #'equal**) is like (member *x 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

  (member *x L* :test #'equal) ⇒ NIL

  Example: (member 'K '(2 A (9) 9 A B) :test #'equal) ⇒ NIL

- If some element of the list is **EQUAL** to *x*'s value, then

  (member *x L* :test #'equal) ⇒ the part of the list that begins with the ***1st*** element that is **EQUAL** to *x*'s value

  Example: (member (list 9) '(2 A (9) 9 A B) :test #'equal)
  ⇒ ((9) 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))
```

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)
5
```
True-part takes the negation.

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

- **Recall**: Any value other than NIL represents true!
  - (if 0 1 2) ⇒ 1
  - (if '(D E F) 1 2) ⇒ 1
  - (if (member 'D '(A B C D E F)) 1 2) ⇒ 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)
(YES RUTABAGA IS A SYMBOL)
```

Evaluate true-part.

```
> (symbol-test 12345)
(NO 12345 IS NOT A SYMBOL)
```

Evaluate false-part.

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* ⇒ 0, then
  (kind-of-value *e*) ⇒ (INTEGER-0 0)
- If *e* ⇒ any other number, then
  (kind-of-value *e*) ⇒ (NUM-BUT-NOT-0 <*e*'s value>)
- If *e* ⇒ a symbol or any other atom that's not a number, then
  (kind-of-value *e*) ⇒ (NON-NUMERIC-ATOM <*e*'s value>)
- If *e* ⇒ a nonempty list, then
  (kind-of-value *e*) ⇒ (NONEMPTY-LIST <*e*'s value>)

Examples:

(kind-of-value (- 3 3)) ⇒ (INTEGER-0 0)

(kind-of-value (- 3.0 3)) ⇒ (NUM-BUT-NOT-0 0.0)

(kind-of-value (car '(- 3.0 3))) ⇒ (NON-NUMERIC-ATOM -)

(kind-of-value '(- 3.0 3)) ⇒ (NONEMPTY-LIST (- 3.0 3))

An Example of Nested IFs

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

- If *e* ⇒ 0, then
  (kind-of-value *e*) ⇒ (INTEGER-0 0)
- If *e* ⇒ any other number, then
  (kind-of-value *e*) ⇒ (NUM-BUT-NOT-0 <*e*'s value>)
- If *e* ⇒ a symbol or any other atom that's not a number, then
  (kind-of-value *e*) ⇒ (NON-NUMERIC-ATOM <*e*'s value>)
- If *e* ⇒ a nonempty list, then
  (kind-of-value *e*) ⇒ (NONEMPTY-LIST <*e*'s value>)

```
(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 (cond $(g_1\ e_1)$ $(g_2\ e_2)$ ⋮ $(g_n\ e_n)$)

is equivalent to n nested ifs:

```
(if g1
    e1
    (if g2
        e2
           ⋱
              (if gn
                  en) ⋯ ))
```

- 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 ⇒ 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`

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))))
```

Another example, from p. 117 of Touretzky:

```
(defun where-is (x)
  (cond ((equal x 'paris) 'france)
        ((equal x 'london) 'england)
        ((equal x 'beijing) 'china)
        (t 'unknown)))
```

From pp. 119 – 20 of Touretzky:

Parenthesis errors can play havoc with COND expressions. Most COND clauses begin with exactly two parentheses. The first marks the beginning of the clause, and the second marks the beginning of the clause's test.

...

If the test part of a clause is just a symbol, not a call to a function, then the clause should begin with a single parenthesis. Notice that in WHERE-IS the clause with T as the test begins with only one parenthesis.

From p. 120 of Touretzky:

Here are two of the more common parenthesis errors made with COND. First, suppose we leave a parenthesis out of a COND clause. What would happen?

```
(cond (equal x 'paris 'france)
      (. . .)
      (. . .)
      (t 'unknown))
```

The first clause of the COND starts with only one left parenthesis instead of two. As a result, the test part of this clause is just the symbol EQUAL. When the test is evaluated, EQUAL will cause an unassigned variable error.

On the other hand, consider what happens when too many parentheses are used:

```
(cond ((. . .) 'france)
      ((. . .) 'england)
      ((. . .) 'china)
      ((t 'unknown)))
```

If X has the value HACKENSACK, we will reach the fourth COND clause. Due to the presence of an extra pair of parentheses in this clause, the test is (T 'UNKNOWN) instead of simply T. T is not a function, so this test will generate an undefined function error.