Lisp implementations deal with S-expressions via pointers:

- When passing an S-expression as argument to a function call or returning an S-expression as the result of a function call, *what do we actually pass or return*?

  Answer: We pass or return a ***pointer*** *to the S-expression's data object*.

  - In some implementations this may not actually be true when the S-expression in question is a number or a character, but even then it's OK for Lisp programmers to assume it's true, as that assumption won't lead to wrong predictions about the *observable behavior* of the code.

Lisp implementations deal with S-expressions via pointers:

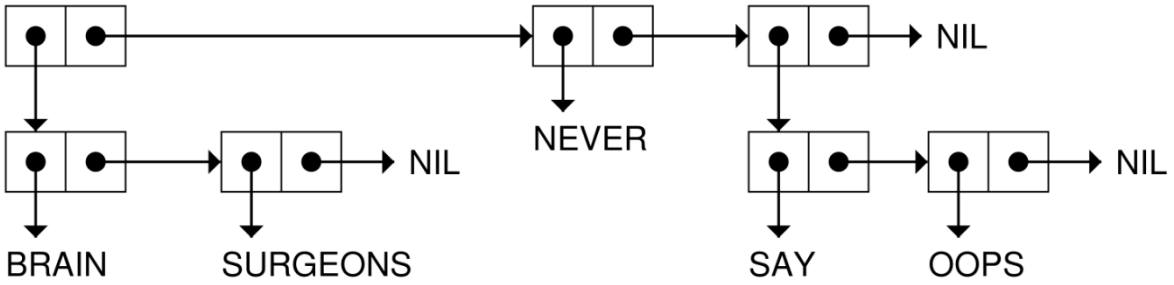
- When passing an S-expression as argument to a function call or returning an S-expression as the result of a function call, *what do we actually pass or return*?

  Answer: We pass or return a ***pointer to the S-expression's data object***.

- Every nonempty list has a "root" cons cell, which is *the unique cons cell of the list that is **not** pointed at by a car or cdr pointer of another cons cell of the list.*

  Example:

This cons cell is the "root" cons cell of the list.



- When passing a nonempty list as argument to a function call, or returning a nonempty list as the result of a function call, we pass or return ***a pointer to the "root" cons cell of that list***.

Predicates and Conditionals

- In Lisp, a ***predicate*** is a function whose calls return values that represent ***true*** or ***false***.
- In Lisp (i.e., Common Lisp):
  - ***False*** is represented by the symbol NIL.
  - ***True*** can be represented by ***any value other than*** NIL:
    - **T**, **19.5**, **0**, **""**, **DOG**, and **(A (B C) D)** all represent *true*!
  - The symbol T is the *usual* way to represent ***true***: Use some other value only if there's a good reason!

  Exercise: What is the value of **(if 0 1 2)** in Lisp?

  Answer: As 0 represents *true*, the value is 1.

  Exercise: What is the value of **(if () 1 2)** in Lisp?

  Answer: As () is the same as the symbol NIL, the value is 2.

  - **In Scheme,** T and NIL have no predefined meaning--**in Scheme,** #t and #f represent *true* and *false*, and () doesn't mean *false*!
- Recall that T and NIL are constant symbols of Lisp that evaluate to themselves: So T and NIL never have to be quoted, just as numbers never have to be quoted!

Equality Predicates

Equality Predicates

- Lisp has several predicates for testing equality, of which the following four are the most commonly used:
  - equal
  - eql
  - eq
  - =
- (equal *x y*) ⇒ T if the argument values are equal
  (equal *x y*) ⇒ NIL if the argument values are not equal
  - (equal (car '(a b c)) (cadr '(1 a b c))) ⇒ T
  - (equal (cdr '(a b c)) (cdr '(1 a b c))) ⇒ NIL
  - (equal (list 'a 'b 'c) (cdr '(1 a b c))) ⇒ T
  - (equal (+ 1 2) 3) ⇒ T
  - (equal (+ 1 2) 3.0) ⇒ NIL
  - (equal (+ 1.0 2) 3.0) ⇒ T
  - (equal 0.5 1/2) ⇒ NIL
  - (equal (/ 1 2) 1/2) ⇒ T
  - (equal 0.5 (/ 1 2)) ⇒ NIL

- (equal *x y*) ⇒ T if the argument values are equal
  (equal *x y*) ⇒ NIL if the argument values are not equal
  - (equal (cdr '(a b c)) (cdr '(1 a b c))) ⇒ NIL
  - (equal (list 'a 'b 'c) (cdr '(1 a b c))) ⇒ T
- (eq *x y*) = (equal *x y*) if *x* ⇒ a symbol or *y* ⇒ a symbol.
  Otherwise:
  1. (eq *x y*) ⇒ NIL if (equal *x y*) ⇒ NIL
  2. (eq *x y*) ⇒ T or NIL if (equal *x y*) ⇒ T

  Explanation of fact 2:
  (eq *x* y) compares the ***pointers*** passed as arguments:
  - **(eq *x y*) ⇒ T** ***if the pointers are the same--i.e., x and y refer to the same identical data object.***
    (eq *x y*) ⇒ NIL ***if the pointers are not the same--i.e., x and y refer to 2 distinct data objects.***
  - ***If x and y refer to 2 distinct data objects, we may still have that* (equal *x y*) ⇒ T, as in this case:**
  - When *x*, *y* ⇒ nonempty lists, **(equal *x y*)** *compares the lists' contents* like ***x.equals(y)*** in Java, but **(eq *x y*)** is like ***x == y*** in Java.

- (eq *x y*) = (equal *x y*) if *x* ⇒ a symbol or *y* ⇒ a symbol. Otherwise:
  1. (eq *x y*) ⇒ NIL if (equal *x y*) ⇒ NIL
  2. (eq *x y*) ⇒ T or NIL if (equal *x y*) ⇒ T

  Explanation of fact 2:
  (eq *x* y) compares the ***pointers*** passed as arguments:
  - **(eq *x y*) ⇒ T** ***if the pointers are the same--i.e., x and y refer to the same identical data object.***
    (eq *x y*) ⇒ NIL ***if the pointers are not the same--i.e., x and y refer to 2 distinct data objects.***
  - ***If x and y refer to 2 distinct data objects, we may still have that* (equal *x y*) ⇒ T.**
  - When *x*, *y* ⇒ nonempty lists, **(equal *x y*)** *compares the lists' contents* like ***x.equals(y)*** in Java, but **(eq *x y*)** is like ***x == y*** in Java.

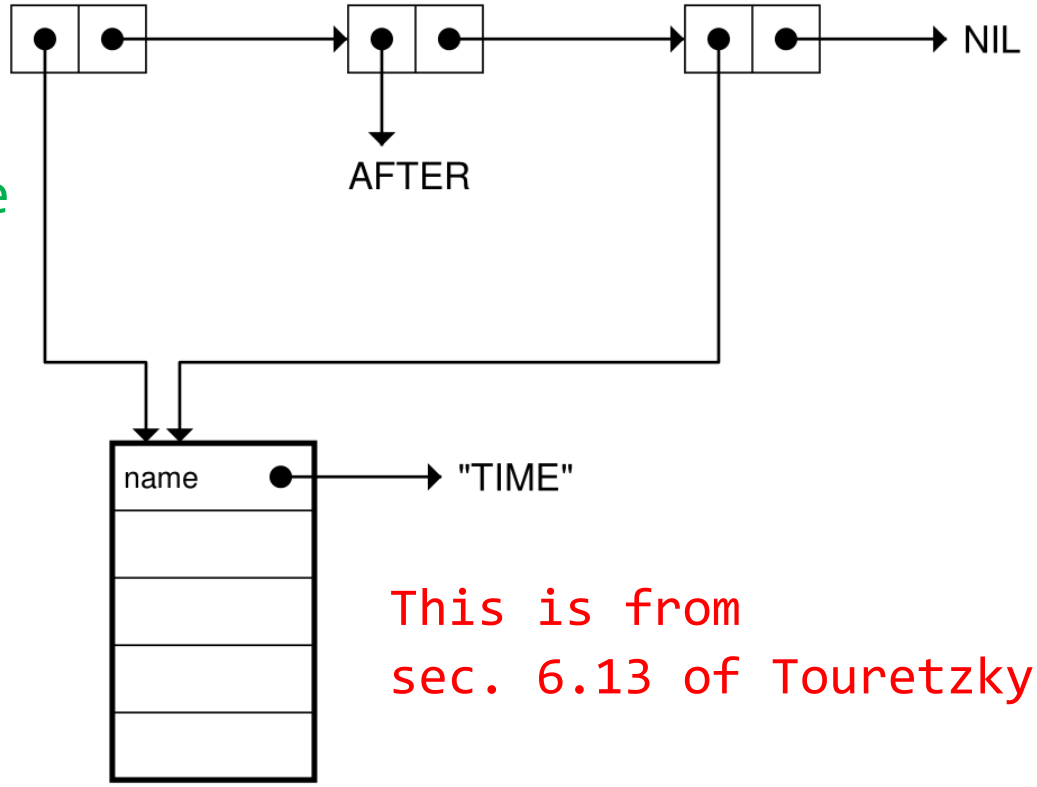
Examples

- (eq (cons 2 '(a)) (cons 2 '(a))) ⇒ NIL
- (eq (first '(a b c)) (fourth '(d c b a))) ⇒ T
  because ***symbols are memory unique***!

Memory Uniqueness of Symbols

In Lisp, symbols are unique, meaning there can be only one symbol in the computer's memory with a given name.** Every object in the memory has a numbered location, called its **address**. Since a symbol exists in only one place in memory, symbols have unique addresses. So in the list (TIME AFTER TIME), the two occurrences of the symbol TIME must refer to the same address. There cannot be two separate symbols named TIME.

The following more detailed depiction of the data structure represented by **(TIME AFTER TIME)** is given on p. 196 of Touretzky:



There is ***just one*** TIME ***symbol object***!

This is from sec. 6.13 of Touretzky.

- (eq *x y*) = (equal *x y*) if *x* ⇒ a symbol or *y* ⇒ a symbol. Otherwise:
  1. (eq *x y*) ⇒ NIL if (equal *x y*) ⇒ NIL
  2. (eq *x y*) ⇒ T or NIL if (equal *x y*) ⇒ T

  Explanation of fact 2:
  (eq *x* y) compares the ***pointers*** passed as arguments:
  - **(eq *x y*) ⇒ T** ***if the pointers are the same--i.e., x and y refer to the same identical data object.***
    (eq *x y*) ⇒ NIL ***if the pointers are not the same--i.e., x and y refer to 2 distinct data objects.***
  - ***If x and y refer to 2 distinct data objects, we may still have that* (equal *x y*) ⇒ T.**
  - When *x*, *y* ⇒ nonempty lists, **(equal *x y*)** *compares the lists' contents* like ***x.equals(y)*** in Java, but **(eq *x y*)** is like ***x == y*** in Java.

Examples

- (eq (cons 2 '(a)) (cons 2 '(a))) ⇒ NIL
- (eq (first '(a b c)) (fourth '(d c b a))) ⇒ T
  because ***symbols are memory unique***!

- (eq *x y*) = (equal *x y*) if *x* ⇒ a symbol or *y* ⇒ a symbol. Otherwise:
  1. (eq *x y*) ⇒ NIL if (equal *x y*) ⇒ NIL
  2. (eq *x y*) ⇒ T or NIL if (equal *x y*) ⇒ T

  (eq *x* y) compares the ***pointers*** passed as arguments:
  - **(eq *x y*) ⇒ T** ***if the pointers are the same--i.e., x and y refer to the same identical data object.***

    (eq *x y*) ⇒ NIL ***if the pointers are not the same--i.e., x and y refer to 2 distinct data objects.***

- If the two arguments values are ***equal numbers***, then the result of (eq *x y*) is implementation dependent!

  Examples [! is a predefined factorial function in clisp.]
  - (eq (! 11) (! 11)) ⇒ NIL in clisp on a PC, but ⇒ T on mars.
  - (eq 3.0 3.0) ⇒ T in sbcl on a PC, but ⇒ NIL in cl on mars.

- **Rule of Thumb:** Use (eq *x y*) ***only when you know at least one of the two argument values is a symbol***.
  - In this case (eq *x y*) = (equal *x y*) but (eq *x y*) is a little faster.

From p. 196 of Touretzky:

If the corresponding elements of two lists are equal, then the lists themselves are considered equal.

```
> (setf x1 (list 'a 'b 'c))          Make a fresh list (A B C).
(A B C)

> (setf x2 (list 'a 'b 'c))          Make another list (A B C).
(A B C)

> (equal x1 x2)                      The lists are EQUAL.
T
```

If we want to tell whether two pointers point to the same object, we must compare their addresses. The EQ predicate (pronounced ‘‘eek’’) does this. Lists are EQ to each other only if they have the same address; no element by element comparison is done.

```
> (eq x1 x2)                         The two lists are not EQ.
NIL
```

From p. 197 of Touretzky:

```
> (setf z x1)
(A B C)
```
Now Z points to the same list as X1.

```
> (eq z x1)
T
```
So Z and X1 are EQ.

```
> (eq z '(a b c))
NIL
```
These lists have different addresses.

```
> (equal z '(a b c))
T
```
But they have the same elements.

The EQ function is faster than the EQUAL function because EQ only has to compare an address against another address, whereas EQUAL has to first test if its inputs are lists, and if so it must compare each element of one against the corresponding element of the other.

Numbers have different internal representations in different Lisp systems. In some implementations each number has a unique address, whereas in others this is not true. Therefore EQ should never be used to compare numbers.

- (eq *x* *y*) = (equal *x* *y*) if *x* ⇒ a symbol or *y* ⇒ a symbol.
  Otherwise:
  1. (eq *x* *y*) ⇒ NIL if (equal *x* *y*) ⇒ NIL
  2. (eq *x* *y*) ⇒ T or NIL if (equal *x* *y*) ⇒ T

  (eq *x* y) compares the ***pointers*** passed as arguments:
  - **(eq *x* *y*) ⇒ NIL *if the pointers are not the same*.**

- **Rule of Thumb:** Use (eq *x* *y*) ***only when you know at least one of the two argument values is a symbol***.

- (eql *x* *y*) = (equal *x* *y*) if *x* ⇒ a symbol, number, or char
  or *y* ⇒ a symbol, number, or char.

  (eql *x* *y*) = (eq *x* *y*) otherwise.

- EQL is a *more stringent* equality test than EQUAL
  but is a *less stringent* equality test than EQ:
  - If (equal *x* *y*) ⇒ NIL then (eq *x* *y*) ⇒ NIL
    and so (eql *x* *y*) ⇒ NIL as well.
  - If (eq *x* *y*) ⇒ T then (equal *x* *y*) ⇒ T
    and so (eql *x* *y*) ⇒ T as well.

- **Rule of Thumb:** Use (eq *x y*) ***only when you know at least one of the two argument values is a symbol***.

- (eql *x y*) = (equal *x y*) if *x* ⇒ a symbol, number, or char
  or *y* ⇒ a symbol, number, or char.
  (eql *x y*) = (eq *x y*) otherwise.

- EQL is a *more stringent* equality test than EQUAL
  but is a *less stringent* equality test than EQ:
  - If (equal *x y*) ⇒ NIL then (eq *x y*) ⇒ NIL
    and so (eql *x y*) ⇒ NIL as well.
  - If (eq *x y*) ⇒ T then (equal *x y*) ⇒ T
    and so (eql *x y*) ⇒ T as well.

- **Examples** [! is a predefined factorial function in clisp.]
  - (eql 3.0 3.0) ⇒ **T**
  - (eql 3 3.0) ⇒ **NIL**
  - (eql (! 20) (! 20)) ⇒ **T**
  - (eql (list 1) (list 1)) ⇒ **NIL**

- **Rule of Thumb:** Use (eql *x y*) ***only when you know at least one of the two argument values is a symbol, number, or character***.

From p. 197 of Touretzky:

The EQL predicate is a slightly more general variant of EQ. It compares the addresses of objects like EQ does, except that for two numbers of the same type (for example, both integers), it will compare their values instead. Numbers of different types are not EQL, even if their values are the same.

```
(eql 'foo 'foo)   ⇒  t

(eql 3 3)   ⇒  t

(eql 3 3.0)   ⇒  nil        Different types.
```

EQL is the ‘‘standard’’ comparison predicate in Common Lisp. Functions such as MEMBER and ASSOC that contain implicit equality tests do them using EQL unless told to use some other predicate.

- In **Scheme**, the analogs of equal, eql, and eq are named **equal?**, **eqv?**, and **eq?**, but member and assoc use equal? rather than eqv? to test equality.

(= x_1 … x_n) can be evaluated only if the value of each of the arguments is a number.

- If any argument value is not a number, then evaluation of (= x_1 … x_n) ***produces an error***!
- If the argument values are all rational or all floating point, then:
  - (= x_1 … x_n) ⇒ T if the argument values are all equal.
  - (= x_1 … x_n) ⇒ NIL otherwise.
- 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 1/2) ⇒ T even though (equal 0.5 1/2) ⇒ NIL.
- Coercing floating point values to rational ***involves no rounding***, as any floating pt. value is mathematically equal to a rational value. This is better than rounding rational values to floating pt. because unequal rationals may round to the same floating pt. value—just as 333/1000 and 3331/10000 would both round to 0.333 if floats were represented with 3 ***decimal*** digits of precision: If = rounded rationals to floating point, there'd be pairs of ***unequal*** rationals that'd each be = to ***the same*** floating point value, so = would not be transitive!

(= x_1 … x_n) can be evaluated only if the value of each of the arguments is a number.

- If any argument value is not a number, then evaluation of (= x_1 … x_n) ***produces an error***!
- If the argument values are all rational or all floating point, then:
  - (= x_1 … x_n) ⇒ T if the argument values are all equal.
  - (= x_1 … x_n) ⇒ NIL otherwise.
- 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 1/2) ⇒ T even though (equal 0.5 1/2) ⇒ NIL.
- When there's a floating-point argument value, (= x_1 … x_n) may unexpectedly return NIL because of rounding error!

  Example: Clisp FLOATs are stored to a precision of 24 significant bits. 1/5 = 0.2 (decimal) is 0.00110011001100110011001100 ... in binary, which rounds to the float 0.00 1100 1100 1100 1100 1100 1101. Thus the float representing 0.2 ***slightly exceeds*** 1/5; so (= 0.2 1/5) ⇒ NIL.

Some Frequently Used Predicates

- (not *x*) = (null *x*) = (eq *x* nil)

  NOT and NULL are equivalent but are used differently:

  - (not *x*) is used to negate a boolean expression *x*, as in
    ```
    (if (not (eql x 1)) ...
    ```
  - (null *x*) is used to test if *x* ⇒ the empty list.
    - (null 17) ⇒ NIL
    - (null (cdr '(17))) ⇒ T
    - (null (cdr L)) ⇒ T if L ⇒ a proper list of length ≤ 1.
      (null (cdr L)) ⇒ NIL if L ⇒ a list of length ≥ 2.

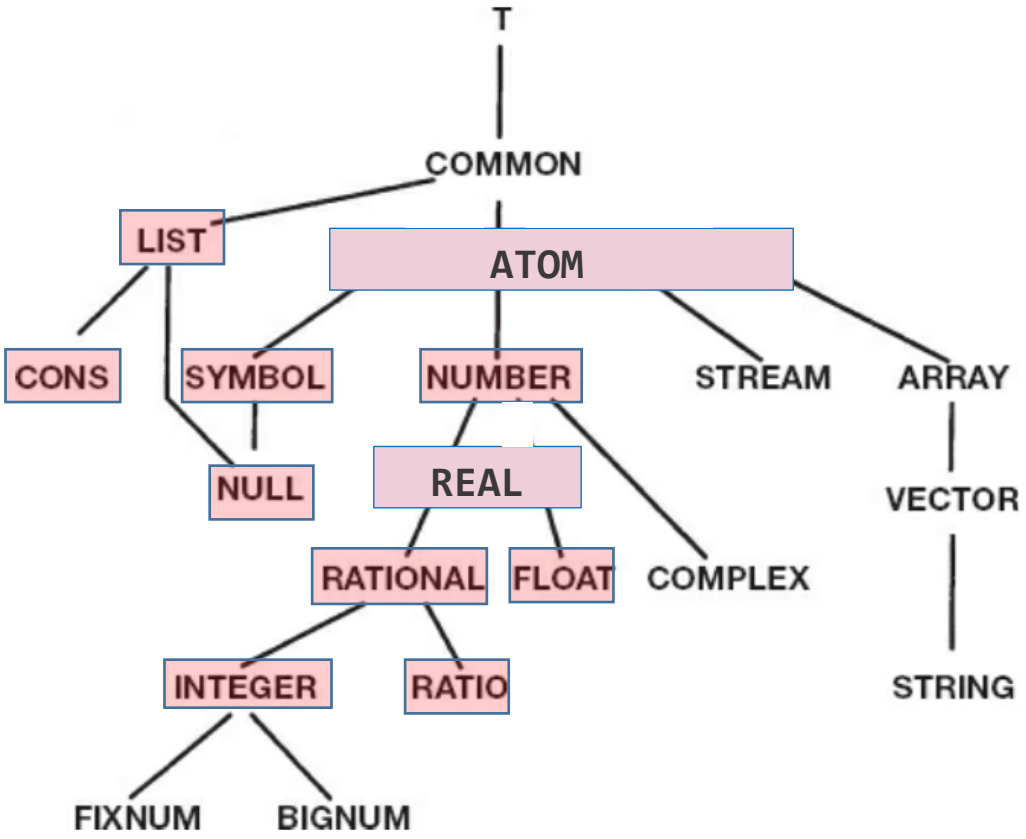
The NULL predicate returns T if its input is NIL. Its behavior is the same as the NOT predicate. By convention, Lisp programmers reserve NOT for logical operations: changing *true* to *false* and *false* to *true*. They use NULL when they want to test whether a list is empty. **[From Touretzky, p. 67.]**

ENDP is a variant of NULL that produces an evaluation ***error*** if the argument value is not a list:

- If *x* ⇒ a list, then (endp *x*) = (null *x*).
- Otherwise, evaluation of (endp *x*) produces an error.
  E.g., evaluation of (endp 7) or (endp 'a) produces an error.

(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.



From p. 367 of Touretzky (with **ATOM** and **REAL** types added)

Figure 12-1 A portion of the Common Lisp type hierarchy.

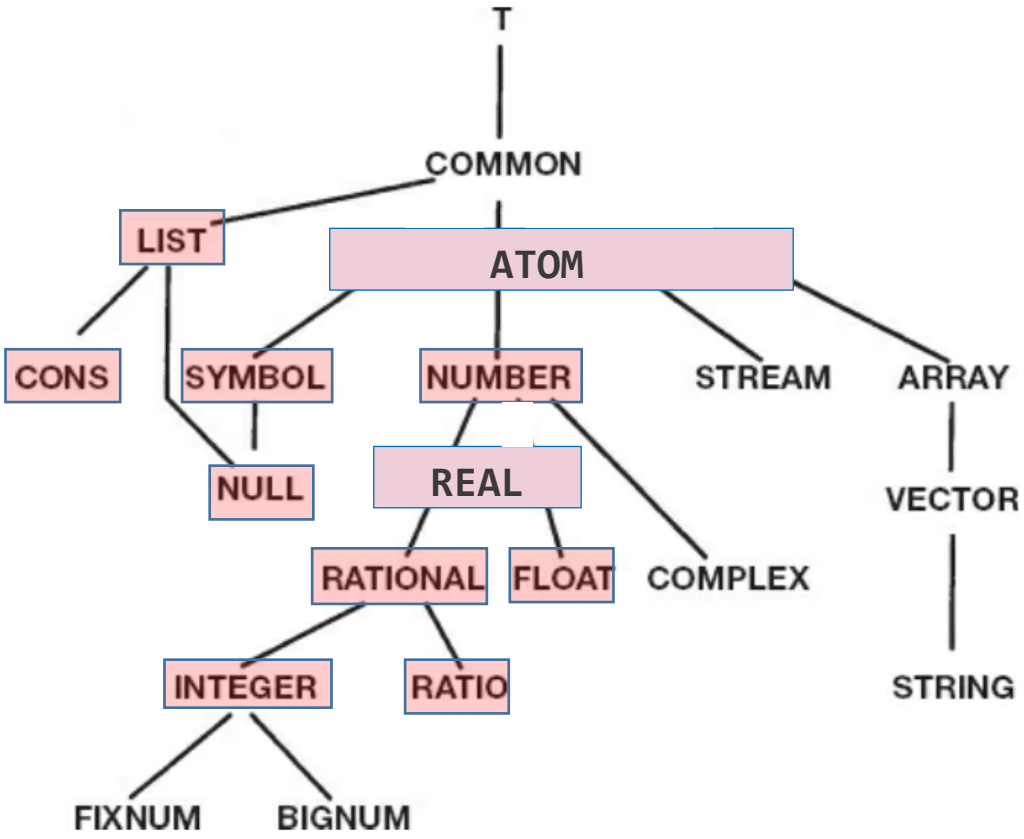
From p. 366 of Touretzky:

The TYPEP predicate returns true if an object is of the specified type. Type specifiers may be complex expressions, but we will only deal with simple cases here.

```
(typep 3 'number)   ⇒  t

(typep 3 'integer)  ⇒  t

(typep 3 'float)    ⇒  nil

(typep 'foo 'symbol)  ⇒  t
```

Figure 12-1 shows a portion of the Common Lisp type hierarchy. This diagram has many interesting features. T appears at the top of the hierarchy, because all objects are instances of type T, and all types are subtypes of T. ~~Type COMMON includes all the types that are built in to Common Lisp.~~ Type NULL includes only the symbol NIL. Type LIST subsumes the types CONS and NULL. NULL is therefore a subtype of both SYMBOL and LIST.

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



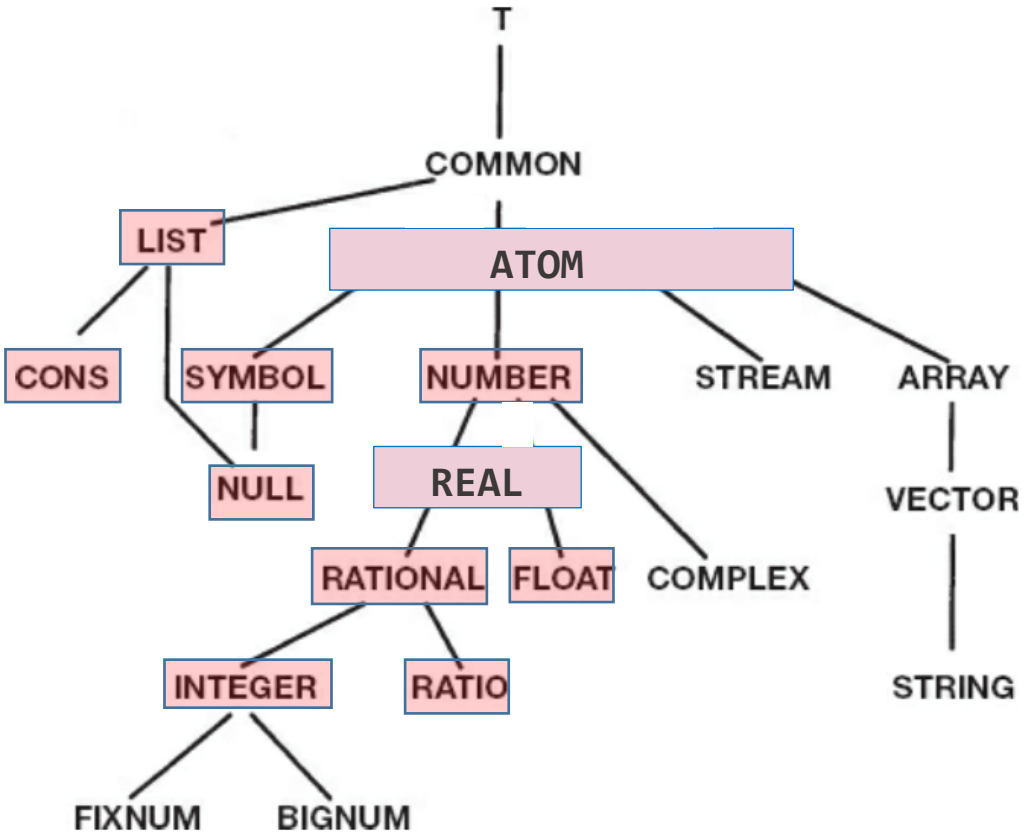
<type> *can be any of the type names shown in the tree on the right* except for COMMON, which is now obsolete.

The 11 boxed types **LIST, ATOM, CONS, SYMBOL, NUMBER, NULL, REAL, RATIONAL, FLOAT, INTEGER, and RATIO** will be used in this course. We'll also use **STRING**s, but only as filenames.

From p. 367 of Touretzky (with ATOM and REAL types added)

Figure 12-1 A portion of the Common Lisp type hierarchy.

(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.

For 8 of the 11 boxed types (all ***except*** **ATOM, NULL,** and **RATIO**),

```
    (<type>p x)
 =  (typep x '<type>).
```

Example:

```
    (integerp x)
 =  (typep x 'integer)
```

From p. 367 of Touretzky (with ATOM and REAL types added)

Figure 12-1 A portion of the Common Lisp type hierarchy.