- `(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`

- `(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`

- (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.
  2.

  Explanation of fact 2:

- (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.

  Explanation of fact 2:

- (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:

- (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***

    (eq *x* y) ⇒ NIL ***if***
  -
  -

- (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 separate data objects.***
  -
  -

- (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 separate data objects.***
  - ***If x and y refer to 2 separate data objects, we may still have that* (equal *x* y) ⇒ T, as in this case:**
  -

- (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 <u>not</u> the same--i.e., x and y refer to 2 separate data objects.***
  - ***If x and y refer to 2 separate data objects, we may still have that*** **(equal *x* y) ⇒ T, as in this case:**
  - When *x*, *y* ⇒ lists, ***(eq x y)*** is like ***x == y*** in Java, but **(equal *x* *y*) tests if the lists have the same contents.**

- (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 separate data objects.***
  - ***If x and y refer to 2 separate data objects, we may still have that*** **(equal *x* y) ⇒ T, as in this case:**
  - When *x*, *y* ⇒ lists, **(eq *x* *y*)** is like ***x* == *y*** in Java, but **(equal *x* *y*) tests if the lists have the same contents.**

- (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 separate data objects.***
  - ***If x and y refer to 2 separate data objects, we may still have that*** **(equal *x* y) ⇒ T, as in this case:**
  - When *x*, *y* ⇒ lists, **(eq *x* *y*)** is like ***x* == *y*** in Java, but **(equal *x* *y*) tests if the lists have the same contents.**

- (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 separate data objects.***
  - ***If x and y refer to 2 separate data objects, we may still have that* (equal *x* y) ⇒ T, as in this case:**
  - When *x*, *y* ⇒ lists, **(eq *x* *y*)** is like ***x* == *y*** in Java, but **(equal *x* *y*) tests if the lists have the same contents.**

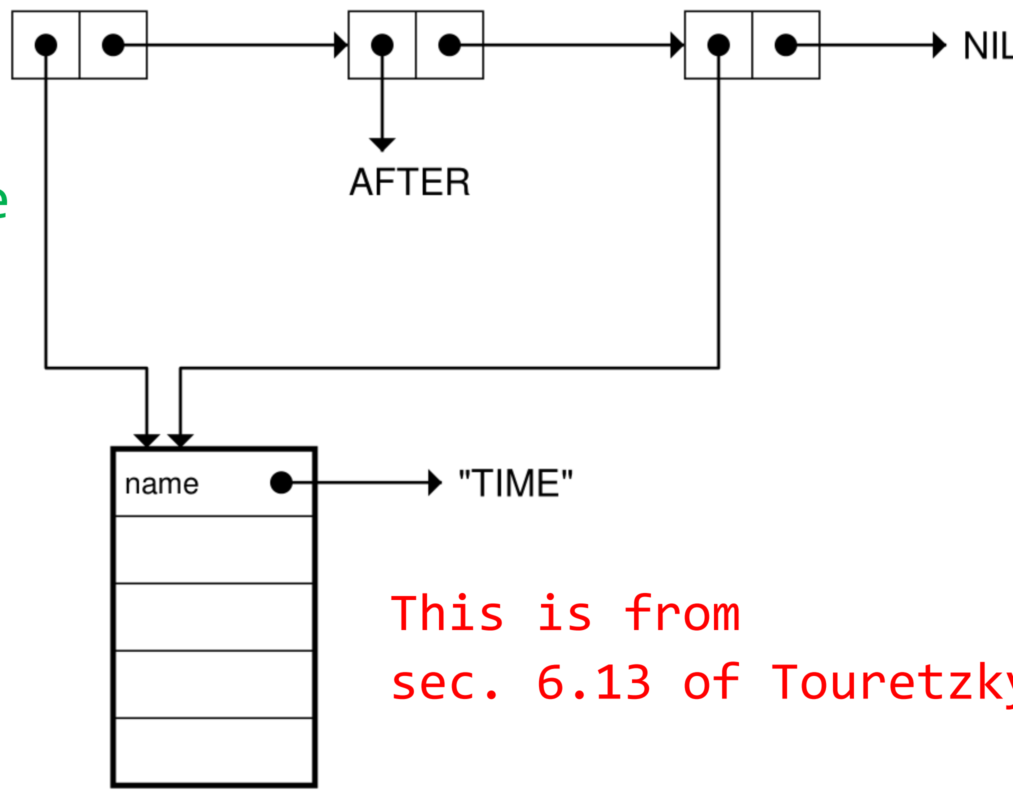
Examples
- (eq (cons 2 '(a)) (cons 2 '(a))) ⇒ NIL
- (eq (first '(a b c)) (fourth '(d c b a))) ⇒ T
  because ***symbols are memory unique***!

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 separate data objects.***
  - ***If x and y refer to 2 separate data objects, we may still have that* (equal *x* y) ⇒ T, as in this case:**
  - When *x*, *y* ⇒ lists, **(eq *x* *y*)** is like ***x* == *y*** in Java, but **(equal *x* *y*) tests if the lists have the same contents.**

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 separate data objects.***

- (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 separate data objects.***

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

  Examples

-

- (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 separate 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)) ⇒ T or NIL: Try on a PC and venus.
  -
-

- (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 separate 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)) ⇒ T or NIL: Try on a PC and venus.
  - (eq 3.0 3.0) ⇒ T or NIL: Try in cl vs clisp on venus.
-

- (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 separate 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)) ⇒ T or NIL: Try on a PC and venus.
  - (eq 3.0 3.0) ⇒ T or NIL: Try in cl vs clisp on venus.
- **Rule of Thumb:** Use (eq *x* *y*) ***only when you know at least one of the two argument values is a symbol***.
  -

- (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 separate 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)) ⇒ T or NIL: Try on a PC and venus.
  - (eq 3.0 3.0) ⇒ T or NIL: Try in cl vs clisp on venus.
- **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

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.

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***.

- (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.
-

- (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:
  -
  -

- (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*) ⇒
    and so (eql *x* *y*) ⇒
  - If (eq *x* *y*) ⇒ T then (equal *x* *y*) ⇒
    and so (eql *x* *y*) ⇒

- (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*) ⇒
    and so (eql *x* *y*) ⇒

- (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.

- **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*) ***<u>only when you know at least one of the two argument values is a symbol</u>***.

- (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) ⇒
  - (eql 3 3.0) ⇒
  - (eql (! 20) (! 20)) ⇒
  - (eql (list 1) (list 1)) ⇒

-

- **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) ⇒
  - (eql (! 20) (! 20)) ⇒
  - (eql (list 1) (list 1)) ⇒

-

- **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) ⇒
  - (eql (! 20) (! 20)) ⇒ **T**
  - (eql (list 1) (list 1)) ⇒

-

- **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)) ⇒

-

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

- (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 (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.

•

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***!
-
  -
  -
-
-

$(= x_1 \ldots x_n)$ 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 $(= x_1 \ldots x_n)$ ***produces an error***!
- If the argument values are all rational or all floating point, then:
  - $(= x_1 \ldots x_n) \Rightarrow$ T if the argument values are all equal.
  - $(= x_1 \ldots x_n) \Rightarrow$ NIL otherwise.
-
-

`(=` 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.,
-

`(=` 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`.
-

(= 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:

$(= x_1 \dots x_n)$ 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 $(= x_1 \dots x_n)$ ***produces an error***!
- If the argument values are all rational or all floating point, then:
  - $(= x_1 \dots x_n)$ ⇒ T if the argument values are all equal.
  - $(= x_1 \dots x_n)$ ⇒ 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, $(= x_1 \dots x_n)$ may unexpectedly return NIL because of rounding error!

  Example: FLOATs are stored to a precision of 24 significant bits.
  In ***binary***, 2/10 = 1/5 = 0.0011001100110011001100110011001100 ...
  which rounds to the float 0.00110011001100110011001101.
  Thus the float 0.2 ***slightly exceeds*** 1/5, and 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*) = (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 (equal x 'dog)) ...`

  - (null *x*) is used to test if *x* ⇒ the empty list.
    - (null 17) ⇒
    - (null (cdr '(17))) ⇒
    - (null (cdr L)) ⇒ T if L ⇒ a proper list of length ≤

      (null (cdr L)) ⇒ NIL if L ⇒ a list of length ≥

- (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 (equal x 'dog)) ...`
  - (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 ≤

      (null (cdr L)) ⇒ NIL if L ⇒ a list of length ≥

- `(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 (equal x 'dog)) ...`

  - `(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.

- `(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 (equal x 'dog)) ...`

  - `(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.]**

- `(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 (equal x 'dog)) ...`
  - `(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:

-
-

- (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 (equal x 'dog)) ...
    ```

  - (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.

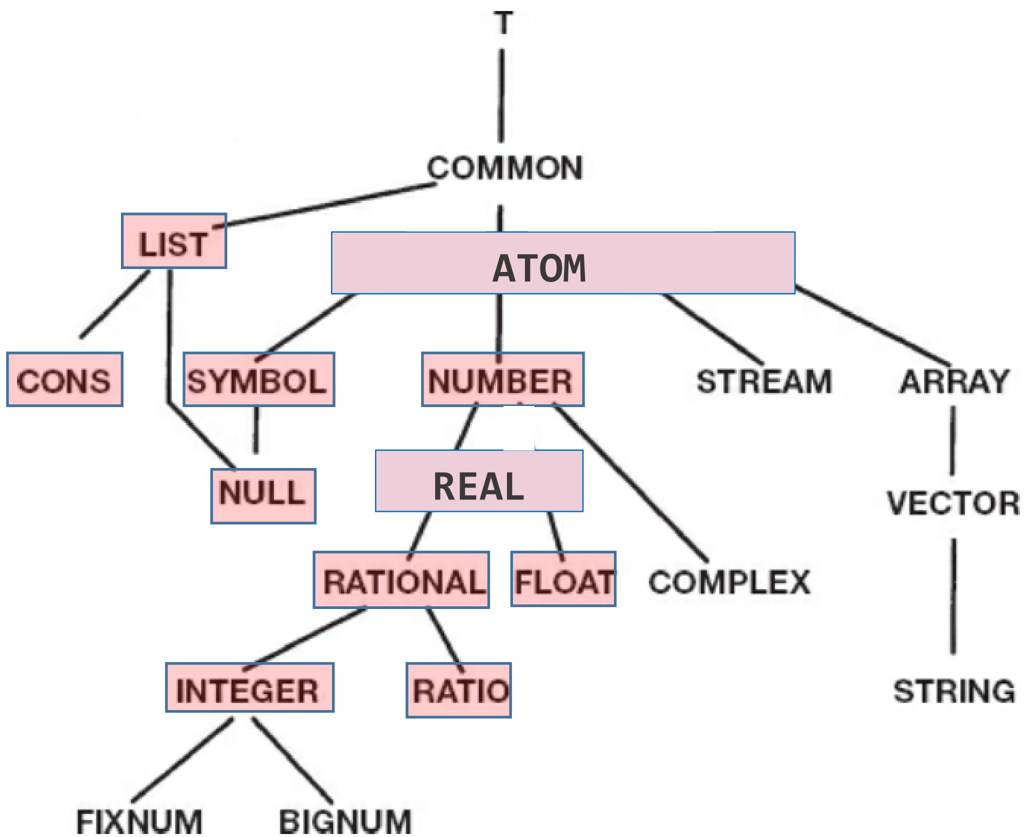
  E.g., evaluation of (endp 7) or (endp 'a) produces an error.

(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.

(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)  ⇒

(typep 3 'integer)  ⇒

(typep 3 'float)  ⇒

(typep 'foo 'symbol)  ⇒
```

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
```

From p. 366 of Touretzky:

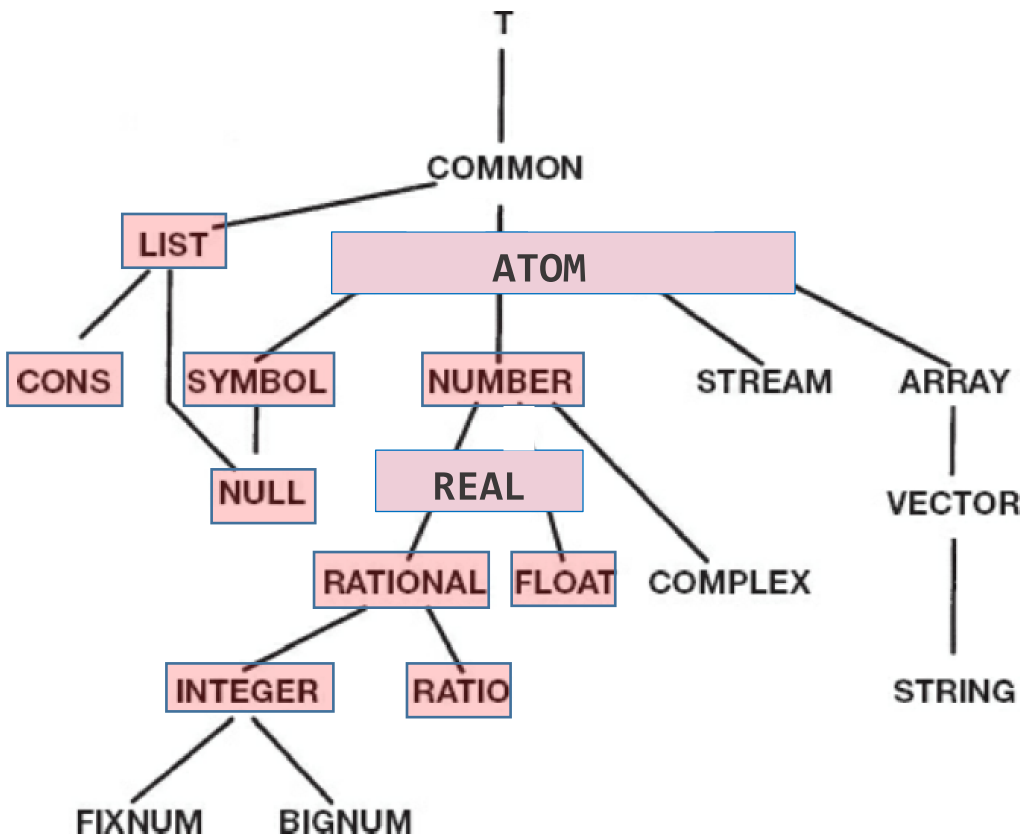
The TYPEP predicate returns true if an object is of the specified type. Type specifiers may be complex expressions, but we will only deal with simple cases here.

```
(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.



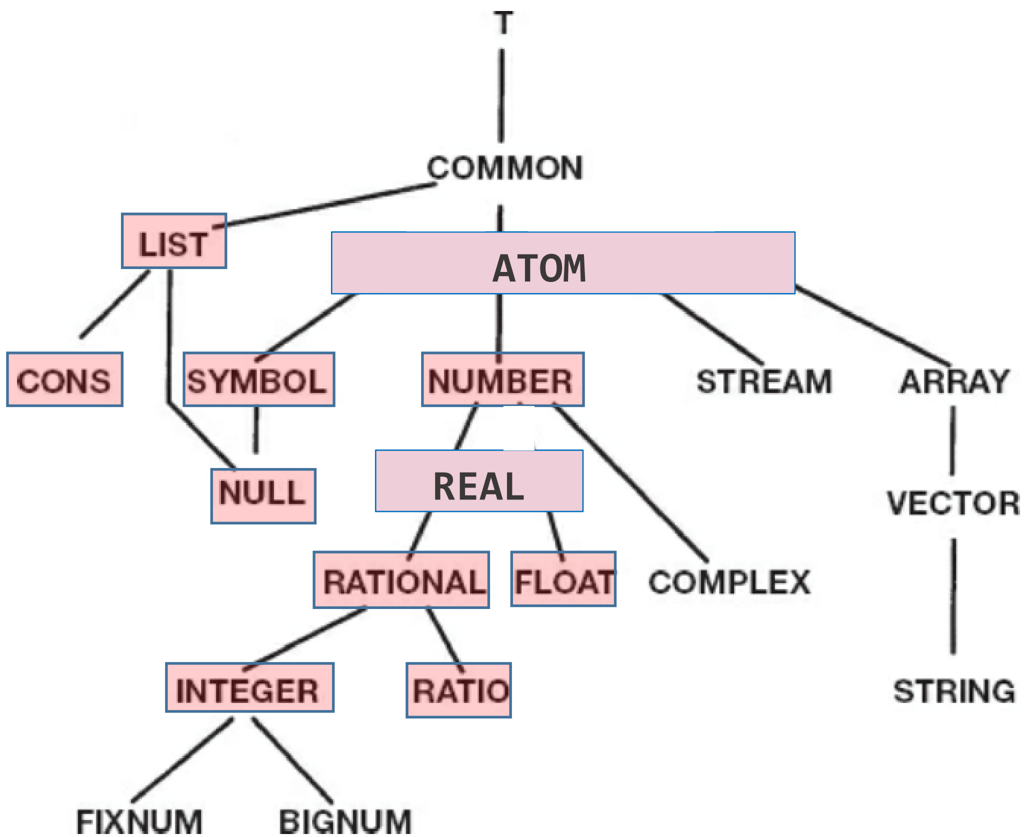
From p. 367 of Touretzky (with ATOM and REAL types added)

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.

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.

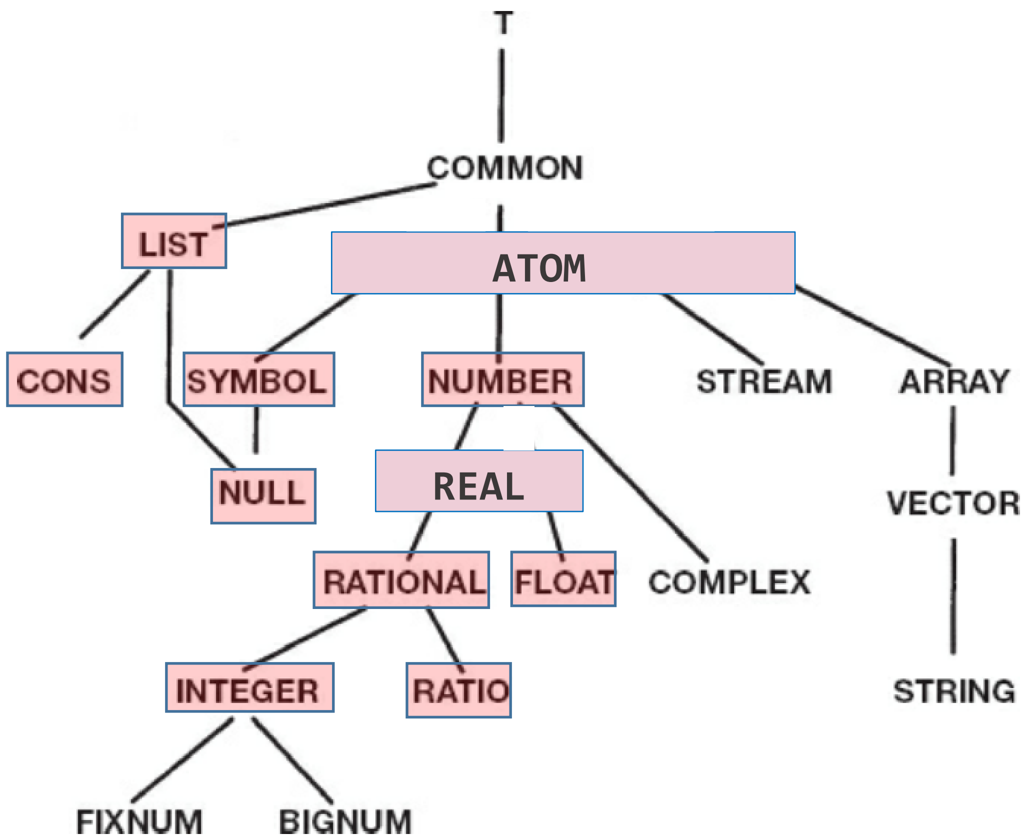


From p. 367 of Touretzky (with ATOM and REAL types added)

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.



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

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

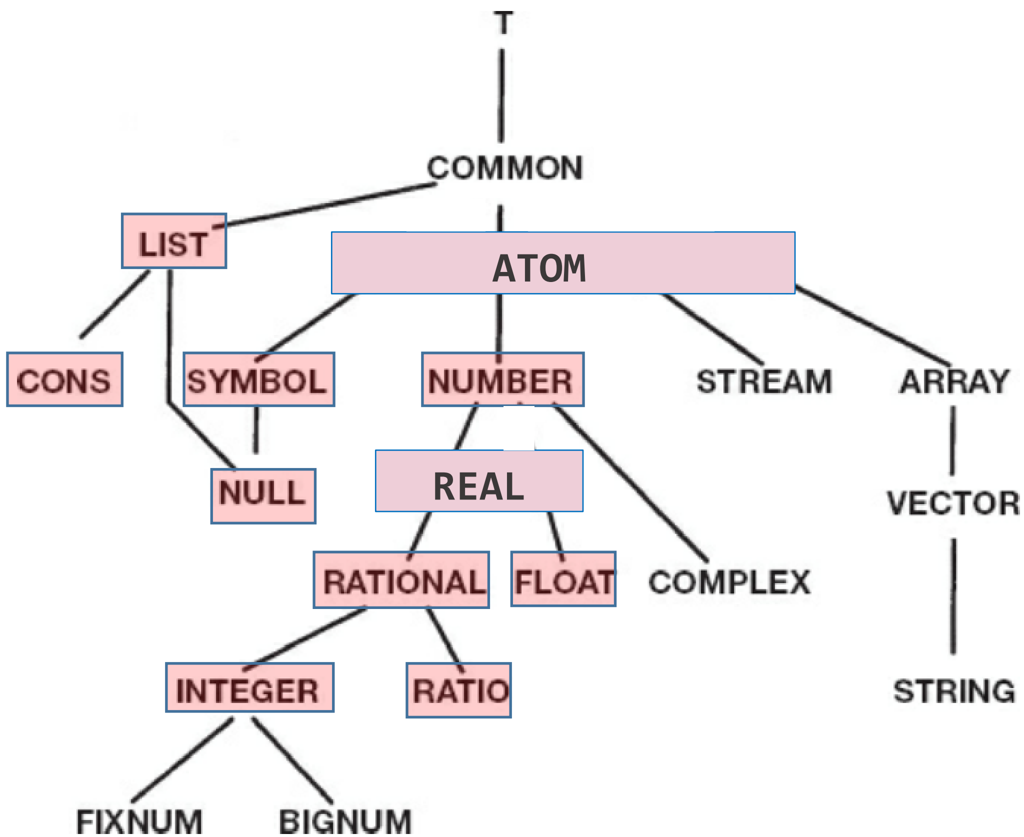
(typep *x* '<type>) ⇒ T if *x* ⇒ a value of type <type>.
(typep *x* '<type>) ⇒ NIL if *x* ⇒ a value whose type is not <type>.

<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 ***<u>except</u>*** **ATOM, NULL,** and **RATIO**),

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

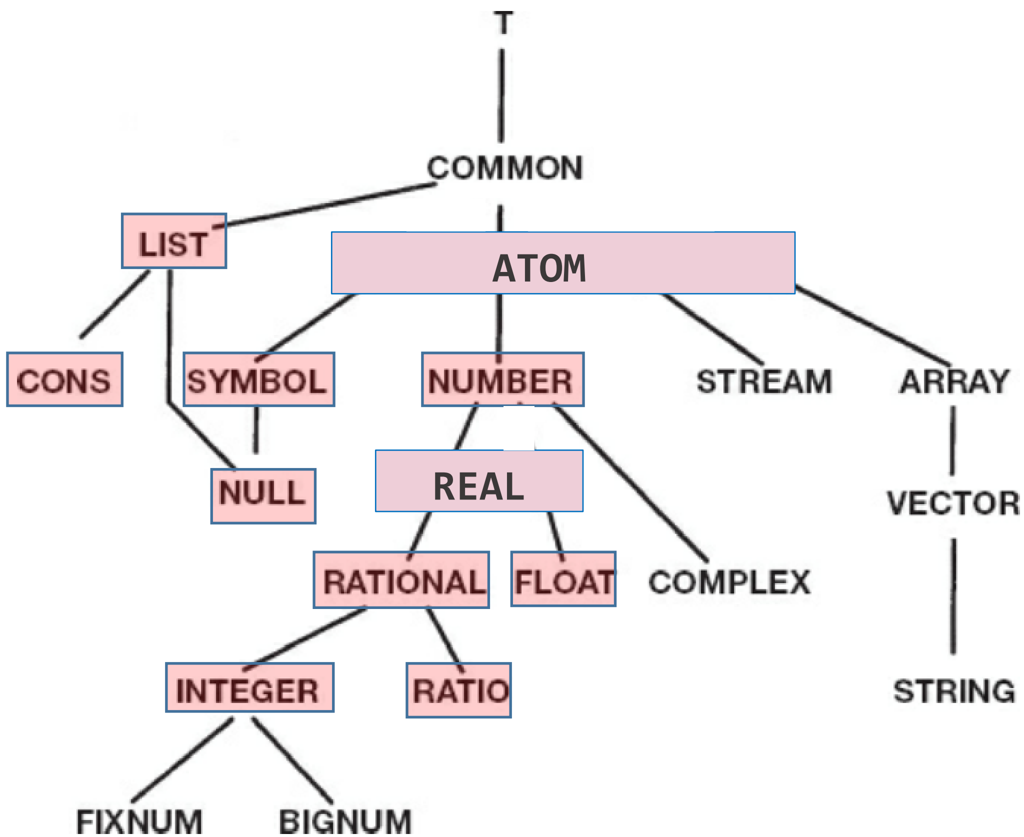
Example:



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>).

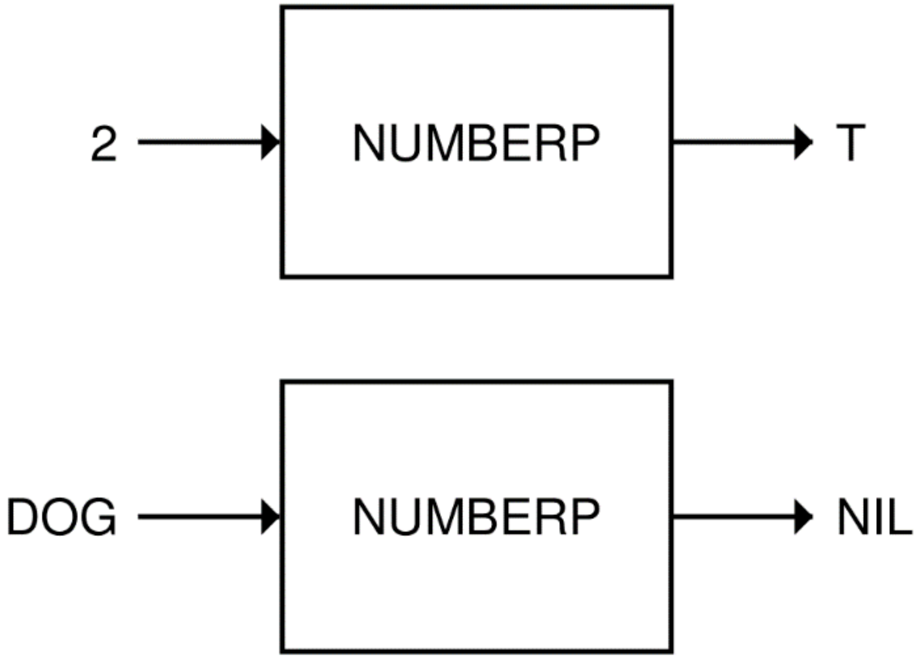
Example:
(**integerp** ***x***)
= (**typep x 'integer**)

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

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

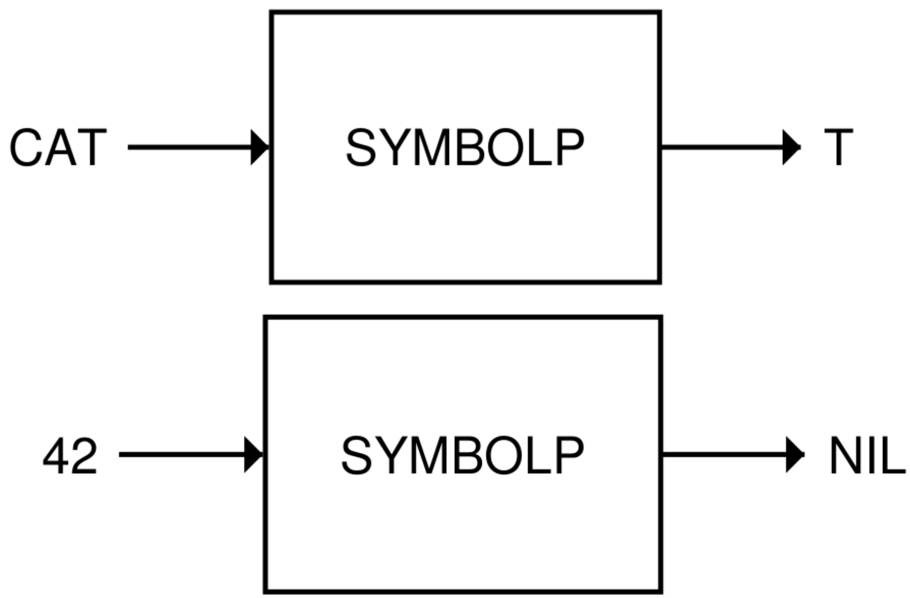
From p. 8 of Touretzky:

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:



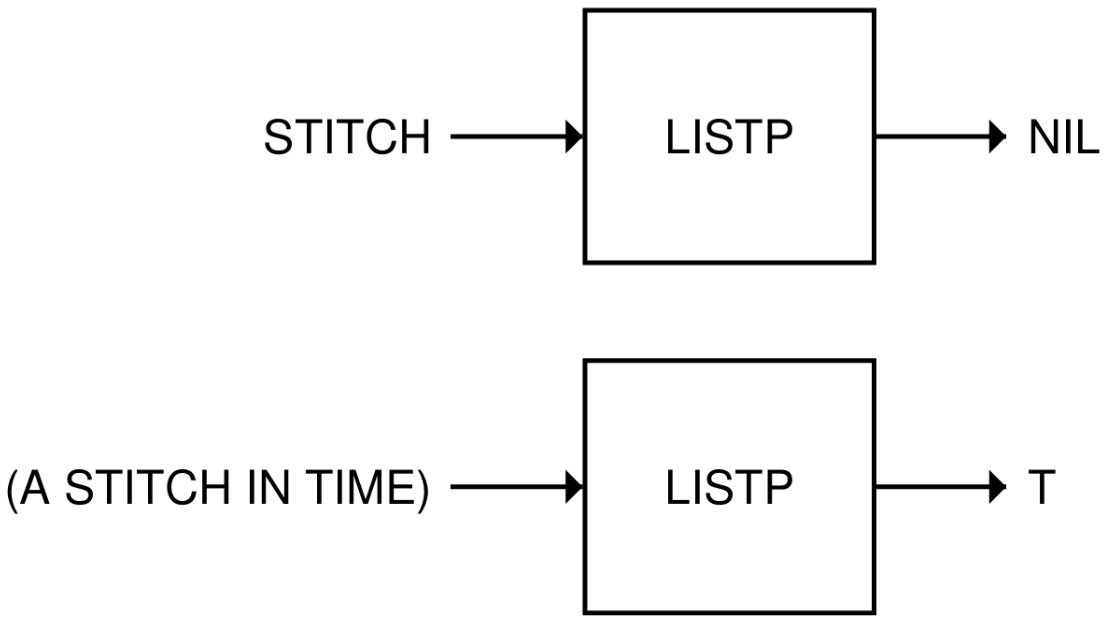
From pp. 8 – 9 of Touretzky:

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.

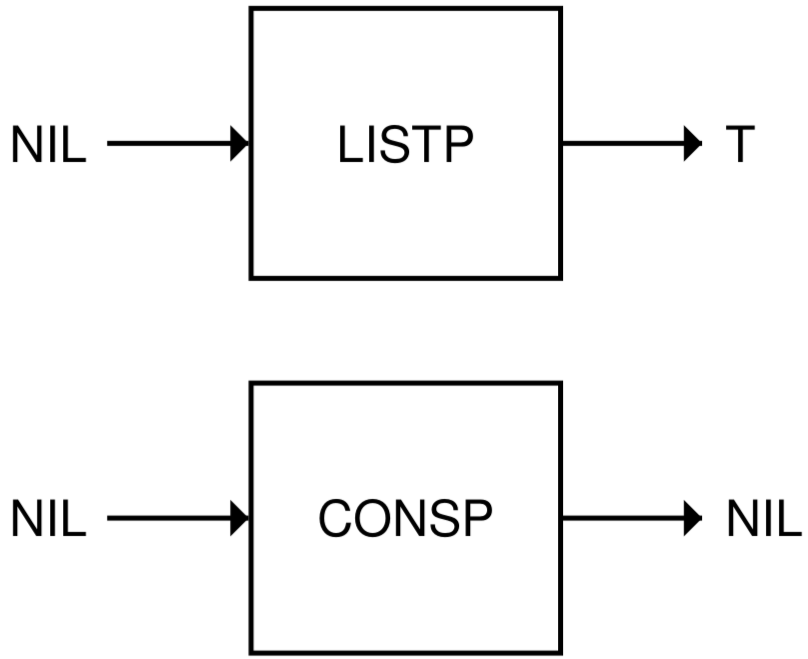


From pp. 66 – 7 of Touretzky:

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.

From pp. 66 - 7 of Touretzky:

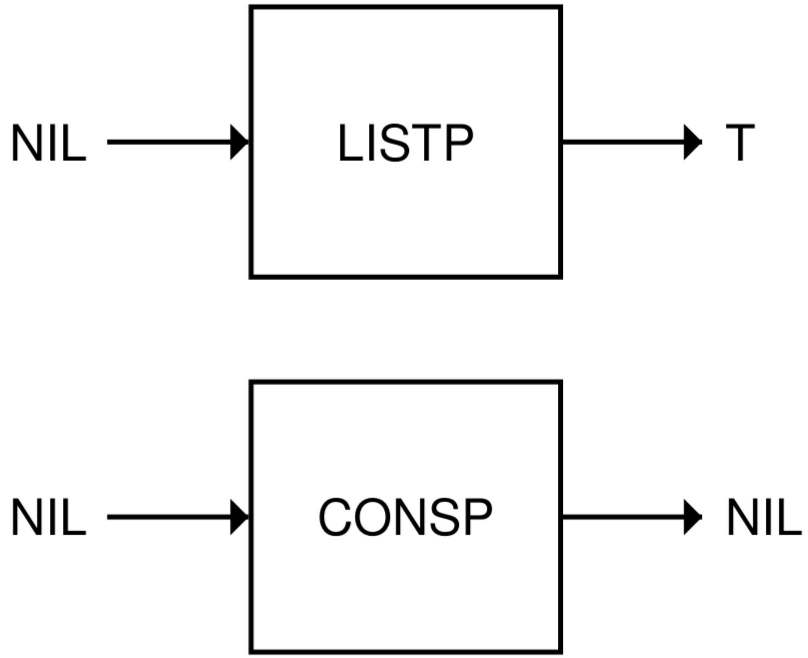
The CONSP predicate returns T if its input is a cons cell. CONSP is almost the same as LISTP; the difference is in their treatment of NIL. NIL is a list, but it is not a cons cell.



-

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.



- (consp *x*) = (typep *x* 'cons) ⇒ T if *x* ⇒ a ***nonempty*** list.
  (consp *x*) = (typep *x* 'cons) ⇒ NIL if *x* ⇒ an atom.

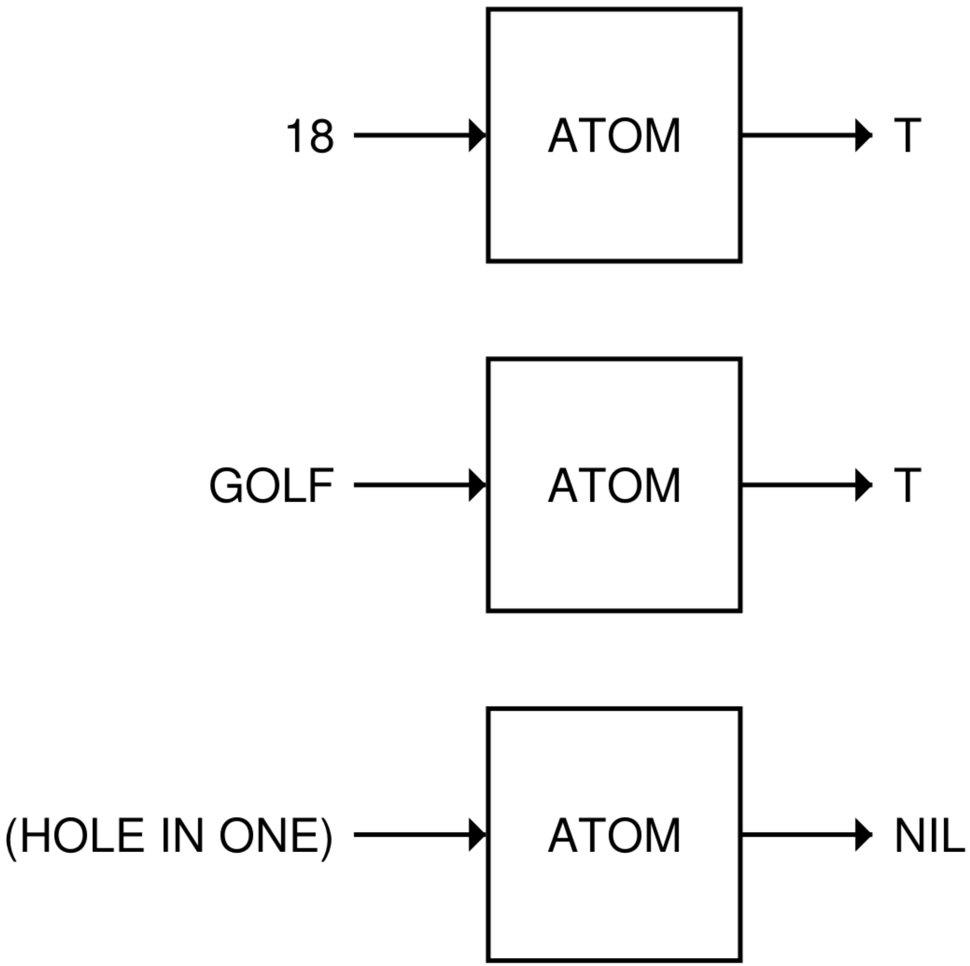
From p. 67 of Touretzky:

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.

From p. 67 of Touretzky:

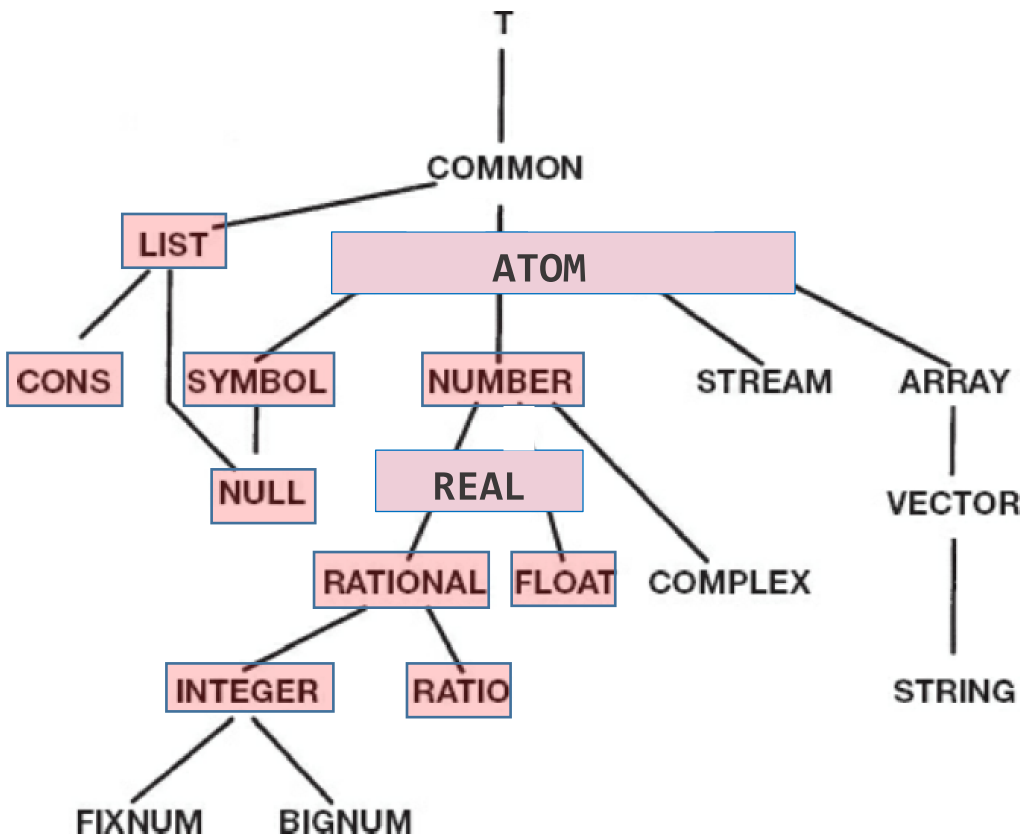
The ATOM predicate returns T if its input is anything other than a cons cell. ATOM and CONSP are opposites; when one returns T, the other always returns NIL.

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



(typep *x* '<type>) ⇒ 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.

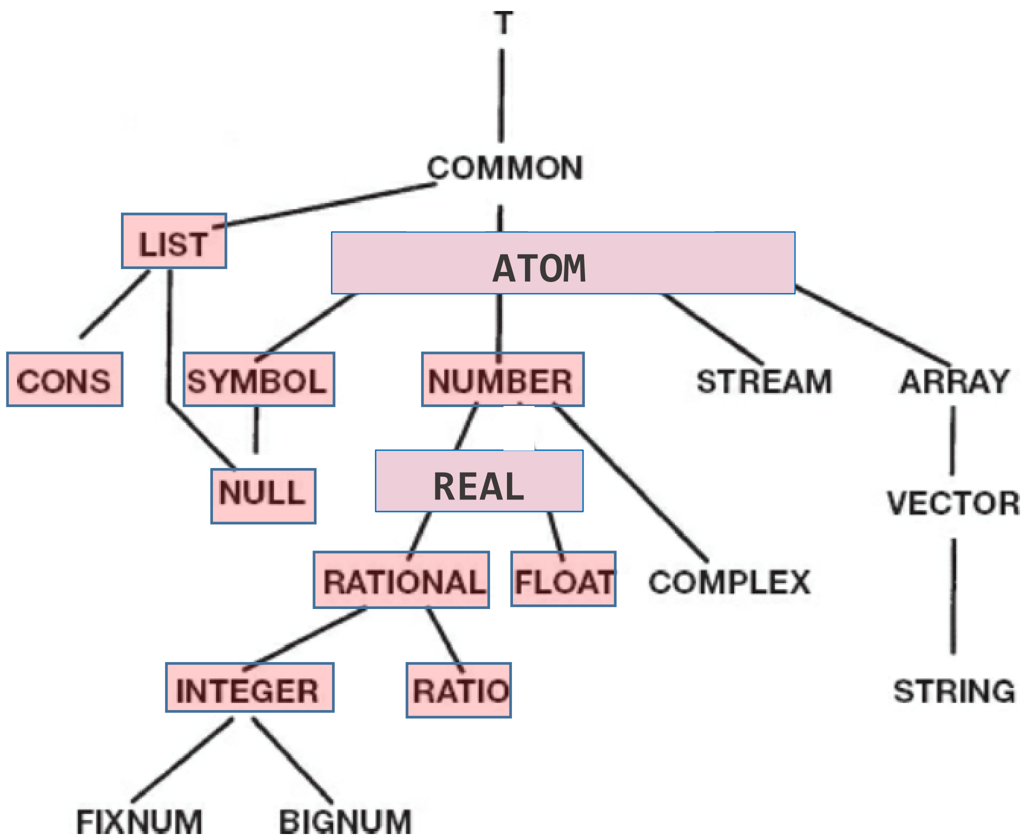
(typep *x* '<type>) ⇒ T if *x* ⇒ a value of type <type>.
(typep *x* '<type>) ⇒ NIL if *x* ⇒ a value whose type is not <type>.

<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.

2.

3.

`(>` $x_1 \ldots x_n$`)`, `(>=` $x_1 \ldots x_n$`)`, `(<` $x_1 \ldots x_n$`)`, and `(<=` $x_1 \ldots x_n$`)`

1. If any argument value is not a real number, then evaluation of `(>` $x_1 \ldots x_n$`)` produces an error!
2.
3.

`(>` $x_1 \ldots x_n$`)`, `(>=` $x_1 \ldots x_n$`)`, `(<` $x_1 \ldots x_n$`)`, and `(<=` $x_1 \ldots x_n$`)`

1. If any argument value is not a real number, then evaluation of `(>` $x_1 \ldots x_n$`)` produces an error!
2. If the argument values are all rational or all floating point, then:
   - `(>` $x_1 \ldots x_n$`)` $\Rightarrow$ `T` **if**
   - `(>` $x_1 \ldots x_n$`)` $\Rightarrow$ `NIL` otherwise.
3.

(> 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.

(> 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.,**

(> 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 ≤ 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.

(> 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 (> …):

-
-

$(> x_1 \ldots x_n)$, $(>= x_1 \ldots x_n)$, $(< x_1 \ldots x_n)$, and $(<= x_1 \ldots x_n)$

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

(> 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 ≤ 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 <=.

(> 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**!

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);
-
-
-
-
-
-

Other Useful Numerical Predicates:

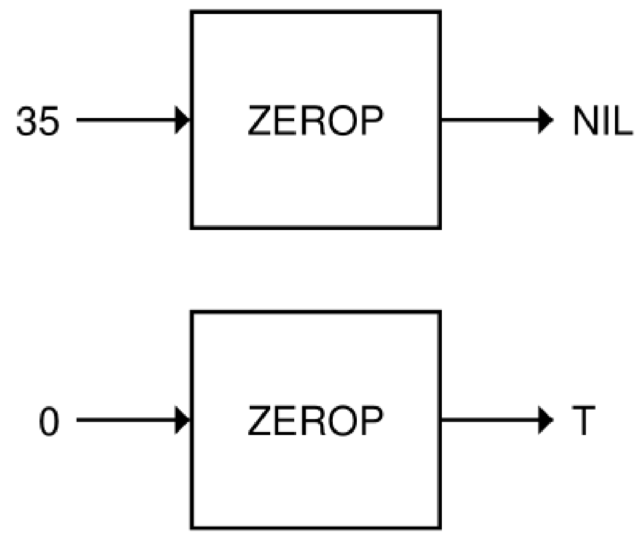
zerop, evenp, oddp, plusp, minusp, /=

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);

From p. 9 of Touretzky:

ZEROP returns T if its input is zero.



Other Useful Numerical Predicates:

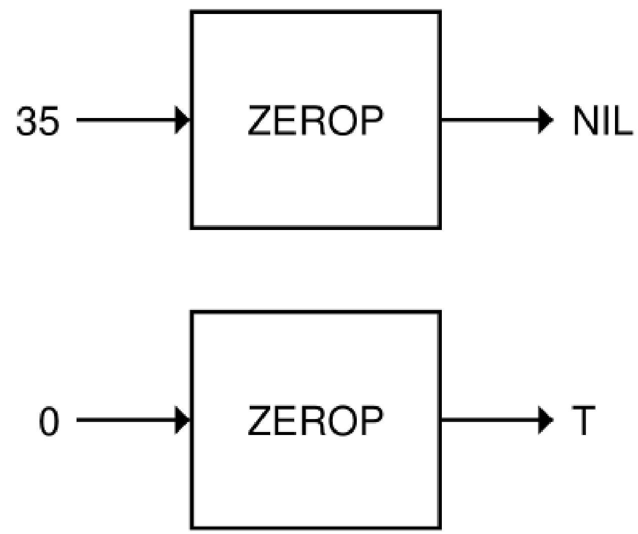
`zerop`, `evenp`, `oddp`, `plusp`, `minusp`, `/=`

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

- (zerop *x*) = (= *x* 0);
  note that (zerop 0.0) ⇒ 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**!

- `(zerop x) = (= x 0)`;
  note that `(zerop 0.0)` ⇒ `T`!
-
-
-
-
-
-

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.
-
-
-
-
-

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.
-
-
-
-

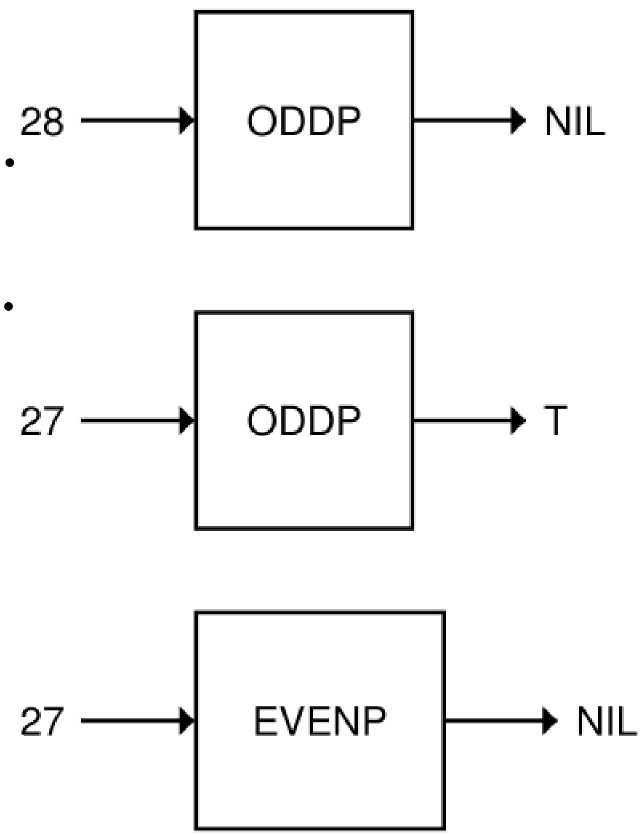
Other Useful Numerical Predicates:

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.
-
-
-
-

From Touretzky, p. 9:



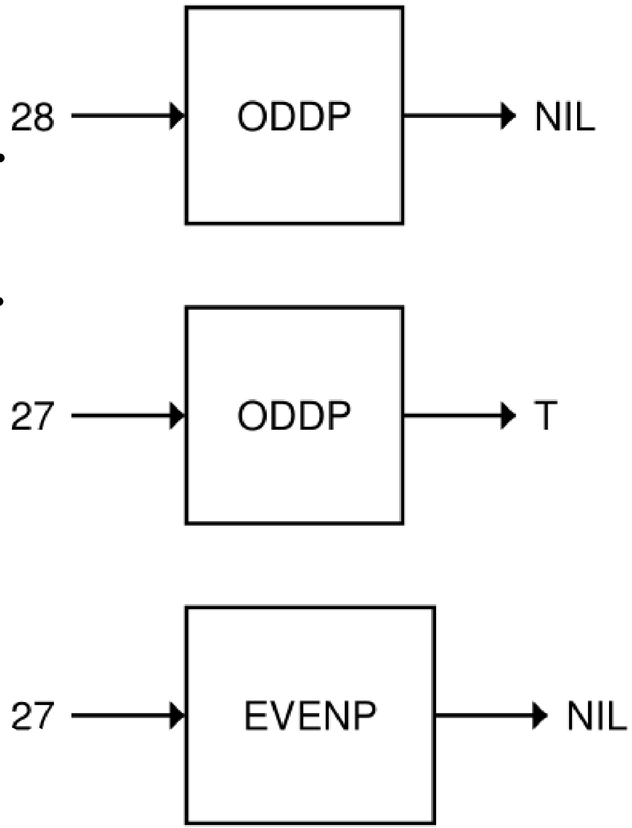
Other Useful Numerical Predicates:

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**!

- (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!
-
-
-

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**!

- (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!
-
-
-

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).
-
-

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).
-

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:

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 (/= $z_1 \ldots z_n$) ⇒ 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 (member x L) is given by these rules:

-

-

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: 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: 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.

-

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: 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 ***1***st element that is **EQL** to the value of *x*

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

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
  ***1**^st^* 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)) ⇒
  (member 9 '(2 A (9) 9 A B)) ⇒
  (member 'B '(2 A (9) 9 A B)) ⇒

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 ***1***st 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)) ⇒
  (member 'B '(2 A (9) 9 A B)) ⇒

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)) ⇒

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.

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.