C … R Functions

Each C … R function is equivalent to the **composition of a certain sequence of CARs and/or CDRs**:

- (CADR *l*) = (CAR (CDR *l*)) = (SECOND *l*)
- (CADDR *l*) = (CAR (CDR (CDR *l*))) = (THIRD *l*)
- (CADDDR *l*) = (CAR (CDR (CDR (CDR *l*)))) = (FOURTH *l*)
- (CAAR *l*) = (CAR (CAR *l*)) = (FIRST (FIRST *l*))
- (CDAR *l*) = (CDR (CAR *l*)) = (REST (FIRST *l*))
- (CDADDR *l*) = (CDR (CAR (CDR (CDR *l*))))
  = (CDR (CADDR *l*)) = (REST (THIRD *l*))
- (CADADR *l*) = (CAR (CDR (CAR (CDR *l*))))
  = (CADR (CADR *l*)) = (SECOND (SECOND *l*))

and similarly for all other sequences of **2 – 4** **A**s and/or **D**s.

- Although authors sometimes write C … R function names that contain ***more than 4*** **A**s and/or **D**s, functions with such names are ***not*** built-in functions of most implementations of Common Lisp!
- C … R function names are pronounceable: This is one reason the nondescriptive names CAR and CDR are still used.

From p. 48
of Touretzky:

CAR/CDR Pronunciation Guide

Function	Pronunciation	Alternate Name
CAR	*kar*	FIRST
CDR	*cou-der*	REST
CAAR	*ka-ar*	
CADR	*kae-der*	SECOND
CDAR	*cou-dar*	
CDDR	*cou-dih-der*	
CAAAR	*ka-a-ar*	
CAADR	*ka-ae-der*	
CADAR	*ka-dar*	
CADDR	*ka-dih-der*	THIRD
CDAAR	*cou-da-ar*	
CDADR	*cou-dae-der*	
CDDAR	*cou-dih-dar*	
CDDDR	*cou-did-dih-der*	
CADDDR	*ka-dih-dih-der*	FOURTH

and so on

- In Lisp, a ***predicate*** is a function whose calls return values that represent ***true*** or ***false***.
- In 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 NIL, the value is 2.

  - The fact that () represents false will actually be important when we consider the predicate MEMBER!
- Recall that T and NIL are constant symbols that evaluate to themselves: So T and NIL never have to be quoted, just as numbers never have to be quoted!

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

- (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.**

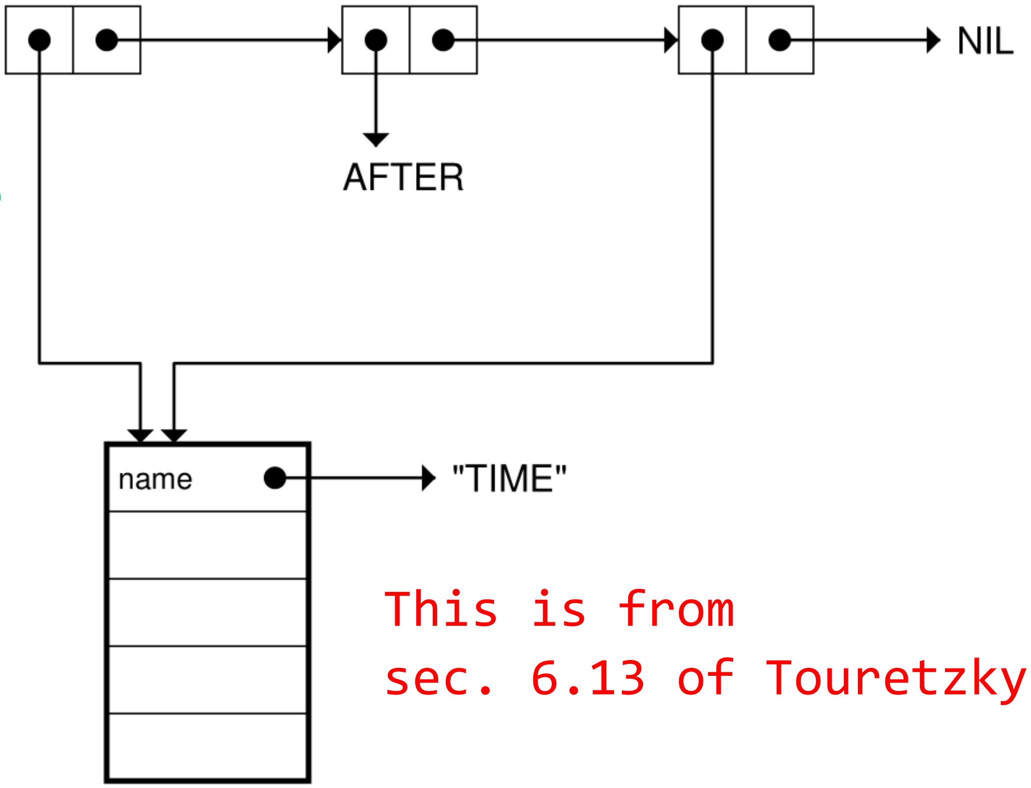
Examples
- (eq (cons 2 '(a)) (cons 2 '(a))) ⇒ NIL
- (eq (first '(a b c)) (fourth '(d c b a))) ⇒ T
  because ***symbols are memory unique***!

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

  (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. 197 of Touretzky:

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

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

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

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

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.

- **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:

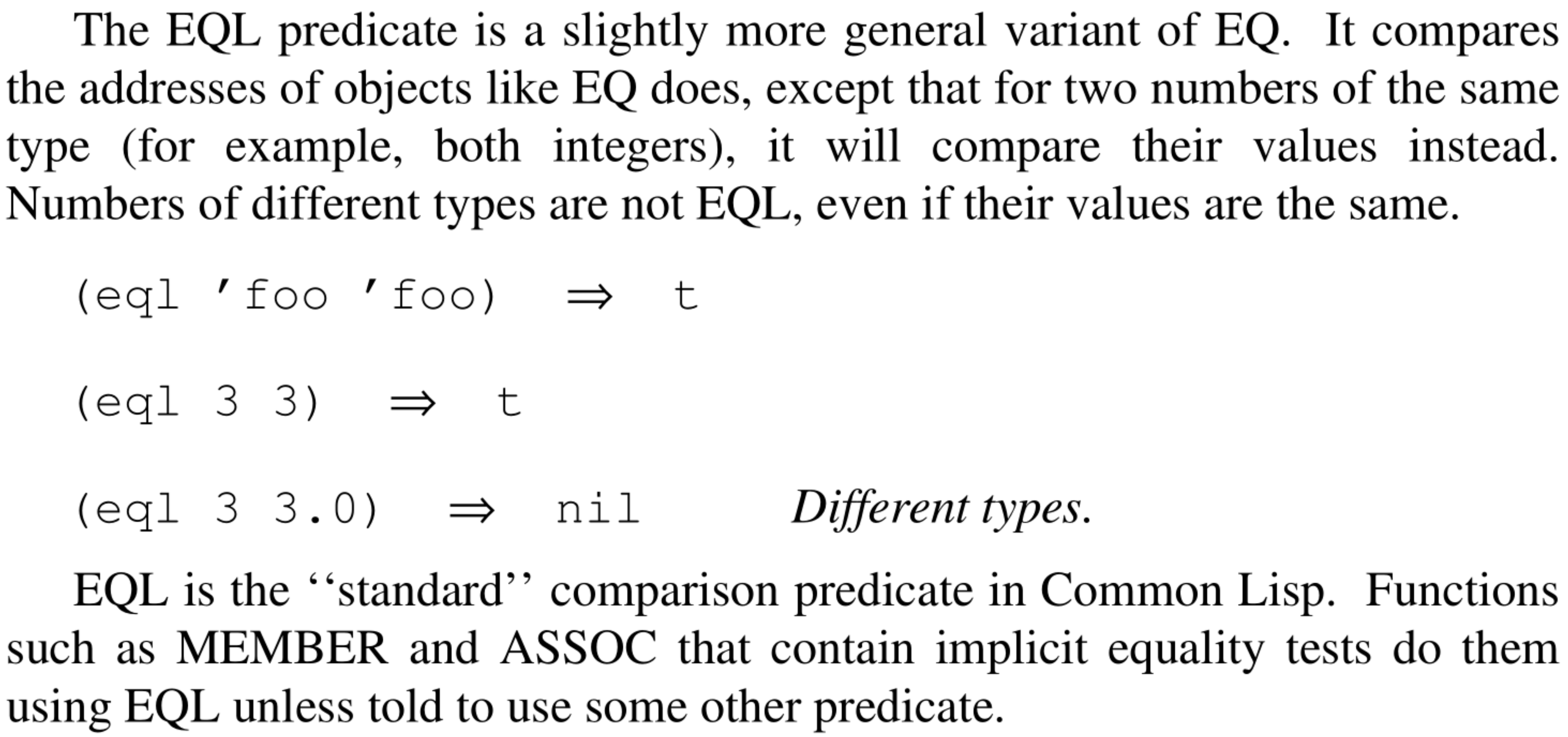
The EQL predicate is a slightly more general variant of EQ. It compares the addresses of objects like EQ does, except that for two numbers of the same type (for example, both integers), it will compare their values instead. Numbers of different types are not EQL, even if their values are the same.

```
(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***!
- 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: 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.

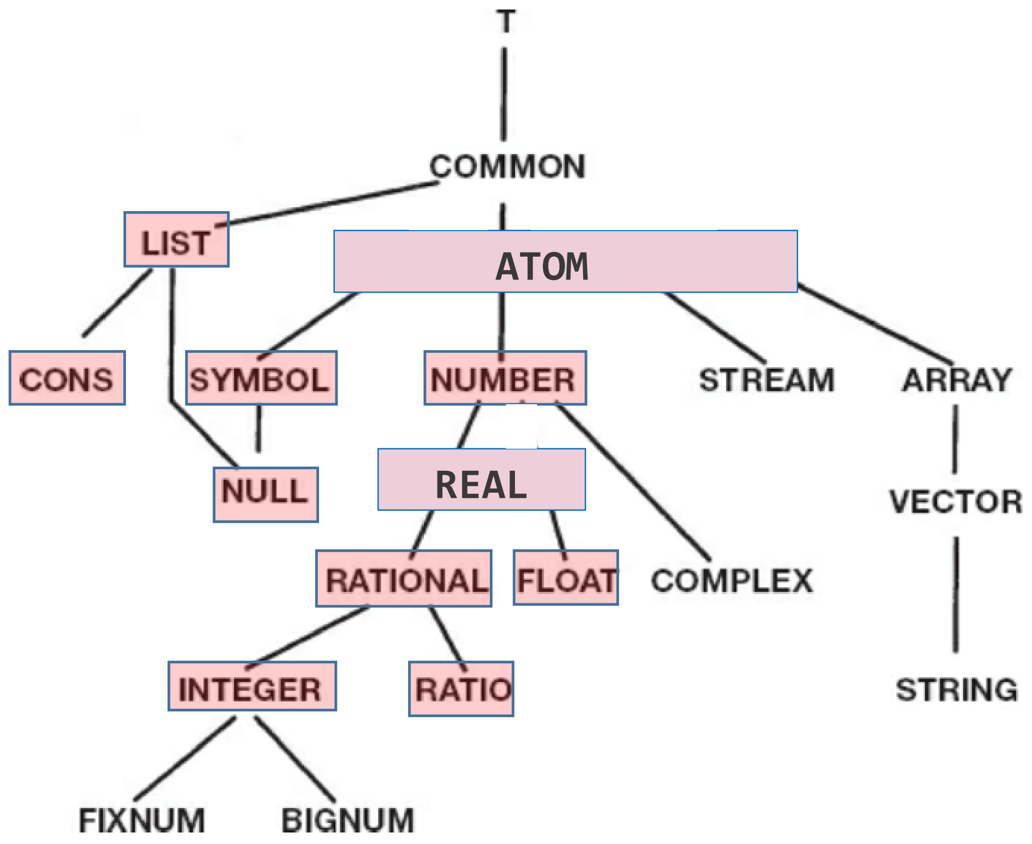
- (not *x*) = (null *x*) = (eq *x* nil)
  NOT and NULL are equivalent but are used differently:
  - (not *x*) is used to negate a boolean expression *x*, as in
    `(if (not (equal x 'dog)) ...`
  - (null *x*) is used to test if *x* ⇒ the empty list.
    - (null 17) ⇒ NIL
    - (null (cdr '(17))) ⇒ T
    - (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.

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

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

Example:

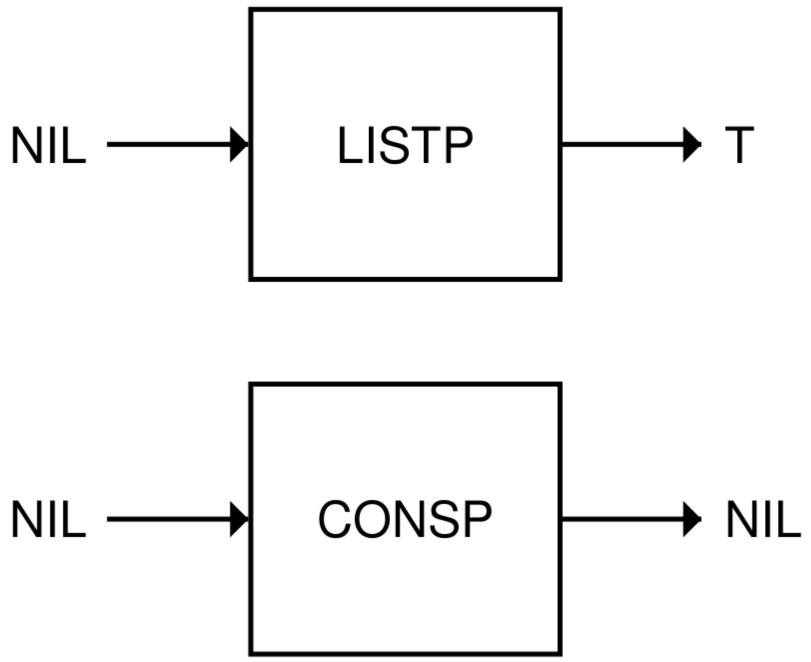
```
    (integerp x)
 =  (typep x 'integer)
```

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

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

From 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.

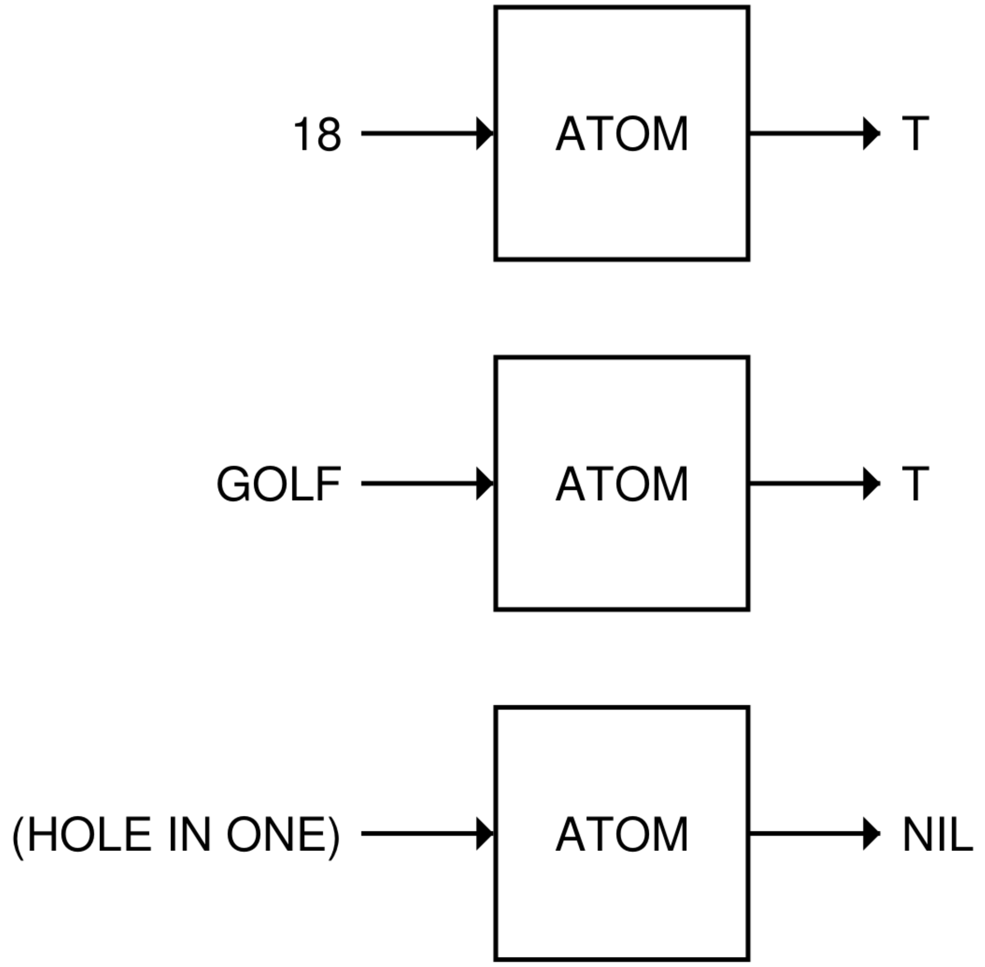


- `(consp` *x*`) = (typep` *x* `'cons) ⇒ T if` *x* `⇒ a` ***nonempty*** `list.`
  `(consp` *x*`) = (typep` *x* `'cons) ⇒ NIL if` *x* `⇒ an atom.`

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.

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


18
ATOM
T
GOLF
ATOM
T
(HOLE IN ONE)
ATOM
NIL