From p. 171 of Touretzky:

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

Here's an example of why it is useful for MEMBER to return a sublist.

From p. 171 of Touretzky:

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

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

or (eql x y) and x is in L

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

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

From p. 171 of Touretzky:

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

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

or (eql x y) and x is in L

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

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

```
> (beforep 'not 'whom
           '(ask not for whom the bell tolls))
```

From p. 171 of Touretzky:

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

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

or (eql x y) and x is in L

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

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

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

From p. 171 of Touretzky:

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

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

or (eql x y) and x is in L

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

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

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

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

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

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

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

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

Yes!

Suppose *L* ⇒ a proper list.

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

-
-

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

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

Yes!

Suppose *L* ⇒ a proper list.

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

- If *no* element of the list is **EQUAL** to *x*'s value, then

  (member *x* *L* :test #'equal) ⇒ NIL

  Example:

-

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

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

Yes!

Suppose *L* ⇒ a proper list.

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

- If *no* element of the list is **EQUAL** to *x*'s value, then

  (member *x* *L* :test #'equal) ⇒ NIL

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

-

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

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

Yes!

Suppose *L* ⇒ a proper list.

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

- If *no* element of the list is **EQUAL** to *x*'s value, then

  (member *x* *L* :test #'equal) ⇒ NIL

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

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

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

  Example:

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

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

Yes!

Suppose *L* ⇒ a proper list.

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

- If *no* element of the list is **EQUAL** to *x*'s value, then

  (member *x* *L* :test #'equal) ⇒ NIL

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

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

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

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

From pp. 199 – 200 of Touretzky:

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

From pp. 199 – 200 of Touretzky:

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

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

(member '(5 diamonds) cards)  ⇒  nil
```

From pp. 199 – 200 of Touretzky:

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

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

(member '(5 diamonds) cards)  ⇒  nil

(second cards)  ⇒  (5    diamonds)

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

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

From pp. 199 – 200 of Touretzky:

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

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

(member '(5 diamonds) cards)  ⇒  nil

(second cards)  ⇒  (5    diamonds)

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

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

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

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

IF, COND, AND, and OR

Review of the IF Special Operator

From p. 114 of Touretzky:

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

Review of the IF Special Operator

From p. 114 of Touretzky:

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

```
(if (oddp 1) 'odd 'even)  ⇒

(if (oddp 2) 'odd 'even)  ⇒

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


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


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

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

Review of the IF Special Operator

From p. 114 of Touretzky:

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

```
(if (oddp 1) 'odd 'even)  ⇒  odd

(if (oddp 2) 'odd 'even)  ⇒

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

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

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

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

Review of the IF Special Operator

From p. 114 of Touretzky:

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

```
(if (oddp 1) 'odd 'even)  ⇒  odd

(if (oddp 2) 'odd 'even)  ⇒  even

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


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


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

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

Review of the IF Special Operator

From p. 114 of Touretzky:

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

```
(if (oddp 1) 'odd 'even)  ⇒  odd

(if (oddp 2) 'odd 'even)  ⇒  even

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

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


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

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

Review of the IF Special Operator

From p. 114 of Touretzky:

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

```
(if (oddp 1) 'odd 'even)  ⇒  odd

(if (oddp 2) 'odd 'even)  ⇒  even

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

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

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

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

Review of the IF Special Operator

From p. 114 of Touretzky:

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

```
(if (oddp 1) 'odd 'even)  ⇒  odd

(if (oddp 2) 'odd 'even)  ⇒  even

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

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

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

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

Review of the IF Special Operator

From p. 114 of Touretzky:

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

```
(if (oddp 1) 'odd 'even)  ⇒  odd

(if (oddp 2) 'odd 'even)  ⇒  even

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

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

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

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

From p. 114 of Touretzky:

Let’s use IF to construct a function that takes the absolute value of a number. Absolute values are always nonnegative. For negative numbers the absolute value is the negation of the number; for positive numbers and zero the absolute value is the number itself.

From p. 114 of Touretzky:

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

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

From p. 114 of Touretzky:

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

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

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

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

•

From p. 114 of Touretzky:

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

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

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

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

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

From p. 114 of Touretzky:

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

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

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

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

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

From p. 115 of Touretzky:

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

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

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

From p. 115 of Touretzky:

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

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

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

```
> (symbol-test 'rutabaga)
```
Evaluate true-part.

From p. 115 of Touretzky:

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

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

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

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

From p. 115 of Touretzky:

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

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

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

```
> (symbol-test 'rutabaga)
(YES RUTABAGA IS A SYMBOL)
```

Evaluate true-part.

```
> (symbol-test 12345)
```

Evaluate false-part.

From p. 115 of Touretzky:

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

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

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

```
> (symbol-test 'rutabaga)
(YES RUTABAGA IS A SYMBOL)
```

Evaluate true-part.

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

Evaluate false-part.

From p. 115 of Touretzky:

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

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

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

```
> (symbol-test 'rutabaga)
(YES RUTABAGA IS A SYMBOL)
```

Evaluate true-part.

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

Evaluate false-part.

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

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

From p. 115 of Touretzky:

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

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

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

```
> (symbol-test 'rutabaga)
(YES RUTABAGA IS A SYMBOL)
```

Evaluate true-part.

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

Evaluate false-part.

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

```
(if t 'happy)  ⇒  happy
```

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

From p. 115 of Touretzky:

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

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

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

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

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

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

```
(if t 'happy)  ⇒  happy

(if nil 'happy)  ⇒  nil
```

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

An Example of Nested IFs

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

-

-

-

-

Examples:

An Example of Nested IFs

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

- If $e \Rightarrow$ 0, then
  (kind-of-value e) $\Rightarrow$ (INTEGER-0 0)
-
-
-

Examples:

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

An Example of Nested IFs

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

- If *e* ⇒ 0, then
  (kind-of-value *e*) ⇒ (INTEGER-0 0)
- If *e* ⇒ any other number, then
  (kind-of-value *e*) ⇒ (NUM-BUT-NOT-0 <*e*'s value>)
-
-

Examples:

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

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

An Example of Nested IFs

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

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

Examples:

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

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

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

An Example of Nested IFs

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

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

Examples:

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

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

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

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

An Example of Nested IFs

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

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

An Example of Nested IFs

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

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

```
(defun kind-of-value (e)
   (if (eql 0 e)

       (if (numberp e)

           (if (atom e)

                                        ))))
```

An Example of Nested IFs

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

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

```
(defun kind-of-value (e)
   (if (eql 0 e)
       '(integer-0 0)
       (if (numberp e)

           (if (atom e)

                                             ))))
```

An Example of Nested IFs

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

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

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

                                        ))))
```

An Example of Nested IFs

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

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

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

An Example of Nested IFs

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

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

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

The COND Macro Operator

The macro form $(\texttt{cond}\ (g_1\ e_1)$

$$(g_2\ e_2)$$

$$\vdots$$

$$(g_n\ e_n))$$

is equivalent to n nested ifs:

The COND Macro Operator

The macro form

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

is equivalent to n nested ifs:

$$
\begin{array}{l}
\texttt{(if } g_1 \\
\quad\ \ e_1 \\
\quad\ \ \texttt{(if } g_2 \\
\qquad\quad e_2 \\
\qquad\qquad \ddots \\
\qquad\qquad\quad \texttt{(if } g_n \\
\qquad\qquad\qquad\ \ e_n\texttt{)} \cdots \texttt{))}
\end{array}
$$

-
-
-

The COND Macro Operator

The macro form
```
(cond (g1 e1)
      (g2 e2)
        ⋮
      (gn en))
```

$$(\mathtt{cond}\ (g_1\ e_1)\ (g_2\ e_2)\ \vdots\ (g_n\ e_n))$$

is equivalent to n nested ifs:

$$(\mathtt{if}\ g_1\ e_1\ (\mathtt{if}\ g_2\ e_2\ \ddots\ (\mathtt{if}\ g_n\ e_n)\ \cdots\ ))$$

- Each $(g_i\ e_i)$ is called a ***clause*** of the COND.
-
-

The COND Macro Operator

The macro form

```
(cond (g1 e1)
      (g2 e2)
        ⋮
      (gn en))
```

is equivalent to n nested ifs:

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

- Each (g_i e_i) is called a ***clause*** of the COND.
- In any clause (g e):
  - g is called the .
  - e is called the .
-

The COND Macro Operator

The macro form

```
(cond (g1 e1)
      (g2 e2)
        ⋮
      (gn en))
```

$$(\texttt{cond}\ (g_1\ e_1)\ (g_2\ e_2)\ \vdots\ (g_n\ e_n))$$

is equivalent to n nested ifs:

$$(\texttt{if}\ g_1\ e_1\ (\texttt{if}\ g_2\ e_2\ \ddots\ (\texttt{if}\ g_n\ e_n) \cdots))$$

- Each $(g_i\ e_i)$ is called a ***clause*** of the COND.
- In any clause $(g\ e)$:
  - g is called the ***test***.
  - e is called the ***consequent***.
-

The COND Macro Operator

The macro form

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

is equivalent to n nested ifs:

$$
\begin{array}{l}
\texttt{(if } g_1 \\
\qquad e_1 \\
\qquad \texttt{(if } g_2 \\
\qquad\qquad e_2 \\
\qquad\qquad\quad \ddots \\
\qquad\qquad\qquad \texttt{(if } g_n \\
\qquad\qquad\qquad\qquad e_n\texttt{)} \cdots \texttt{))}
\end{array}
$$

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

From p. 116 of Touretzky:

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

From p. 116 of Touretzky:

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

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

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

-
-
-
-

From p. 116 of Touretzky:

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

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

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

- `(compare   4    8)` ⇒
- `(compare   9  7.3)` ⇒
- `(compare 7.3  7.3)` ⇒
- `(compare   7  7.0)` ⇒

From p. 116 of Touretzky:

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

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

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

- `(compare 4 8)` ⇒ `FIRST-IS-SMALLER`
- `(compare 9 7.3)` ⇒
- `(compare 7.3 7.3)` ⇒
- `(compare 7 7.0)` ⇒

From p. 116 of Touretzky:

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

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

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

- (compare 4 8) $\Rightarrow$ FIRST-IS-SMALLER
- (compare 9 7.3) $\Rightarrow$ FIRST-IS-BIGGER
- (compare 7.3 7.3) $\Rightarrow$
- (compare 7 7.0) $\Rightarrow$

From p. 116 of Touretzky:

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

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

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

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

From p. 116 of Touretzky:

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

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

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

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

Often, the last test is t.

Then, since (if t e) is equivalent to e,

$$
\begin{array}{ll}
\texttt{(cond} & (g_1\ e_1) \\
 & (g_2\ e_2) \\
 & \quad\vdots \\
 & (g_{n\text{-}1}\ e_{n\text{-}1}) \\
 & (\texttt{t}\quad e_n))
\end{array}
$$

is equivalent to

Often, the last test is t.

Then, since (if t *e*) is equivalent to *e*,

$$
\begin{array}{l}
(\texttt{cond}\ (g_1\ e_1) \\
\qquad\quad (g_2\ e_2) \\
\qquad\qquad \vdots \\
\qquad\quad (g_{n\text{-}1}\ e_{n\text{-}1}) \\
\qquad\quad (\texttt{t}\quad e_n))
\end{array}
$$

is equivalent to

$$
\begin{array}{l}
(\texttt{if}\ g_1 \\
\qquad e_1 \\
\qquad (\texttt{if}\ g_2 \\
\qquad\qquad e_2 \\
\qquad\qquad\quad \ddots \\
\qquad\qquad\qquad (\texttt{if}\ g_{n\text{-}1} \\
\qquad\qquad\qquad\qquad e_{n\text{-}1} \\
\qquad\qquad\qquad\qquad e_n)\ \cdots\ ))
\end{array}
$$

-

Often, the last test is t.

Then, since (if t e) is equivalent to e,

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

$$
\begin{array}{l}
\texttt{(cond } (g_1\ e_1) \\
\qquad\quad\ (g_2\ e_2) \\
\qquad\qquad\ \vdots \\
\qquad\quad\ (g_{n\text{-}1}\ e_{n\text{-}1}) \\
\qquad\quad\ (\texttt{t}\quad e_n))
\end{array}
$$

is equivalent to

$$
\begin{array}{l}
\texttt{(if } g_1 \\
\qquad e_1 \\
\qquad \texttt{(if } g_2 \\
\qquad\qquad e_2 \\
\qquad\qquad\quad \ddots \\
\qquad\qquad\qquad \texttt{(if } g_{n\text{-}1} \\
\qquad\qquad\qquad\qquad e_{n\text{-}1} \\
\qquad\qquad\qquad\qquad e_n) \cdots))
\end{array}
$$

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

The above function definition

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

can be rewritten using COND as follows:

The above function definition

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

can be rewritten using COND as follows:

```
(defun kind-of-value (e)
   (cond (                           )
         (                                   )
         (                                   )
         (                         )))
```

The above function definition

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

can be rewritten using COND as follows:

```
(defun kind-of-value (e)
   (cond ((eql 0 e) '(integer-0 0))
         (                                    )
         (                                    )
         (                          )))
```

The above function definition

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

can be rewritten using COND as follows:

```
(defun kind-of-value (e)
   (cond ((eql 0 e) '(integer-0 0))
         ((numberp e) (list 'num-but-not-0 e))
         (                                   )
         (                         )))
```

The above function definition

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

can be rewritten using COND as follows:

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

The above function definition

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

can be rewritten using COND as follows:

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

Another example, from p. 117 of Touretzky:

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

From pp. 119 – 20 of Touretzky:

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

• • •

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

From p. 120 of Touretzky:

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

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

From p. 120 of Touretzky:

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

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

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

From p. 120 of Touretzky:

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

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

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

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

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

From p. 120 of Touretzky:

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

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

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

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

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

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

The OR Macro Operator

(OR e_1 … e_n) is analogous to e_1 || … || e_n in C++ or Java, except that when (OR e_1 … e_n)'s value is *true* its value is ***the value of the first e whose value isn't NIL***.

The OR Macro Operator

(OR e_1 … e_n) is analogous to e_1 || … || e_n in C++ or Java, except that when (OR e_1 … e_n)'s value is *true* its value is ***the value of the first e whose value isn't NIL***.

From p. 122 of Touretzky:

suppose we want a function GTEST that takes two numbers as input and returns T if either the first is greater than the second or one of them is zero. These conditions form a disjunctive set; only one need be true for GTEST to return T. OR is used for disjunctions.

```
(defun gtest (x y)
  (or (> x y)
      (zerop x)
      (zerop y)))
```

The OR Macro Operator

(OR e_1 … e_n) is analogous to e_1 || … || e_n in C++ or Java, except that when (OR e_1 … e_n)'s value is *true* its value is ***the value of the first e whose value isn't NIL***.

Another Example

The OR Macro Operator

(OR e_1 … e_n) is analogous to e_1 || … || e_n in C++ or Java, except that when (OR e_1 … e_n)'s value is *true* its value is ***the value of the first e whose value isn't NIL***.

Another Example Suppose f is defined as follows:

```
(defun f (x)  (or (member x '(A B C)) (member x '(2 3 B))))
```

Then:
- (f 'B) ⇒
- (f 2) ⇒
- (f 'A) ⇒
- (f 6) ⇒

The OR Macro Operator

(OR e_1 … e_n) is analogous to e_1 || … || e_n in C++ or Java, except that when (OR e_1 … e_n)'s value is *true* its value is ***the value of the first e whose value isn't NIL***.

Another Example Suppose f is defined as follows:

```
(defun f (x)  (or (member x '(A B C)) (member x '(2 3 B))))
```

Then:
- (f 'B) ⇒ (B C)
- (f 2) ⇒
- (f 'A) ⇒
- (f 6) ⇒

The OR Macro Operator

(OR e_1 … e_n) is analogous to e_1 || … || e_n in C++ or Java, except that when (OR e_1 … e_n)'s value is *true* its value is ***the value of the first e whose value isn't NIL***.

Another Example Suppose f is defined as follows:

```
(defun f (x)  (or (member x '(A B C)) (member x '(2 3 B))))
```

Then:
- (f 'B) ⇒ (B C)
- (f 2) ⇒
- (f 'A) ⇒ (A B C)
- (f 6) ⇒

The OR Macro Operator

(OR e_1 … e_n) is analogous to e_1 || … || e_n in C++ or Java, except that when (OR e_1 … e_n)'s value is *true* its value is ***the value of the first e whose value isn't NIL***.

Another Example Suppose f is defined as follows:

```
(defun f (x)  (or (member x '(A B C)) (member x '(2 3 B))))
```

Then:
- (f 'B) ⇒ (B C)
- (f 2) ⇒ (2 3 B)
- (f 'A) ⇒ (A B C)
- (f 6) ⇒

The OR Macro Operator

(OR e_1 … e_n) is analogous to e_1 || … || e_n in C++ or Java, except that when (OR e_1 … e_n)'s value is *true* its value is ***the value of the first e whose value isn't NIL***.

Another Example Suppose f is defined as follows:

```
(defun f (x)  (or (member x '(A B C)) (member x '(2 3 B))))
```

Then:

- (f 'B) ⇒ (B C)
- (f 'A) ⇒ (A B C)
- (f 2) ⇒ (2 3 B)
- (f 6) ⇒ NIL

The OR Macro Operator

(OR e_1 … e_n) is analogous to e_1 || … || e_n in C++ or Java, except that when (OR e_1 … e_n)'s value is *true* its value is ***the value of the first e whose value isn't NIL***.

Another Example Suppose f is defined as follows:

```
(defun f (x)  (or (member x '(A B C)) (member x '(2 3 B))))
```

Then:
- (f 'B) ⇒ (B C)
- (f 'A) ⇒ (A B C)
- (f 2) ⇒ (2 3 B)
- (f 6) ⇒ NIL

Like e_1 || … || e_n in C++ or Java, (OR e_1 … e_n) is evaluated using ***short-circuit evaluation***, as follows:

•

•

The OR Macro Operator

(OR e_1 … e_n) is analogous to e_1 || … || e_n in C++ or Java, except that when (OR e_1 … e_n)'s value is *true* its value is ***the value of the first e whose value isn't NIL***.

Another Example Suppose f is defined as follows:

```
(defun f (x)  (or (member x '(A B C)) (member x '(2 3 B))))
```

Then:
- (f 'B) ⇒ (B C)
- (f 'A) ⇒ (A B C)
- (f 2) ⇒ (2 3 B)
- (f 6) ⇒ NIL

Like e_1 || … || e_n in C++ or Java, (OR e_1 … e_n) is evaluated using ***short-circuit evaluation***, as follows:

- The expressions e_1, …, e_n are evaluated *in that order*, but evaluation of these expressions stops when an expression e_i is found to have a value that **isn't NIL**: When that happens,
-

The OR Macro Operator

(OR e_1 … e_n) is analogous to e_1 || … || e_n in C++ or Java, except that when (OR e_1 … e_n)'s value is *true* its value is ***the value of the first e whose value isn't NIL***.

Another Example Suppose f is defined as follows:

```
(defun f (x)  (or (member x '(A B C)) (member x '(2 3 B))))
```

Then:
- (f 'B) ⇒ (B C)
- (f 'A) ⇒ (A B C)
- (f 2) ⇒ (2 3 B)
- (f 6) ⇒ NIL

Like e_1 || … || e_n in C++ or Java, (OR e_1 … e_n) is evaluated using ***short-circuit evaluation***, as follows:

- The expressions e_1, …, e_n are evaluated *in that order*, but evaluation of these expressions stops when an expression e_i is found to have a value that **isn't NIL**: When that happens, the value of e_i is returned as the value of (OR e_1 … e_n), and any subsequent expressions e_{i+1}, …, e_n are ***not*** evaluated.
-

The OR Macro Operator

(OR e_1 … e_n) is analogous to e_1 || … || e_n in C++ or Java, except that when (OR e_1 … e_n)'s value is *true* its value is ***the value of the first e whose value isn't NIL***.

Another Example Suppose f is defined as follows:

```
(defun f (x)  (or (member x '(A B C)) (member x '(2 3 B))))
```

Then:
- (f 'B) ⇒ (B C)
- (f 'A) ⇒ (A B C)
- (f 2) ⇒ (2 3 B)
- (f 6) ⇒ NIL

Like e_1 || … || e_n in C++ or Java, (OR e_1 … e_n) is evaluated using ***short-circuit evaluation***, as follows:

- The expressions e_1, …, e_n are evaluated *in that order*, but evaluation of these expressions stops when an expression e_i is found to have a value that **isn't NIL**: When that happens, the value of e_i is returned as the value of (OR e_1 … e_n), and any subsequent expressions e_{i+1}, …, e_n are ***not*** evaluated.

- If e_1, …, e_n all have value NIL, then the value of (OR e_1 … e_n) is also NIL.

Like e_1 || … || e_n in C++ or Java, (OR e_1 … e_n) is evaluated using ***short-circuit evaluation***, as follows:

- The expressions e_1, …, e_n are evaluated *in that order*, but evaluation of these expressions stops when an expression e_i is found to have a value that **isn't NIL**: When that happens, the value of e_i is returned as the value of (OR e_1 … e_n), and any subsequent expressions e_{i+1}, …, e_n are ***not*** evaluated.
- If e_1, …, e_n all have value NIL, then the value of (OR e_1 … e_n) is also NIL.

Like e_1 || … || e_n in C++ or Java, (OR e_1 … e_n) is evaluated using ***short-circuit evaluation***, as follows:

- The expressions e_1, …, e_n are evaluated *in that order*, but evaluation of these expressions stops when an expression e_i is found to have a value that **isn't NIL**: When that happens, the value of e_i is returned as the value of (OR e_1 … e_n), and any subsequent expressions e_{i+1}, …, e_n are ***not*** evaluated.
- If e_1, …, e_n all have value NIL, then the value of (OR e_1 … e_n) is also NIL.

Like e_1 || … || e_n in C++ or Java, (OR e_1 … e_n) is evaluated using ***short-circuit evaluation***, as follows:

- The expressions e_1, …, e_n are evaluated *in that order*, but evaluation of these expressions stops when an expression e_i is found to have a value that **isn't NIL**: When that happens, the value of e_i is returned as the value of (OR e_1 … e_n), and any subsequent expressions e_{i+1}, …, e_n are ***not*** evaluated.
- If e_1, …, e_n all have value NIL, the value of (OR e_1 … e_n) is NIL.

Like e_1 || … || e_n in C++ or Java, (OR e_1 … e_n) is evaluated using ***short-circuit evaluation***, as follows:

- The expressions e_1, …, e_n are evaluated *in that order*, but evaluation of these expressions stops when an expression e_i is found to have a value that **isn't NIL**: When that happens, the value of e_i is returned as the value of (OR e_1 … e_n), and any subsequent expressions e_{i+1}, …, e_n are ***not*** evaluated.
- If e_1, …, e_n all have value NIL, the value of (OR e_1 … e_n) is NIL.

From p. 123 of Touretzky:

```
(or 'fee 'fie 'foe) ⇒

(or nil 'foe nil) ⇒

(or nil t t)   ⇒

(or 'george nil 'harry)   ⇒

(or 'george 'fred 'harry)   ⇒

(or nil 'fred 'harry)   ⇒
```

Like e_1 || … || e_n in C++ or Java, (OR e_1 … e_n) is evaluated using ***short-circuit evaluation***, as follows:

- The expressions e_1, …, e_n are evaluated *in that order*, but evaluation of these expressions stops when an expression e_i is found to have a value that **isn't NIL**: When that happens, the value of e_i is returned as the value of (OR e_1 … e_n), and any subsequent expressions e_{i+1}, …, e_n are ***not*** evaluated.
- If e_1, …, e_n all have value NIL, the value of (OR e_1 … e_n) is NIL.

From p. 123 of Touretzky:

(or 'fee 'fie 'foe) ⇒ fee (more precisely, FEE)

(or nil 'foe nil) ⇒

(or nil t t) ⇒

(or 'george nil 'harry) ⇒

(or 'george 'fred 'harry) ⇒

(or nil 'fred 'harry) ⇒

Like e_1 || … || e_n in C++ or Java, (OR e_1 … e_n) is evaluated using ***short-circuit evaluation***, as follows:

- The expressions e_1, …, e_n are evaluated *in that order*, but evaluation of these expressions stops when an expression e_i is found to have a value that **isn't NIL**: When that happens, the value of e_i is returned as the value of (OR e_1 … e_n), and any subsequent expressions e_{i+1}, …, e_n are ***not*** evaluated.
- If e_1, …, e_n all have value NIL, the value of (OR e_1 … e_n) is NIL.

From p. 123 of Touretzky:

(or 'fee 'fie 'foe) ⇒ fee (more precisely, FEE)

(or nil 'foe nil) ⇒ foe

(or nil t t) ⇒

(or 'george nil 'harry) ⇒

(or 'george 'fred 'harry) ⇒

(or nil 'fred 'harry) ⇒

Like e_1 || … || e_n in C++ or Java, (OR e_1 … e_n) is evaluated using ***short-circuit evaluation***, as follows:

- The expressions e_1, …, e_n are evaluated *in that order*, but evaluation of these expressions stops when an expression e_i is found to have a value that **isn't NIL**: When that happens, the value of e_i is returned as the value of (OR e_1 … e_n), and any subsequent expressions e_{i+1}, …, e_n are ***not*** evaluated.
- If e_1, …, e_n all have value NIL, the value of (OR e_1 … e_n) is NIL.

From p. 123 of Touretzky:

(or 'fee 'fie 'foe) ⇒ fee (more precisely, FEE)

(or nil 'foe nil) ⇒ foe

(or nil t t) ⇒ t

(or 'george nil 'harry) ⇒

(or 'george 'fred 'harry) ⇒

(or nil 'fred 'harry) ⇒

Like e_1 || … || e_n in C++ or Java, (OR e_1 … e_n) is evaluated using ***short-circuit evaluation***, as follows:

- The expressions e_1, …, e_n are evaluated *in that order*, but evaluation of these expressions stops when an expression e_i is found to have a value that **isn't NIL**: When that happens, the value of e_i is returned as the value of (OR e_1 … e_n), and any subsequent expressions e_{i+1}, …, e_n are ***not*** evaluated.
- If e_1, …, e_n all have value NIL, the value of (OR e_1 … e_n) is NIL.

From p. 123 of Touretzky:

(or 'fee 'fie 'foe) ⇒ fee (more precisely, FEE)

(or nil 'foe nil) ⇒ foe

(or nil t t) ⇒ t

(or 'george nil 'harry) ⇒ george

(or 'george 'fred 'harry) ⇒

(or nil 'fred 'harry) ⇒

Like e_1 || … || e_n in C++ or Java, (OR e_1 … e_n) is evaluated using ***short-circuit evaluation***, as follows:

- The expressions e_1, …, e_n are evaluated *in that order*, but evaluation of these expressions stops when an expression e_i is found to have a value that **isn't NIL**: When that happens, the value of e_i is returned as the value of (OR e_1 … e_n), and any subsequent expressions e_{i+1}, …, e_n are ***not*** evaluated.
- If e_1, …, e_n all have value NIL, the value of (OR e_1 … e_n) is NIL.

From p. 123 of Touretzky:

(or 'fee 'fie 'foe) ⇒ fee (more precisely, FEE)

(or nil 'foe nil) ⇒ foe

(or nil t t) ⇒ t

(or 'george nil 'harry) ⇒ george

(or 'george 'fred 'harry) ⇒ george

(or nil 'fred 'harry) ⇒

Like e_1 || … || e_n in C++ or Java, (OR e_1 … e_n) is evaluated using ***short-circuit evaluation***, as follows:

- The expressions e_1, …, e_n are evaluated *in that order*, but evaluation of these expressions stops when an expression e_i is found to have a value that **isn't NIL**: When that happens, the value of e_i is returned as the value of (OR e_1 … e_n), and any subsequent expressions e_{i+1}, …, e_n are ***not*** evaluated.
- If e_1, …, e_n all have value NIL, the value of (OR e_1 … e_n) is NIL.

From p. 123 of Touretzky:

(or 'fee 'fie 'foe) ⇒ fee (more precisely, FEE)

(or nil 'foe nil) ⇒ foe

(or nil t t) ⇒ t

(or 'george nil 'harry) ⇒ george

(or 'george 'fred 'harry) ⇒ george

(or nil 'fred 'harry) ⇒ fred

The AND Macro Operator

(AND e_1 … e_n) is analogous to e_1 && … && e_n in C++ or Java, except that when (AND e_1 … e_n)'s value is *true* its value is ***the value of*** e_n (which will not be NIL but need not be T).

The AND Macro Operator

(AND e_1 … e_n) is analogous to e_1 && … && e_n in C++ or Java, except that when (AND e_1 … e_n)'s value is *true* its value is ***the value of e_n*** (which will not be NIL but need not be T).

From p. 122 of Touretzky:

Suppose we want a predicate for small (no more than two digit) positive odd numbers. We can use AND to express this conjunction of simple conditions:

```
(defun small-positive-oddp (x)
  (and (< x 100)
       (> x 0)
       (oddp x)))
```

The AND Macro Operator

(AND e_1 … e_n) is analogous to e_1 && … && e_n in C++ or Java, except that when (AND e_1 … e_n)'s value is *true* its value is ***the value of e_n*** (which will not be NIL but need not be T).

Another Example

The AND Macro Operator

(AND e_1 … e_n) is analogous to e_1 && … && e_n in C++ or Java, except that when (AND e_1 … e_n)'s value is *true* its value is ***the value of*** e_n (which will not be NIL but need not be T).

Another Example Suppose g is defined as follows:

```
(defun g (x)  (and (member x '(A B)) (member x '(C B 2 3))))
```

Then:

- (g 'B) ⇒
- (g 'A) ⇒
- (g 2) ⇒
- (g 6) ⇒

The AND Macro Operator

(AND e_1 … e_n) is analogous to e_1 && … && e_n in C++ or Java, except that when (AND e_1 … e_n)'s value is *true* its value is ***the value of*** e_n (which will not be NIL but need not be T).

Another Example Suppose g is defined as follows:

```
(defun g (x)  (and (member x '(A B)) (member x '(C B 2 3))))
```

Then:
- (g 'B) ⇒ (B 2 3)
- (g 'A) ⇒
- (g 2) ⇒
- (g 6) ⇒

The AND Macro Operator

(AND e_1 … e_n) is analogous to e_1 && … && e_n in C++ or Java, except that when (AND e_1 … e_n)'s value is *true* its value is ***the value of*** e_n (which will not be NIL but need not be T).

Another Example Suppose g is defined as follows:

```
(defun g (x)  (and (member x '(A B)) (member x '(C B 2 3))))
```

Then:
- (g 'B) ⇒ (B 2 3)
- (g 'A) ⇒ NIL
- (g 2) ⇒
- (g 6) ⇒

The AND Macro Operator

(AND e_1 … e_n) is analogous to e_1 && … && e_n in C++ or Java, except that when (AND e_1 … e_n)'s value is *true* its value is ***the value of e_n*** (which will not be NIL but need not be T).

Another Example Suppose g is defined as follows:

```
(defun g (x)  (and (member x '(A B)) (member x '(C B 2 3))))
```

Then:
- (g 'B) ⇒ (B 2 3)
- (g 'A) ⇒ NIL
- (g 2) ⇒ NIL
- (g 6) ⇒

The AND Macro Operator

(AND e_1 … e_n) is analogous to e_1 && … && e_n in C++ or Java, except that when (AND e_1 … e_n)'s value is *true* its value is ***the value of e_n*** (which will not be NIL but need not be T).

Another Example Suppose g is defined as follows:

```
(defun g (x)  (and (member x '(A B)) (member x '(C B 2 3))))
```

Then:
- (g 'B) ⇒ (B 2 3)
- (g 'A) ⇒ NIL
- (g 2) ⇒ NIL
- (g 6) ⇒ NIL

The AND Macro Operator

(AND e_1 … e_n) is analogous to e_1 && … && e_n in C++ or Java, except that when (AND e_1 … e_n)'s value is *true* its value is ***the value of*** e_n (which will not be NIL but need not be T).

Another Example Suppose g is defined as follows:

```
(defun g (x)  (and (member x '(A B)) (member x '(C B 2 3))))
```

Then:

- (g 'B) ⇒ (B 2 3)
- (g 'A) ⇒ NIL
- (g 2) ⇒ NIL
- (g 6) ⇒ NIL

Like e_1 && … && e_n in C++ or Java, (AND e_1 … e_n) is evaluated using ***short-circuit evaluation***, as follows:

•

•

The AND Macro Operator

(AND e_1 … e_n) is analogous to e_1 && … && e_n in C++ or Java, except that when (AND e_1 … e_n)'s value is *true* its value is ***the value of e_n*** (which will not be NIL but need not be T).

Another Example Suppose g is defined as follows:

```
(defun g (x)  (and (member x '(A B)) (member x '(C B 2 3))))
```

Then:
- (g 'B) ⇒ (B 2 3)
- (g 'A) ⇒ NIL
- (g 2) ⇒ NIL
- (g 6) ⇒ NIL

Like e_1 && … && e_n in C++ or Java, (AND e_1 … e_n) is evaluated using ***short-circuit evaluation***, as follows:

- The expressions e_1, …, e_n are evaluated *in that order*, but evaluation of these expressions stops when an expression e_i is found to have value NIL: When that happens,

-

The AND Macro Operator

(AND e_1 … e_n) is analogous to e_1 && … && e_n in C++ or Java, except that when (AND e_1 … e_n)'s value is *true* its value is ***the value of*** e_n (which will not be NIL but need not be T).

Another Example Suppose g is defined as follows:

```
(defun g (x)  (and (member x '(A B)) (member x '(C B 2 3))))
```

Then:
- (g 'B) ⇒ (B 2 3)
- (g 'A) ⇒ NIL
- (g 2) ⇒ NIL
- (g 6) ⇒ NIL

Like e_1 && … && e_n in C++ or Java, (AND e_1 … e_n) is evaluated using ***short-circuit evaluation***, as follows:

- The expressions e_1, …, e_n are evaluated *in that order*, but evaluation of these expressions stops when an expression e_i is found to have value NIL: When that happens, NIL is returned as the value of (AND e_1 … e_n), and any subsequent expressions e_{i+1}, …, e_n are ***not*** evaluated.
-

The AND Macro Operator

(AND e_1 … e_n) is analogous to e_1 && … && e_n in C++ or Java, except that when (AND e_1 … e_n)'s value is *true* its value is ***the value of e_n*** (which will not be NIL but need not be T).

Another Example Suppose g is defined as follows:

```
(defun g (x)  (and (member x '(A B)) (member x '(C B 2 3))))
```

Then:
- (g 'B) ⇒ (B 2 3)
- (g 'A) ⇒ NIL
- (g 2) ⇒ NIL
- (g 6) ⇒ NIL

Like e_1 && … && e_n in C++ or Java, (AND e_1 … e_n) is evaluated using ***short-circuit evaluation***, as follows:

- The expressions e_1, …, e_n are evaluated *in that order*, but evaluation of these expressions stops when an expression e_i is found to have value NIL: When that happens, NIL is returned as the value of (AND e_1 … e_n), and any subsequent expressions e_{i+1}, …, e_n are ***not*** evaluated.
- If e_1, …, e_n all have non-NIL values, then the value of (AND e_1 … e_n) is the value of e_n.