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, 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).

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.

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); 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.

From p. 123 of Touretzky:

```
(and 'fee 'fie 'foe) ⇒ foe (more precisely, FOE)

(and 'fee 'fie nil) ⇒ nil

(and (equal 'abc 'abc) 'yes) ⇒ yes

(and 'george nil 'harry) ⇒ nil

(and 'george 'fred 'harry) ⇒ harry

(and 1 2 3 4 5) ⇒ 5
```

In Java programming, it is common for the correctness of code to depend on the fact that && and || expressions are evaluated using short-circuit evaluation.

Example 1A: If s == **null**, evaluation of the Java expression

`(s.length() == 4 && s != null)`

BAD!

throws a **NullPointerException** when `s.length()` is called. (Assume `s` is of type **String**.)

Example 1B: If s == **null**, evaluation of the Java expression

`(s != null && s.length() == 4)`

GOOD!

returns **false** with ***no*** **NullPointerException** being thrown, as `s.length() == 4` isn't evaluated.

Example 2A: If n is an **int** and n == 0, the Java expression

`(100/n > 7 || n == 0)`

BAD!

has no value: When `100/n` is evaluated, an **ArithmeticException** is thrown because of ÷-by-0.

Example 2B: If n is an **int** and n == 0, the Java expression

`(n == 0 || 100/n > 7)`

GOOD!

evaluates to **true**: ***No*** **ArithmeticException** is thrown, as `100/n > 7` is not evaluated.

In Lisp, suppose we define PAY-BONUSES-P as follows:

```
 (defun pay-bonuses-p (pool num-awardees)
   (and (> num-awardees 0) (> (/ pool num-awardees) 1000)))
```

Q. What happens if we evaluate (pay-bonuses-p 10000 0)?

A. NIL is returned: There's no ÷-by-0 error because (/ pool num-awardees) is never evaluated: The (and …) immediately returns NIL when (> num-awardees 0) ⇒ NIL.

Suppose we define ALT2-PAY-BONUSES-P as follows:

```
 (defun strict-and (x y) (and x y))

 (defun alt2-pay-bonuses-p (pool num-awardees)
   (strict-and (> num-awardees 0)
               (> (/ pool num-awardees) 1000)))
```

Q. What happens if we evaluate (alt2-pay-bonuses-p 10000 0)?

A. ÷-by-0 error, because strict-and is an ordinary function, and so calls of strict-and are evaluated as follows:

1. Evaluate the call's argument expressions and place their values into strict-and's formal parameters x and y.
2. Evaluate strict-and's body—i.e., evaluate (and x y).

Step 1 gives a ÷-by-0 error when (/ pool num-awardees) is evaluated while evaluating (> (/ pool num-awardees) 1000).

Q. What happens when (car 7) is evaluated?
A. An *evaluation error* occurs because 7 is not a list.
Suppose we define IS-ATOM-OR-HAS-ATOMIC-CAR-P like this:

```
 (defun is-atom-or-has-atomic-car-p (e)
   (or (atom e) (atom (car e))))
```

Q. What happens if we evaluate (is-atom-or-has-atomic-car-p 7)?
A. T is returned: There is *no evaluation error* as (car e) is not evaluated: The (or …) returns T when (atom e) ⇒ T.

Suppose we define ALT2-IS-ATOM-OR-HAS-ATOMIC-CAR-P like this:

```
 (defun strict-or (x y) (or x y))
 (defun alt2-is-atom-or-has-atomic-car-p (e)
   (strict-or (atom e) (atom (car e))))
```

Q. What happens if we evaluate
(alt2-is-atom-or-has-atomic-car-p 7)?

A. An *evaluation error* occurs, as strict-or is an ordinary function and so its argument (atom (car e)) is evaluated:

- When (car e) is evaluated during evaluation of strict-or's argument (atom (car e)), an *evaluation error* occurs because e ⇒ 7, which is not a list.

p. 125 of Touretzky gives another example of the usefulness of short-circuit evaluation:

```
(defun posnump (x)
  (and (numberp x) (plusp x)))
```

POSNUMP returns T if its input is a number and is positive. The built-in PLUSP predicate can be used to tell if a number is positive, but if PLUSP is used on something other than a number, it signals a ‘‘wrong type input’’ error, so it is important to make sure that the input to POSNUMP is a number *before* invoking PLUSP. If the input isn't a number, we must not call PLUSP.

Here is an incorrect version of POSNUMP:

```
(defun faulty-posnump (x)
  (and (plusp x) (numberp x)))
```

If FAULTY-POSNUMP is called on the symbol FRED instead of a number, the first thing it does is check if FRED is greater than 0, which causes a wrong type input error. However, if the regular POSNUMP function is called with input FRED, the NUMBERP predicate returns NIL, so AND returns NIL *without ever calling* PLUSP.

LET gives values to ***local*** variables for use in an expression.

Example:
```
(let ((x (- 2 1))
      (y 3)
      (z (* 2 4)))
  (+ x (* y z)))   ⇒  25
```

This LET expression can be understood as meaning
    (+ x (* y z)) **where** x = (- 2 1), y = 3, z = (* 2 4)

The scope of the local variables introduced by a LET expression ***is confined to the body of the expression***.

- To illustrate this, suppose we define a function as follows:

```
(defun g (x)
  (list (let ((x 10))
          (* x x))
        x))
```

Then ***this*** x is the parameter x of g, which is ***unrelated* to the local variable x of the LET**!

Hence: (g 3) ⇒ **(100 3)**

Examples from pp. 141 – 3 of Touretzky:

So far, the only local variables we've seen have been those created by calling user-defined functions, such as DOUBLE or AVERAGE. Another way to create a local variable is with the LET special function. For example, since the average of two numbers is half their sum, we might want to use a local variable called SUM inside our AVERAGE function. We can use LET to create this local variable and give it the desired initial value. Then, in the body of the LET form, we can compute the average.

```
(defun average (x y)
  (let ((sum (+ x y)))
    (list x y 'average 'is (/ sum 2.0))))

> (average 3 7)
(3 7 AVERAGE IS 5.0)
```

The right way to read a LET form such as

```
(let ((x 2)
      (y 'aardvark))
  (list x y))
```

is to say "Let X be 2, and Y be AARDVARK; return (LIST X Y)."

Examples from pp. 141 – 3 of Touretzky:

```
(defun average (x y)
  (let ((sum (+ x y)))
    (list x y 'average 'is (/ sum 2.0))))

> (average 3 7)
(3 7 AVERAGE IS 5.0)


(defun switch-billing (x)
  (let ((star (first x))
        (co-star (third x)))
    (list co-star 'accompanied 'by star)))

> (switch-billing '(fred and ginger))
(GINGER ACCOMPANIED BY FRED)
```

- These examples illustrate one reason we use LET (or LET*): ***To give meaningful names (e.g.,*** `sum`***,*** `star`***, and*** `co-star`***) to the values of certain expressions and so make code more readable***. Another reason to use LET or LET* will be discussed later.

Evaluation of LET Forms

The general syntax of LET is:

```
(LET ((var-1 value-1)
      (var-2 value-2)
      ...
      (var-n value-n))
  body)
```

This is from p. 142 of Touretzky.

In functional programming, `body` **consists of *just one* expression whose value will be returned as the value of the entire LET form.**

The first argument to LET is a list of variable-value pairs. The *n* value forms are evaluated, then *n* local variables are created to hold the results, finally the forms in the body of the LET are evaluated.