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), 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.

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) ⇒

(and 'fee 'fie nil) ⇒

(and (equal 'abc 'abc) 'yes) ⇒

(and 'george nil 'harry) ⇒

(and 'george 'fred 'harry) ⇒

(and 1 2 3 4 5) ⇒
```

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) ⇒

(and (equal 'abc 'abc) 'yes) ⇒

(and 'george nil 'harry) ⇒

(and 'george 'fred 'harry) ⇒

(and 1 2 3 4 5) ⇒

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) ⇒

(and 'george nil 'harry) ⇒

(and 'george 'fred 'harry) ⇒

(and 1 2 3 4 5) ⇒

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) ⇒

(and 'george 'fred 'harry) ⇒

(and 1 2 3 4 5) ⇒

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) ⇒

(and 1 2 3 4 5) ⇒

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) ⇒

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

Code Whose Correctness Depends on Short-Circuit Evaluation

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:

Example 1B:

Example 2A:

Example 2B:

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:

Example 2A:

Example 2B:

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:

Example 2B:

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:

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.

A.

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.

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

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.

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.`

Next, suppose we define ALT1-PAY-BONUSES-P as follows:

```
(defun alt1-pay-bonuses-p (pool num-awardees)
  (and (> (/ pool num-awardees) 1000) (> num-awardees 0)))
```

Q. What happens if we evaluate (alt1-pay-bonuses-p 10000 0)?

A.

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.

Next, suppose we define ALT1-PAY-BONUSES-P as follows:

```
(defun alt1-pay-bonuses-p (pool num-awardees)
  (and (> (/ pool num-awardees) 1000) (> num-awardees 0)))
```

Q. What happens if we evaluate (alt1-pay-bonuses-p 10000 0)?

A. **÷-by-0 error** occurs when evaluating (/ pool num-awardees) during evaluation of (> (/ pool num-awardees) 1000).

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.

Next, suppose we define ALT1-PAY-BONUSES-P as follows:

```
(defun alt1-pay-bonuses-p (pool num-awardees)
  (and (> (/ pool num-awardees) 1000) (> num-awardees 0)))
```

Q. What happens if we evaluate (alt1-pay-bonuses-p 10000 0)?

A. **÷-by-0 error** occurs when evaluating (/ pool num-awardees) during evaluation of (> (/ pool num-awardees) 1000).

Our next example depends on the fact that calls of ordinary Lisp functions are evaluated using ***call-by-value*** **parameter passing** (just as calls of Java functions are): Arguments are evaluated and their ***values*** are passed into the call.

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.

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.

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.

A.

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.

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.

2.

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.

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).

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.

Q. What happens when (car 7) is evaluated?
A. An *evaluation error* occurs because 7 is not a list.

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.

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.

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 ALT1-IS-ATOM-OR-HAS-ATOMIC-CAR-P like this:

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

Q. What happens if we evaluate
(alt1-is-atom-or-has-atomic-car-p 7)?

A.

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 ALT1-IS-ATOM-OR-HAS-ATOMIC-CAR-P like this:

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

Q. What happens if we evaluate
(alt1-is-atom-or-has-atomic-car-p 7)?

A. An *evaluation error* occurs when (car e) is evaluated during evaluation of (atom (car e)), because e ⇒ 7, which is not a list.

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.

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.

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.

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:

-

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 and LET*

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

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

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
```

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)

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***.

•

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

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) ⇒

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))))
```

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)
```

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)
```

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)
```

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)
```

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))
```

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)
```

-

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:

This is from p. 142 of Touretzky.

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

•

•

•

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.

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.

•

•

•

Evaluation of LET Forms

The

general syntax of LET is:

This is from p. 142 of Touretzky.

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

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.

-
-
-

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.

- The expressions `value-1`, … , `value-n` are evaluated using the variable bindings that ***existed just before*** the LET expression.
-
-

Evaluation of LET Forms

The general syntax of LET is:

This is from p. 142 of Touretzky.

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

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.

- The expressions value-1, … , value-n are evaluated using the variable bindings that *existed just **before*** the LET expression.
- The local variables var-1, … , var-n will be given the values of the expressions value-1, … , value-n, but *they are **not** visible when those expressions are being evaluated*!
-

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.

- The expressions value-1, … , value-n are evaluated using the variable bindings that ***existed just before*** the LET expression.
- The local variables var-1, … , var-n will be given the values of the expressions value-1, … , value-n, but *they are **not** visible when those expressions are being evaluated*!
- Once the body expression has been evaluated, *the n local variables cease to exist*: var-1, … , var-n will then have the values, ***if any***, that they had before the LET expression.

```
(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)
```

From p. 143 of Touretzky.

Here is an evaltrace showing exactly how LET creates the local variables STAR and CO-STAR. Note that the two value forms, (FIRST X) and (THIRD X), are both evaluated before any local variables are created.

```
(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)
```

From p. 143 of Touretzky.

Here is an evaltrace showing exactly how LET creates the local variables STAR and CO-STAR. Note that the two value forms, (FIRST X) and (THIRD X), are both evaluated before any local variables are created.

(switch-billing '(fred and ginger))

Enter SWITCH-BILLING with input (FRED AND GINGER)

create variable X, with value (FRED AND GINGER)

(let ...)

(first x)

FRED

(third x)

GINGER

Enter LET body

```
(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)
```

From p. 143 of Touretzky.

Here is an evaltrace showing exactly how LET creates the local variables STAR and CO-STAR. Note that the two value forms, (FIRST X) and (THIRD X), are both evaluated before any local variables are created.

(switch-billing '(fred and ginger))

Enter SWITCH-BILLING with input (FRED AND GINGER)

create variable X, with value (FRED AND GINGER)

(let ...)

(first x)

FRED

(third x)

GINGER

Enter LET body

create variable STAR, with value FRED

create variable CO-STAR, with value GINGER

```
(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)
```

From p. 143 of Touretzky.

Here is an evaltrace showing exactly how LET creates the local variables STAR and CO-STAR. Note that the two value forms, (FIRST X) and (THIRD X), are both evaluated before any local variables are created.

```
(switch-billing '(fred and ginger))
Enter SWITCH-BILLING with input (FRED AND GINGER)
  create variable X, with value (FRED AND GINGER)
    (let ...)
      (first x)
      FRED
      (third x)
      GINGER
    Enter LET body
      create variable STAR, with value FRED
      create variable CO-STAR, with value GINGER
      (list co-star 'accompanied 'by star)
      (GINGER ACCOMPANIED BY FRED)
```

```
(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)
```

From p. 143 of Touretzky.

Here is an evaltrace showing exactly how LET creates the local variables STAR and CO-STAR. Note that the two value forms, (FIRST X) and (THIRD X), are both evaluated before any local variables are created.

```
(switch-billing '(fred and ginger))
Enter SWITCH-BILLING with input (FRED AND GINGER)
  create variable X, with value (FRED AND GINGER)
    (let ...)
      (first x)
      FRED
      (third x)
      GINGER
    Enter LET body
      create variable STAR, with value FRED
      create variable CO-STAR, with value GINGER
      (list co-star 'accompanied 'by star)
      (GINGER ACCOMPANIED BY FRED)
    Result of LET is (GINGER ACCOMPANIED BY FRED)
Result of SWITCH-BILLING is (GINGER ACCOMPANIED BY FRED)
```

Another example:

```
(defun g (w x y)
  (+ (let ((a (sqrt x))
           (b (* x 2))
           (c (+ x y)))
       (/ (+ a b c) w))
     x))
```

Evaluation of (g 3 4 7):

(g 3 4 7) ⇒

Another example:

```
(defun g (w x y)
  (+ (let ((a (sqrt x))
           (b (* x 2))
           (c (+ x y)))
       (/ (+ a b c) w))
     x))
```

Evaluation of (g 3 4 7):

w ⇒ x ⇒ y ⇒

the LET's local a ⇒

the LET's local b ⇒

the LET's local c ⇒

LET ⇒

x ⇒

(g 3 4 7) ⇒

Another example:

```
(defun g (w x y)
  (+ (let ((a (sqrt x))
           (b (* x 2))
           (c (+ x y)))
       (/ (+ a b c) w))
     x))
```

Evaluation of (g 3 4 7):

w ⇒ **3** x ⇒ **4** y ⇒ **7**

the LET's local a ⇒

the LET's local b ⇒

the LET's local c ⇒

LET ⇒

x ⇒

(g 3 4 7) ⇒

Another example:

```
(defun g (w x y)
  (+ (let ((a (sqrt x))
           (b (* x 2))
           (c (+ x y)))
       (/ (+ a b c) w))
     x))
```

Evaluation of (g 3 4 7):

w ⇒ **3** x ⇒ **4** y ⇒ **7**

the LET's local a ⇒ **2**

the LET's local b ⇒

the LET's local c ⇒

LET ⇒

x ⇒

(g 3 4 7) ⇒

Another example:

```
(defun g (w x y)
  (+ (let ((a (sqrt x))
           (b (* x 2))
           (c (+ x y)))
       (/ (+ a b c) w))
     x))
```

```
Evaluation of (g 3 4 7):
w ⇒ 3  x ⇒ 4  y ⇒ 7
the LET's local a ⇒ 2
the LET's local b ⇒ 8
the LET's local c ⇒
LET ⇒
x ⇒

(g 3 4 7) ⇒
```

Another example:

```
(defun g (w x y)
  (+ (let ((a (sqrt x))
           (b (* x 2))
           (c (+ x y)))
       (/ (+ a b c) w))
     x))
```

Evaluation of (g 3 4 7):

w ⇒ **3** x ⇒ **4** y ⇒ **7**

the LET's local a ⇒ **2**

the LET's local b ⇒ **8**

the LET's local c ⇒ **11**

LET ⇒

x ⇒

(g 3 4 7) ⇒

Another example:

```
(defun g (w x y)
  (+ (let ((a (sqrt x))
           (b (* x 2))
           (c (+ x y)))
       (/ (+ a b c) w))
     x))
```

Evaluation of (g 3 4 7):

w ⇒ **3** x ⇒ **4** y ⇒ **7**
the LET's local a ⇒ **2**
the LET's local b ⇒ **8**
the LET's local c ⇒ **11**
LET ⇒ **(2+8+11)/3 = 7**
x ⇒

(g 3 4 7) ⇒

Another example:

```
(defun g (w x y)
  (+ (let ((a (sqrt x))
           (b (* x 2))
           (c (+ x y)))
       (/ (+ a b c) w))
     x))
```

Evaluation of (g 3 4 7):

w ⇒ **3**   x ⇒ **4**   y ⇒ **7**

the LET's local a ⇒ **2**

the LET's local b ⇒ **8**

the LET's local c ⇒ **11**

LET ⇒ **(2+8+11)/3 = 7**

x ⇒ **4**

(g 3 4 7) ⇒

Another example:

```
(defun g (w x y)
  (+ (let ((a (sqrt x))
           (b (* x 2))
           (c (+ x y)))
       (/ (+ a b c) w))
     x))
```

Evaluation of (g 3 4 7):

w ⇒ **3** x ⇒ **4** y ⇒ **7**

the LET's local a ⇒ **2**

the LET's local b ⇒ **8**

the LET's local c ⇒ **11**

LET ⇒ **(2+8+11)/3 = 7**

x ⇒ **4**

(g 3 4 7) ⇒ **7+4 = 11**

A related example:

```
(defun h (w x y)
  (+ (let ((x (sqrt x))
           (y (* x 2))
           (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h 3 4 7):

(h 3 4 7) ⇒

A related example:

```
(defun h (w x y)
  (+ (let ((x (sqrt x))
           (y (* x 2))
           (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h 3 4 7):

w ⇒ x ⇒ y ⇒

the LET's local x ⇒

the LET's local y ⇒

the LET's local z ⇒

LET ⇒

x ⇒

(h 3 4 7) ⇒

A related example:

```
(defun h (w x y)
  (+ (let ((x (sqrt x))
           (y (* x 2))
           (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h 3 4 7):

w ⇒ **3**   x ⇒ **4**   y ⇒ **7**

the LET's local x ⇒

the LET's local y ⇒

the LET's local z ⇒

LET ⇒

x ⇒

(h 3 4 7) ⇒

A related example:

```
(defun h (w x y)
  (+ (let ((x (sqrt x))
           (y (* x 2))
           (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h 3 4 7):

```
w ⇒ 3  x ⇒ 4  y ⇒ 7
the LET's local x ⇒ 2
the LET's local y ⇒
the LET's local z ⇒
LET ⇒
x ⇒

(h 3 4 7) ⇒
```

A related example:

```
(defun h (w x y)
  (+ (let ((x (sqrt x))
           (y (* x 2))
           (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h 3 4 7):

w ⇒ **3** x ⇒ **4** y ⇒ **7**

the LET's local x ⇒ **2**

the LET's local y ⇒ **8**

the LET's local z ⇒

LET ⇒

x ⇒

(h 3 4 7) ⇒

A related example:

```
(defun h (w x y)
  (+ (let ((x (sqrt x))
           (y (* x 2))
           (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h 3 4 7):

w ⇒ **3** x ⇒ **4** y ⇒ **7**

the LET's local x ⇒ **2**

the LET's local y ⇒ **8**

the LET's local z ⇒ **11**

LET ⇒

x ⇒

(h 3 4 7) ⇒

A related example:

```
(defun h (w x y)
  (+ (let ((x (sqrt x))
           (y (* x 2))
           (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h 3 4 7):

w ⇒ **3** x ⇒ **4** y ⇒ **7**

the LET's local x ⇒ **2**

the LET's local y ⇒ **8**

the LET's local z ⇒ **11**

LET ⇒ **(2+8+11)/3 = 7**

x ⇒

(h 3 4 7) ⇒

A related example:

```
(defun h (w x y)
  (+ (let ((x (sqrt x))
           (y (* x 2))
           (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h 3 4 7):

w ⇒ **3** x ⇒ **4** y ⇒ **7**

the LET's local x ⇒ **2**

the LET's local y ⇒ **8**

the LET's local z ⇒ **11**

LET ⇒ **(2+8+11)/3 = 7**

x ⇒ **4**

(h 3 4 7) ⇒

A related example:

```
(defun h (w x y)
  (+ (let ((x (sqrt x))
           (y (* x 2))
           (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h 3 4 7):

w ⇒ **3** x ⇒ **4** y ⇒ **7**
the LET's local x ⇒ **2**
the LET's local y ⇒ **8**
the LET's local z ⇒ **11**
LET ⇒ **(2+8+11)/3 = 7**
x ⇒ **4**

(h 3 4 7) ⇒ **7+4 = 11**

Another example:

```
(defun g (w x y)
  (+ (let ((a (sqrt x))
           (b (* x 2))
           (c (+ x y)))
       (/ (+ a b c) w))
     x))
```

Evaluation of (g 3 4 7):

w ⇒ **3**   x ⇒ **4**   y ⇒ **7**

the LET's local a ⇒ **2**

the LET's local b ⇒ **8**

the LET's local c ⇒ **11**

LET ⇒ **(2+8+11)/3 = 7**

x ⇒ **4**

(g 3 4 7) ⇒ **7+4 = 11**

A related example:

```
(defun h (w x y)
  (+ (let ((x (sqrt x))
           (y (* x 2))
           (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h 3 4 7):

w ⇒ **3**   x ⇒ **4**   y ⇒ **7**

the LET's local x ⇒ **2**

the LET's local y ⇒ **8**

the LET's local z ⇒ **11**

LET ⇒ **(2+8+11)/3 = 7**

x ⇒ **4**

(h 3 4 7) ⇒ **7+4 = 11**

Evaluation of

```
    (let ((x1 expr1)
                ⋮
          (xn exprn))
      body)
```

is roughly equivalent to making a definition

and then calling the function as follows:

Evaluation of

```
(let ((x₁ expr₁)
          ⋮
      (xₙ exprₙ))
  body)
```

is roughly equivalent to making a definition

```
(defun my-helper-function (x₁ … xₙ) body)
```

and then calling the function as follows:

```
Evaluation of
    (let ((x₁ expr₁)
              ⋮
          (xₙ exprₙ))
      body)
is roughly equivalent to making a definition
    (defun my-helper-function (x₁ … xₙ)  body)
and then calling the function as follows:
    (my-helper-function  expr₁  …  exprₙ)

For example,
```

Evaluation of

```
(let ((x₁ expr₁)
        ⋮
      (xₙ exprₙ))
  body)
```

is roughly equivalent to making a definition

(defun my-helper-function (x_1 … x_n) *body*)

and then calling the function as follows:

(my-helper-function *expr*$_1$ … *expr*$_n$)

For example, evaluation of

```
(let ((a (sqrt x))
      (b (* x 2))
      (c (+ x y)))
  (/ (+ a b c) 5))
```

is roughly equivalent to making a definition

and then calling the function as follows:

Evaluation of

```
(let ((x₁ expr₁)
        ⋮
      (xₙ exprₙ))
  body)
```

is roughly equivalent to making a definition

```
(defun my-helper-function (x₁ … xₙ)  body)
```

and then calling the function as follows:

```
(my-helper-function  expr₁  …  exprₙ)
```

For example, evaluation of

```
(let ((a (sqrt x))
      (b (* x 2))
      (c (+ x y)))
  (/ (+ a b c) 5))
```

is roughly equivalent to making a definition

```
(defun my-helper-function (a b c)  (/ (+ a b c) 5))
```

and then calling the function as follows:

Evaluation of

```
(let ((x₁ expr₁)
           ⋮
      (xₙ exprₙ))
  body)
```

is roughly equivalent to making a definition

```
(defun my-helper-function (x₁ … xₙ)  body)
```

and then calling the function as follows:

```
(my-helper-function  expr₁  …  exprₙ)
```

For example, evaluation of

```
(let ((a (sqrt x))
      (b (* x 2))
      (c (+ x y)))
  (/ (+ a b c) 5))
```

is roughly equivalent to making a definition

```
(defun my-helper-function (a b c)  (/ (+ a b c) 5))
```

and then calling the function as follows:

```
(my-helper-function (sqrt x) (* x 2) (+ x y))
```

Evaluation of

```
(let ((x₁ expr₁)
           ⋮
      (xₙ exprₙ))
   body)
```

is roughly equivalent to making a definition

```
(defun my-helper-function (x₁ … xₙ)  body)
```

and then calling the function as follows:

```
(my-helper-function  expr₁  …  exprₙ)
```

Evaluation of

```
    (let ((x₁ expr₁)
               ⋮
          (xₙ exprₙ))
      body)
```

is roughly equivalent to making a definition

(defun my-helper-function (x_1 … x_n) *body*)

and then calling the function as follows:

(my-helper-function $expr_1$ … $expr_n$)

In fact

```
(let ((x₁ expr₁)
           ⋮
      (xₙ exprₙ))
  body)
```

is essentially equivalent to:

Evaluation of

```
(let ((x1 expr1)
          ⋮
      (xn exprn))
  body)
```

is roughly equivalent to making a definition

(defun my-helper-function (x_1 … x_n) *body*)

and then calling the function as follows:

(my-helper-function $expr_1$ … $expr_n$)

In fact

```
(let ((x1 expr1)
          ⋮
      (xn exprn))
  body)
```

is essentially equivalent to:

((lambda (x_1 … x_n) *body*) $expr_1$ … $expr_n$)

•

Evaluation of

```
(let ((x₁ expr₁)
         ⋮
      (xₙ exprₙ))
  body)
```

is roughly equivalent to making a definition

```
(defun my-helper-function (x₁ … xₙ)  body)
```

and then calling the function as follows:

```
(my-helper-function  expr₁  …  exprₙ)
```

In fact

```
(let ((x₁ expr₁)
         ⋮
      (xₙ exprₙ))
  body)
```

is essentially equivalent to:

```
((lambda (x₁ … xₙ) body)  expr₁  …  exprₙ)
```

- The latter expression calls a function that's the same as my-helper-function but has not been given a name.

LET*

LET* forms are equivalent to ***nested*** LET forms:

$$
\begin{array}{l}
\texttt{(let* ((}x_1\ \mathit{expr}_1\texttt{)} \\
\qquad\quad \texttt{(}x_2\ \mathit{expr}_2\texttt{)} \qquad\qquad \texttt{=} \\
\qquad\qquad\quad \vdots \\
\qquad\quad \texttt{(}x_n\ \mathit{expr}_n\texttt{))} \\
\quad \mathit{body}\texttt{)}
\end{array}
$$

LET*

LET* forms are equivalent to *nested* LET forms:

$$\begin{array}{lcl}
\texttt{(let* ((}x_1\ \mathit{expr}_1\texttt{)} & & \texttt{(let ((}x_1\ \mathit{expr}_1\texttt{))} \\
\qquad\texttt{(}x_2\ \mathit{expr}_2\texttt{)} & = & \quad\texttt{(let ((}x_2\ \mathit{expr}_2\texttt{))} \\
\qquad\quad\vdots & & \qquad\ddots \\
\qquad\texttt{(}x_n\ \mathit{expr}_n\texttt{))} & & \qquad\texttt{(let ((}x_n\ \mathit{expr}_n\texttt{))} \\
\quad \mathit{body}\texttt{)} & & \qquad\quad \mathit{body}\texttt{) …))}
\end{array}$$

-

LET*

LET* forms are equivalent to *nested* LET forms:

```
(let* ((x_1 expr_1)
       (x_2 expr_2)
            ⋮
       (x_n expr_n))
  body)
```

=

```
(let ((x_1 expr_1))
  (let ((x_2 expr_2))
       ⋱
    (let ((x_n expr_n))
      body) … ))
```

- Thus for $2 \le k \le n$ each expression $expr_k$ ***can use the previous local variables*** $x_1, \ldots, x_{k-1}$ (which would not be the case if we replaced LET* with LET).

LET*

LET* forms are equivalent to *nested* LET forms:

$$
\begin{array}{lcl}
\texttt{(let* ((}x_1\ expr_1\texttt{)} & & \texttt{(let ((}x_1\ expr_1\texttt{))} \\
\qquad\texttt{(}x_2\ expr_2\texttt{)} & = & \quad\texttt{(let ((}x_2\ expr_2\texttt{))} \\
\qquad\quad\vdots & & \qquad\ddots \\
\qquad\texttt{(}x_n\ expr_n\texttt{))} & & \qquad\texttt{(let ((}x_n\ expr_n\texttt{))} \\
\quad body\texttt{)} & & \qquad\quad body\texttt{)}\ \ldots\ \texttt{))}
\end{array}
$$

- Thus for $2 \le k \le n$ each expression $expr_k$ ***can use the previous local variables*** $x_1, \ldots, x_{k-1}$ (which would not be the case if we replaced LET* with LET).

On p. 144 of Touretzky, the difference between LET* and LET is described as follows:

The LET* special function is similar to LET, except it creates the local variables one at a time instead of all at once. Therefore, the first local variable forms part of the lexical context in which the value of the second variable is computed, and so on. This way of creating local variables is useful when one wants to assign names to several intermediate steps in a long computation.