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.

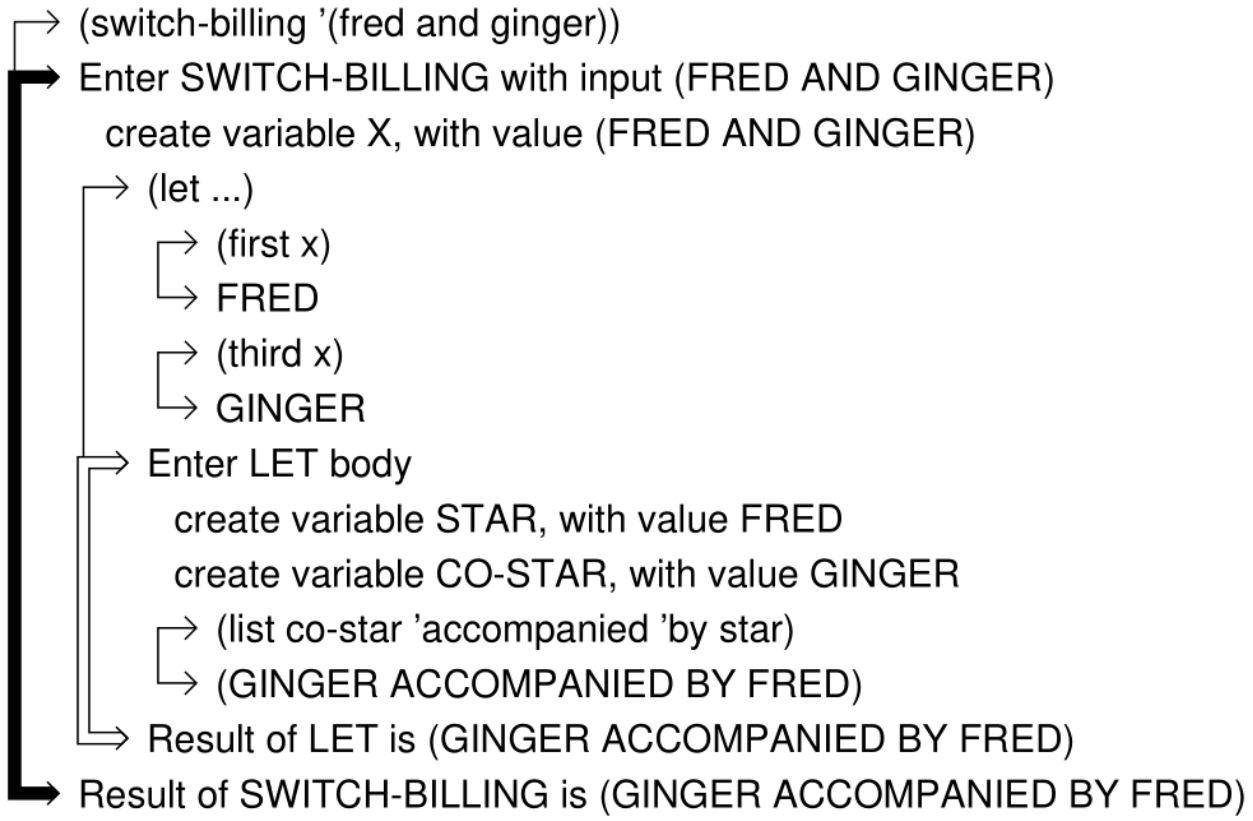
- The expressions `value-1, … , value-n` are evaluated using the variable bindings that ***existed just before*** the LET expression.
- The local variables `var-1, … , var-n` will be given the values of the expressions `value-1, … , value-n`, but ***they are not visible when those expressions are being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



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

```
   (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))
```

LET*

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

```
(let* ((x1 expr1)                    (let ((x1 expr1))
       (x2 expr2)          =           (let ((x2 expr2))
            ⋮                               ⋱
       (xn exprn))                        (let ((xn exprn))
  body)                                     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).

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.

From p. 144 of Touretzky:

For example, suppose we want a function that computes the percent change in the price of widgets given the old and new prices as input. Our function must compute the difference between the two prices, then divide this difference by the old price to get the proportional change in price, and then multiply that by 100 to get the percent change. We can use local variables named DIFF, PROPORTION, and PERCENTAGE to hold these values. We use LET* instead of LET because these variables must be created one at a time, since each depends on its predecessor.

```
(defun price-change (old new)
  (let* ((diff (- new old))
         (proportion (/ diff old))
         (percentage (* proportion 100.0)))
    (list 'widgets 'changed 'by percentage
          'percent)))

> (price-change 1.25 1.35)
(WIDGETS CHANGED BY 8.0 PERCENT)
```

From p. 145 of Touretzky:

A common programming error is to use LET when LET* is required. Consider the following FAULTY-SIZE-RANGE function. It uses MAX and MIN to find the largest and smallest of a group of numbers. MAX and MIN are built in to Common Lisp; they both accept one or more inputs. The extra 1.0 argument to / is used to force the result to be a floating point number rather than a ratio.

```
(defun faulty-size-range (x y z)
  (let ((biggest (max x y z))
        (smallest (min x y z))
        (r (/ biggest smallest 1.0)))
    (list 'factor 'of r)))

> (faulty-size-range 35 87 4)
Error in function SIZE-RANGE:
  BIGGEST unassigned variable.
```

The problem is that the expression (/ BIGGEST SMALLEST 1.0) is being evaluated in a lexical context that does not include these variables. Therefore the symbol BIGGEST is interpreted as a reference to a global variable

The following explanation of the difference between LET and LET* appears on p. 391 of Sethi (p. 7 of the course reader):

A sequential variant of the `let` construct is written with keyword `let*`. Unlike `let`, which evaluates all the expressions $E_1, E_2, \ldots, E_k$ before binding any of the variables, `let*` binds x_i to the value of E_i before E_{i+1} is evaluated. The syntax is

`(let* ((`x_1 E_1`) (`x_2 E_2`) ··· (`x_k E_k`)) F)`

The distinction between `let` and `let*` can be seen from the responses in

Use (setf x 0) here in Common Lisp.

~~(define x 0)~~

~~x~~

```
(let ((x 2) (y x)) y)     ; bind y before redefining x
   0
(let* ((x 2) (y x)) y)    ; bind y after redefining x
   2
```

Recall:

```
(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**

Replacing let with let*, we get:

```
(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 ⇒ **4**
the LET*'s local z ⇒ **6**
LET* ⇒ **(2+4+6)/3 = 4**
x ⇒ **4**

(h* 3 4 7) ⇒ **4+4 = 8**

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.
- Our own version of f can call the supposedly already written f; but when our version is called with an argument value *x*, it is only allowed to call the supposedly already written f *with an argument value that is* ***valid*** *for* f *and* ***smaller*** *in size than x*.

Example Write a function **length-of** such that:
If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

Assuming **length-of** has already been written, here is a function that works ***provided*** L ⇒ a **nonempty** list:

```
(defun my-length-of (L)
  (let ((X (length-of (cdr L))))
    (+ X 1)))
```

- If L ⇒ NIL, this violates the "call the supposedly already written f *with an argument value that is …* ***smaller***" condition.

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.

Example Write a function **length-of** such that:
If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

Assuming **length-of** has already been written, here is a function that works:

```
(defun better-my-length-of (L)
  (if (null L)
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

But this still assumes **length-of** has ***already*** been written.

Q. How can we write **length-of**?
A. We simply ***rename*** **better-my-length-of** to **length-of!**

Example Write a function **length-of** such that:
If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun better-my-length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

- This definition of **length-of** is ***not*** circular, because when **length-of** calls itself it always *passes an argument value that is **smaller** than the argument value it received.*
- ***If*** a recursive call **(length-of (cdr L))** returns the right result, ***then*** the call **(length-of L)** returns the right result.
- So, for all $n > 0$, ***if*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length < *n*, ***then*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length *n*.
- **Example:** ***If*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length 0, 1, 2, or 3, ***then*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length 4.

Example Write a function **length-of** such that:
If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun better-my-length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

- For all *n* > 0, ***if*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length < *n*, ***then*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length *n*.
- (length-of *l*) returns the right result (i.e., 0) when *l* ⇒ a list of length 0.

∴ If *l* ⇒ a proper list of any length,
then (length-of *l*) ⇒ the right result (i.e., *l*'s length).

- Although this function is correct as written, *we can improve / simplify the definition by* ***eliminating the*** **LET**, ***because its local variable*** X ***is never used more than once***.

We then replace the X in **(+ X 1)** with **(length-of (cdr L))**:

Example Write a function **length-of** such that:
If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun better-my-length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

- Although this function is correct as written, *we can improve / simplify the definition by* ***eliminating the*** **LET**, ***because its local variable*** X ***is never used more than once***. We then replace the X in **(+ X 1)** with **(length-of (cdr L))**:

```
(defun length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (+ (length-of (cdr L)) 1)))
```

- We've given a written explanation of a possible thought process that leads to this definition, but an experienced Lisp programmer would likely code simple definitions like this one without giving any explanation!

Example Write a function **factorial** such that:
If n ⇒ a non-negative integer, then (factorial n) ⇒ n!.

- Our own version of f can call the supposedly already written f; but when our version is called with an argument value *x*, it is only allowed to call the supposedly already written f *with an argument value that is **valid** for f and **smaller** in size than x.*

*Assuming **factorial** has already been written correctly*, here is a function that works ***provided*** n ⇒ a **nonzero** integer:

```
(defun my-factorial (n)
  (let ((X (factorial (- n 1))))
    (* n X)))
```

- **We use the fact that:**
  For example:

n * (n-1)! = n!
5 * 4! = 5 * 4 * 3 * 2 * 1 = 5!

- Importantly, **n * (n-1)! = n!** holds even when n = 1, as 0! = 1.
- If n ⇒ 0, the above definition ***violates*** the "call the supposedly already written f *with an argument value that is **valid** for f and **smaller** in size*" condition, because **(- n 1)** is ***not*** a valid argument value for **factorial** if n ⇒ 0.

Example Write a function **factorial** such that:
If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

```
(defun better-my-factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

- This definition of **factorial** is ***not*** circular, because when **factorial** calls itself it always *passes an argument value that is* ***smaller*** *than the argument value it received*.
- ***If*** a recursive call **(factorial (- n 1))** returns the right result, ***then*** the call **(factorial n)** returns the right result.
- So, for all positive integers *k*, ***if*** (factorial *i*) returns the right result whenever *i* ⇒ a nonnegative integer < *k*, ***then*** (factorial *i*) also returns the right result when *i* ⇒ *k*.
- **Example:** ***If*** (factorial *i*) returns the right result when *i* ⇒ 0, 1, 2, or 3, ***then*** (factorial *i*) also returns the right result when *i* ⇒ 4.

Example Write a function **factorial** such that:
If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

```
(defun better-my-factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

- For all positive integers *k*, ***if*** (factorial *i*) returns the right result whenever *i* ⇒ a nonnegative integer < *k*, ***then*** (factorial *i*) also returns the right result when *i* ⇒ *k*.
- (factorial *i*) returns the right result (i.e., 1) when *i* ⇒ 0.

∴ If *i* ⇒ any nonnegative integer,
then (factorial *i*) ⇒ the right result (i.e., *i*!).

- Although this function is correct as written, *we can improve / simplify the definition by* ***eliminating the*** **LET**, ***because its local variable*** X ***is never used more than once***.

We then replace the X in **(* n X)** with **(factorial (- n 1))**:

Example Write a function **factorial** such that:
If n ⇒ a non-negative integer, then (factorial n) ⇒ n!.

```
(defun better-my-factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

- Although this function is correct as written, *we can improve / simplify the definition by* ***eliminating the*** **LET**, ***because its local variable*** X ***is never used more than once***. We then replace the X in **(* n X)** with **(factorial (- n 1))**:

```
(defun factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      (* n (factorial (- n 1)))))
```

- As in the case of **length-of**, we've given a written explanation of a possible thought process that leads to this definition, but a Lisp programmer would likely code simple definitions like these without giving any explanation!

- Recursive functions of one argument, which is a list or a nonnegative integer, can often be written in the above way.
- The resulting definition will then have the following form (before possible elimination of the LET):

```
(defun f (e)
  (if (null e)
      value of (f nil)
      (let ((X (f (cdr e))))
         an expression that ⇒ value of (f e)
         and that involves X and, possibly, e )))
```

OR

```
(defun f (e)
  (if (zerop e)
      value of (f 0)
      (let ((X (f (- e 1))))
         an expression that ⇒ value of (f e)
         and that involves X and, possibly, e )))
```

- Recursive functions of one argument, which is a list or a nonnegative integer, can often be written as follows:

```
(defun f (e)
  (if (null e) or (zerop e)
      value of (f nil) or (f 0)
      (let ((X (f (cdr e)) or (f (- e 1)) ))
           an expression that ⇒ value of (f e)
           and that involves X and, possibly, e )))
```

- The … expression may have more than one case (as in problems **B**, **D**, and **G** of Lisp Assignment 4): The … expression may, e.g., be a **COND** or **IF** expression.
- If there is no case in which **X** is used more than once, then ***eliminate the LET***.
- If the LET isn't eliminated, ***move any case in which X needn't be used out of the*** LET. If the LET ***is*** eliminated but ***there's a case where the recursive call's result isn't needed, deal with such cases as base cases--i.e., without making a recursive call***.