LISP: Part 2

9 March, 2017

Janyl Jumadinova
Variables in LISP

- Let’s consider how you would write a function to convert from dollars to pounds. The procedure is as follows:

- Multiply the number of dollars by the exchange rate.

- The exchange rate is the number of pounds in 1 dollar; let’s say today it’s 0.82. So our procedure definition is simply:

  (defun convert (dollars) (* dollars 0.82))
Variables in LISP

- Let’s consider how you would write a function to convert from dollars to pounds. The procedure is as follows:
  - Multiply the number of dollars by the exchange rate.
  - The exchange rate is the number of pounds in 1 dollar; let’s say today it’s 0.82. So our procedure definition is simply:
    `(defun convert (dollars) (* dollars 0.82))`
  - In the definition of `convert`, the symbol `dollars` is a variable.
  - It’s called a *local variable* because the value is only available inside the body of the function.
Defining a variable: defparameter

- But one problem with `convert` is that we have to redefine it if the exchange rate changes.
- What we need is a variable that we can set to the current exchange rate. We can do that with `defparameter`:

  ```lisp
  (defparameter exchange-rate 0.82)
  ```
But one problem with convert is that we have to redefine it if the exchange rate changes.

What we need is a variable that we can set to the current exchange rate. We can do that with defparameter:

```lisp
(defun convert (dollars) (* dollars exchange-rate))
```

Variables defined by defparameter are called *global variables* because you can refer to them anywhere.

A common convention is to give each global variable a name starting and ending with an asterisk, such as *exchange – rate*, to remind you that it's global.
Creating local variables: \textit{let}* 

- The \textit{let}* construct allows you to define one or more variables, with specified initial values, which are local to the body of the \textit{let}* construct.
- It is written like this:
  \begin{verbatim}
  (let* ((var1 value1) 
         (var2 value2) 
         ...) 
    body)
  \end{verbatim}
- \textit{var1} is the first local variable and \textit{value1} is its initial value.
- \textit{var2} is the second local variable and \textit{value2} is its initial value, and so on.
- \textit{body} is one or more Lisp procedures that can refer to \textit{var1} and \textit{var2}. 
As an example of its use, let’s look at a definition of the average procedure:

```lisp
(defun average (1st-number 2nd-number)
  (/ (+ 1st-number 2nd-number) 2))
```

The parameters 1st-number and 2nd-number are local variables, local to the body of the procedure.
But suppose we wanted to break the calculation into steps, using a local variable sum to store the sum of the two numbers, and another local variable result to store the result of dividing this by 2.

We could do this with let*:

```lisp
(defun average (1st-number 2nd-number)
  (let* ((sum (+ 1st-number 2nd-number))
         (result (/ sum 2)))
    result))
```
But suppose we wanted to break the calculation into steps, using a local variable sum to store the sum of the two numbers, and another local variable result to store the result of dividing this by 2.

We could do this with `let*`:

```
(defun average (1st-number 2nd-number)
  (let* ((sum (+ 1st-number 2nd-number))
         (result (/ sum 2)))
    result))
```

You can also use `let`

`let` performs the bindings in parallel and `let*` does them sequentially
Strings

- You can include a double-quote in a string by prefixing it with a backslash:
  "He shouted \"Help!\" and ran away."
Strings

- You can include a double-quote in a string by prefixing it with a backslash:
  "He shouted \"Help!\" and ran away."

- reverse a string: (reverse "dog")
Strings

- You can include a double-quote in a string by prefixing it with a backslash:
  "He shouted "Help!" and ran away."

- reverse a string: (reverse "dog")

- concatenate two or more strings: (concatenate 'string "band" "age")

- Get a subsequence from a string: (subseq "averylongword" 5 9)
  - It takes three parameters: the string, the first character (starting at 0), and the character after the last character.
  - If you leave out the third parameter you get the rest of the string to the end.
When you evaluate a Lisp procedure in the Listener it returns a value.

But when you’re running a program you might want it to print out several values before the program returns a final result.
When you evaluate a Lisp procedure in the Listener it returns a value.

But when you’re running a program you might want it to print out several values before the program returns a final result.

print: simply prints out a printed representation of its argument, and then returns the argument.

- (print 123)

format: includes options for printing every type of value in every conceivable way.
The format procedure takes two or more parameters.

- The first parameter is either `t`, to tell the format procedure to print the result, or `nil`, to return the result as a string.
- The second parameter is a **format string**, which tells the format procedure how to print the result. This is a text string which can include special format sequences, prefixed by a "∽" character, to insert values in this string.
- The remaining parameters are evaluated to give the values to be inserted into the format string. The most general format sequence is "∽ a" which inserts the value as it would be printed by print.
Format

- `(format t "The answer is \textasciitilde a." (* 2 3))`
  - Inserts the value of (* 2 3) into the string specified by the \textasciitilde a, and prints:
    The answer is 6.

- You can also include \textasciitilde % in the format string to give a line break.

- Alternatively, by specifying the second parameter as `nil` we can use `format` to generate a string for us.
  `(format nil "The answer is \textasciitilde a." (* 2 3))`
= tests whether two or more numbers are equal: (\(= 2 2\))

> tests if the first number is greater than the second: (\(> 4 3\))

EQ test can also be used for numbers, but it’s more intuitive to use =.

string = tests whether strings are equal: (string= "cat" "cat")
Conditional test: if

- It evaluates the first parameter; if it returns a non-nil value, it evaluates and returns the second parameter; otherwise it evaluates and returns the third parameter.
  (if (evenp a)
   (print "Answer is even")
   (t print "Answer is odd"))
- This will print one of the two strings depending on whether a is even or odd.
Conditional test: if

- It evaluates the first parameter; if it returns a non-nil value, it evaluates and returns the second parameter; otherwise it evaluates and returns the third parameter.

  (if (evenp a)
    (print "Answer is even")
    (t print "Answer is odd"))

- This will print one of the two strings depending on whether a is even or odd.

  (defun maximum (a b)
    (if (> a b)
      (a)
      (t b)))

- This finds the maximum of two numbers.
Grouping procedures: progn

- The if form only allows you to give one parameter to be executed if the test is true, and one if the test is false.
- What happens if you want to do two things in either or both of those situations?

```lisp
(if (evenp a)
  (progn (print "Answer is even") 0)
  (progn (print "Answer is odd") 1))
```
Grouping procedures: progn

- The if form only allows you to give one parameter to be executed if the test is true, and one if the test is false.
- What happens if you want to do two things in either or both of those situations?
- The **progn** form allows you to wrap several procedures into one bracketed list.

```lisp
(if (evenp a)
   (progn (print "Answer is even") 0)
   (progn (t print "Answer is odd") 1))
```
Branching functions: case

\[
\text{(case } < \text{key} > \\
\text{(} < \text{key} - 1 > < \text{expr} - 1 - 1 > ... \\
\text{(} < \text{key} - 2 > < \text{expr} - 2 - 1 > ... \\
\text{... \\
\text{(} < \text{key} - m > < \text{expr} - m - 1 > ... ))
\]

- checks (top-down) the value of \(< \text{key} > \) against the unevaluated keys \(< \text{key} - 1 >, ..., < \text{key} - m > \) using eql.
- If the key is found, the corresponding clause is triggered by evaluating all expressions in its body.
Branching functions: case

>` (setq thing ’sphere r 1)
> 1
>`(case thing
>    (> circle (* pi r r))
>    (> sphere (* 4 pi r r)))
Equivalent *cond* expression:
>`(setq thing ’sphere r 1)
> 1
>`(cond
>    (> (eq thing ’circle) (* pi r r))
>    (> (eq thing sphere) (* 4 pi r r)))
Iteration

**loop:**

a loop construct repeatedly executes its body until it hits a return special form

```lisp
> (setq a 4) 4
> (loop
  (setq a (+ a 1))
  (when (> a 7) (return a))) 8
> (loop
  (setq a (- a 1))
  (when (< a 3) (return)))
NIL
```
dolist:

dolist binds a variable to the elements of a list in order and stops when it hits the end of the list

> (dolist (x '(a b c)) (print x)) A
B
C
NIL

- dolist always returns \textit{nil}.
- Note that the value of \texttt{x} in the above example was never nil: the NIL below the C was the value that dolist returned, printed by the read-eval-print loop.
do:
- The first part of a do specifies what variables to bind, what their initial values are, and how to update them.
- The second part specifies a termination condition and a return value.
- The last part is the body.

> (do ((x 1 (+ x 1))
     (y 1 (* y 2))
     ((> x 5) y)
     (print y)
     (print 'working)
)
1 WORKING
2 WORKING
4 WORKING
8 WORKING
16 WORKING
32
Recursion

- Easier to do in Lisp than in imperative languages (Java, C++), but still challenging at first
- Using LISP will give you new respect for and understanding of recursion

(defun count (x) ; Same as LISP's length function
  (if (atom x)
      0
      (1+ (count (cdr x)))))
(defun sum (x)
  (if (atom x)
      0
      (+ (car x) (sum (cdr x)))))
(defun average (x)
  (/ (sum x) (count x)))