Common Lisp Beginner's Guide

Michael Olson

Jan. 3, 2007

Contents

Preface

This document is meant to be a primer for those who have read at least one book on Common Lisp or Emacs Lisp, and are familiar with C. The intent is to show how to solve various problems that the aforementioned people may come across when attempting to learn Common Lisp.

Cons cells and lists

The structure of cons cells and lists is covered well in the Emacs Lisp Intro document that comes with Emacs, so I won't go into too much detail on that end. We will instead give examples of the ways that cons cells are used.

Basically, a cons cell is a singly-linked list item. The main difference between cons cells and most linked-list implementations is that there is no special class (different from the element class itself) that contains a "head" variable. Every list element can be a head, a tail, or even the very last element of a list (without the delimiting NULL value), though doing the latter would not make the result a proper list. Also, there is no limitation on the type of things you can store in a cons cell—the requirement for the contents of cons cells depends on the function you are passing one to.

Here are some examples of how cons cells are used to build lists.

Task 1: Iterate through a list and stop on a particular condition

The first example begins to walk through a list, but stops when some condition is met. In particular, we stop when we find one number that is not strictly greater than the one before it.

```
(defun strictly-greater? (list)
  (let ((current list)
        (previous nil)
        (result t))
    (while current
      (if (and previous
               (not (> (car current) previous)))
          (progn
            ;; this is equivalent to two separate setf statements
            (setf current nil
                  result nil))
          (setf previous (car current))
          (setf current (cdr current))))
    result))
```

Here's a version that uses `catch` and `throw` to beautify the code.

```
(defun strictly-greater? (list)
  (let ((current list)
        (previous nil))
    (catch 'failed
      (while current
        (if (and previous
                 (not (> (car current) previous)))
            (throw 'failed nil))
        (setf previous (car current))
        (setf current (cdr current)))
      ;; we made it through the list, so it is valid
      t)))
```

Task 2: Return a new list based on the given list

Another popular programmatical use of cons cells is to build a new list from the contents of an old list. This can be either a "filter" (something that preserves the order of the old list) or a "reverser" (something that flips the order of the list).

The easiest way to build a new list is to start with the `nil` ending symbol and build the list in reverse. Then if the original order of the list is to be preserved, call `nreverse` on it to flip the order. This is usually much better than building a list front-first, because then you would have to traverse the entire contents of the new list every time you want to add an item to it.

The first example is the "base case" for building a list: reversal.

```
(defun my-reverse (list)
  (let ((newlist nil))
    (dolist (item list)
      (setf newlist (cons item newlist)))
    newlist))
```

We use `dolist` here because it walks through the original list for us, assigning each value of the list to `item`, and then calling `setf`. Since we don't plan on stopping early, we can use `dolist` with impunity. Here's what it would look like if we did not use `dolist`.

```
(defun my-reverse (list)
  (let ((newlist nil)
        (current list))
    (while current
      (setf newlist (cons (car current) newlist))
      (setf current (cdr current)))
    newlist))
```

Now we give an example of turning a list of symbols into a list of strings, where each string is the name of the corresponding symbol. Note that if the user gives us a list of something other than symbols, the call to `symbol-value` will throw an error.

Here, the `mapcar` function calls its first argument (which must be a function) on every item in the second argument (which must be a list). It collects the results from each function call and returns them as a list.

```
(defun my-symbol-to-string (list)
  (mapcar #'symbol-name
          list))
```

Here is a version of that example that does not use `mapcar`.

```
(defun my-symbol-to-string (list)
  (let ((newlist nil)
        (current list))
    (while current
      (setf newlist (cons (symbol-name (car current))
                          newlist))
      (setf current (cdr current)))
    ;; we built the list in reverse order, so reverse it again to get
    ;; it in the correct order
    (nreverse newlist)))
```

The next example is more sophisticated, because instead of allowing an error to be thrown, it uses "INVALID". This is handy when you're just displaying a list instead of re-using it later on. Instead of passing the name of the function `symbol-name` to `mapcar`, we will pass a function that is created on-the-fly. The `#'` construct is a tip to the Lisp compiler to tell it that what follows is a function—it isn't strictly necessary, but it is good practice.

```
(defun my-symbol-to-string (list)
  (mapcar #'(lambda (item)
              (if (symbolp item)
                  (symbol-name item)
                  "INVALID"))
          list))
```

Aside Most types have a predicate (that is, something that tests to see whether the argument is something of that type) called `TYPEp` or `TYPE-p`, where `TYPE` is the name of the type. Sometimes a "?" character may be used instead of "-p", such as `TYPE?` in the variant of Lisp called Scheme. Looking back to our first example, you can see that we made a predicate function. I used the "?" notation there because it feels more modern.

Compiling and executing Lisp programs

Once you start making entire programs or small scripts in Lisp, you may wish to compile your code and run it from the command line, rather than manually running it from a Lisp interpreter.

Generalized solution

For a more general solution, check out the cl-launch[1] package. It enables you to make shell scripts that run Lisp programs and provides advice on what to put in Makefiles.

CLISP

If you're using the CLISP Common Lisp environment, the following applies.

To compile a program named `test.lisp` program into a bytecode file named `test.fas`, do the following.

```
clisp -c test.lisp
```

If you want less output to be shown, do the following instead.

```
clisp -q -c test.lisp
```

To run the compiled (or even uncompiled, if you skip the above step) Lisp file, do the following, assuming that your entry function is named `main`. Normally the result of the `main` function is shown when it's done, but the `(quit)` command prevents that. The `-on-error abort` option prevents clisp from dropping into a debugging prompt, and exits instead when there is an error.

```
clisp -q -q -on-error abort -x '(progn (load "test") (main) (quit))'
```

[1] http://www.cliki.net/cl-launch