SW483, Summer 2017
CS131 Homework Assignment #13, “Recursive Atom Counter”

Symbols Versus Variables

In comparing the LISP language to “imperative” languages , the language construct in LISP that most resembles “variables” in languages like Java, C, C++, or Fortran is the “symbol”. Both variables and symbols can be assigned values and can store values of an array of arithmetic, logical, and textual types in programs. Variables and symbols can be used in program expressions where their assigned values are retrieved and used in the expressions when the expressions are evaluated.

However, symbols in LISP differ from program variables in many ways. While in a language like Java, all variables must be declared to be a particular type before they are used, LISP allows for programs to assign values to symbols without first declaring the symbol. LISP also allows a symbol to be assigned a value of a particular type, and then the same symbol can be assigned a value of a different type later. In Java, a variable is declared to be a particular type once, and it remains that type for as long as its scope exists; in Java and other imperative languages, a variable cannot be redeclared, and it can only be assigned values that match its declared type.

Secondly, in LISP the *name* of a symbol can itself exist unevaluated as a value in program expressions. A symbol can appear in an expression and it can be selectively evaluated to retrieve the symbol’s value, or it can be left unevaluated. A symbol name can be passed as an argument to a function or it can be made the value of another symbol.

Here is an example from the LISP interpreter,

```
1  [1]> (setq attribute 'firstname)
2  FIRSTNAME
3  [2]> (setq firstname 'Bob)
4  BOB
5  [3]> firstname
6  BOB
7  [4]> attribute
8  FIRSTNAME
9  [5]> (eval attribute)
10 BOB
```

In the example above, the setq function is used to assign the value `'firstname` to the symbol `attribute`. The single quote character is used to prevent the evaluation of

`firstname` in the assignment expression. But `firstname` is itself a symbol, and in [2], `firstname` is assigned the value the symbol `Bob`. In `[3]` and `[4]` we can see that firstname and attribute evaluate to the values that we have assigned. Finally in `[5]` we use the `eval` function to add an extra evaluation: first `attribute` evaluates to `firstname`, and then the `eval` function evaluates `firstname` to its value, `Bob`.

Imperative programming languages like Java do not have the capability to use the name of a variable as a value that can be assigned to another variable. At most a Java String can represent the name of a variable, but then that String is only a sequence of text character values, and cannot programmatically be used to retrieve the value of the variable that also has that name [1]

Lists and Atoms

You should already be familiar with the distinction between s-expressions, lists and atoms in LISP. An expression such as `(A B C D)` is a list containing the symbols `A`, `B`, `C`, and `D` whereas any of the individual symbols in the list, like `A` considered alone in an expression is an atom. Other values, such as arithmetic values, like `-8` or `5.91`, or the values `T` or `nil` are also considered atoms. The empty list, denoted by `()`, is equivalent to `nil` and it also considered an atom.

For this assignment, you will implement a function called `recursive-count-atoms` that takes either a list or an atom as an argument. If the argument to the function is a list, it should evaluate to the number of atoms in the list. If the argument to the function is an atom, it should simply evaluate to 1.

A Recursive Decomposition Using Car and Cdr

Here is some starter code for `recursive-count-atoms`:

```
1 (defun recursive-count-atoms (x)
2   (cond
3     ((atom x) 1)
4     ;; Insert code here.
5    )
6   )
```

The function body has a single `cond` and the first form uses the `atom` predicate function which evaluates to T if x, the argument, is an atom. In that case the entire function call returns 1. If x is not an atom, the execution falls through to the end of

[1] Except possibly by very awkward means using the application programming interfaces (APIs) for reflection or introspection that Java provides.

the cond, and the cond automatically returns nil. Therefore this function will correctly return the value 1 if the input is an atom, but will incorrectly return the value `nil` if the input is say `(A B C D)`.

```
CL-USER> (recursive-count-atoms 4)
1
CL-USER> (recursive-count-atoms '(3 4))
NIL
```

Note: the list `(3 4)` must be quoted with the single quote character operator as `'(3 4)` to avoid the number 3 being evaluated as a function.

You can implement recursive-count-atoms by adding to the `cond` to deal with some cases where the argument is a list. The ideal way to decompose the task of counting the number of atoms in a list is to do the following:

1. Examine the first item in the list to see if it is an atom. If it is you'll add 1 to your count. If it is not, you'll add 0 to it.
2. Since you've examined the first item in the list, recursively call recursive-count-atoms on the remaining part of the list, and add that value to what you calculated from the first element in the list. Return the sum.

If you have a list with at least two elements, you can obtain the first element of the list by calling the `first` function (also known as `car`) with the list as the argument. You can obtain a list that consists of everything after the first element by calling the `rest` function (also known as `cdr`) with the list as an argument.

One wrinkle, however, is that `cdr` will return the empty list if the list you call it on only has a single element. The empty list is considered an atom, so the recursive call in this case will return one, adding an extra 1 to the count for an empty list. This will make the count one greater than it should be. The solution is to add an extra form to the cond for the case where the list only has a single element, and `cdr` is the empty list.

Here is some new starter code that handles this case.

```
(defun recursive-count-atoms (x)
  (cond
   ((atom x) 1)
   ((null (cdr x)) 1)
   ;; Insert code here.
   )
  )
```

...