Assignment 4: Common Lisp Program
Due: Thursday, 26 February 2015 at 11:59pm

Overview

The purpose of this assignment is for you to gain some experience designing and implementing LISP programs. In this assignment, however, you will explore only a few of the many interesting LISP features. The assignment is broken into three parts. The first part is a fairly straightforward LISP warm-up. The later parts require you to build a pattern-matching program.

Part 1: Simple Function Definitions

The goal of this part of the homework is to familiarize you with the notions of lists, function definitions, and function applications in Lisp. This part requires you to define a number of simple functions:

1. The lisp function `length` counts the number of elements in the top level of a list. Write a function, `all-length`, that takes a list and counts the number of atoms that occur in a list at all levels.

```
> (all-length '(a ( b ) c))
3
> (all-length '(a (b c) (d (e f))))
6
```

2. Define a function `range` that takes a list of numbers (with at least one element) and returns a list of length 2 of the smallest and largest numbers.

```
> (range '(0 7 8 2 3 -1))
(-1 8)
> (range '(7 6 5 4 3))
(3 7)
```

3. Write a function `before` that searches a list and returns a list of all elements in between the first two arguments (inclusive).

```
> (before 'b 'd '(a b c d))
(B C D)
```

   If only the first argument appears in the list, `before` returns a list containing all the elements from the first occurrence of the first argument to the end of the list.

```
> (before 'a 'd '(a))
(A)
```

4. Write a function `split-if` that returns a list into two parts. It takes two arguments: a function (f) and a list. All members for which f is true go into one list, and the rest go into another list.

```
> (split-if #'(lambda (x) (> x 4)) '(1 2 3 4 5 6 7 8 9 10))
((1 2 3 4) (5 6 7 8 9 10))
```

5. Write a function `group` that takes arguments: a list l and a number n. It returns a new list in which the elements of l are grouped into sublists of length n. The remainder is put in a final sublist.

```
> (group '(a b c d e f g) 2)
((A B) (C D) (E F) (G))
```

6. Write a function `mostn` that takes two arguments: a function f and a list l. It returns a list of all elements for which the function yields the highest score (along with the score itself), where score the value returned from the given function:

```
> (mostn #'length '((a b) (a b c) (a) (e f g)))
( ((A B C) (E F G)) 3)
```

Part 2: Assertions and Simple Pattern-Matching

Before we start building the pattern-matching function, let us first build a set of routines that will allow us to represent facts, called *assertions*. For instance, we can define the following assertions:

```
(this is an assertion)
(color apple red)
(supports table block1)
```

Here each assertion is represented as a list. The set of assertions can be maintained in a database by representing them in a list. For instance, the following list represents an assertion database containing the above assertions:

```
((this is an assertion) (color apple red) (supports table block1))
```

Patterns are like assertions, except that they may contain certain special atoms not allowed in assertions, the single characters ? and !, for instance.

```
(this ! assertion)
(color ? red)
```

Write a function `match` that compares a pattern and an assertion. When a pattern containing no special atoms is compared to an assertion, the two match only if they are exactly the same, with each corresponding position occupied by the same atom.

```
> (match '(color apple red) '(color apple red))
T
> (match '(color apple red) '(color apple green))
NIL
```

The special atom '?' matches any single atom.

```
> (match '(color apple ?) '(color apple red))
T
> (match '(color ? red) '(color apple red))
T
> (match '(color ? red) '(color apple green))
NIL
```

In the last example, `(color ? red)` and `(color apple green)` do not match because `red` and `green` do not match.

The special symbol '!' expands the capability of match by matching any one or more atoms.

```
> (match '(! table !) '(this table supports a block))
T
```

Here, the first '!' symbol matches `this`, `table` matches `table`, and the second '!' symbol matches `supports a block`.

```
> (match '(this table !) '(this table supports a block))
T
> (match '(! brown) '(green red brown yellow))
NIL
```

In the last example, the special symbol '!' matches 'green red'. However, the match fails because `yellow` occurs in the assertion after `brown`, whereas it does not occur in the assertion. However, the following example succeeds:

```
> (match '(! brown) '(green red brown brown))
T
```

In this example, '!' matches the list `(green red brown)`, whereas `brown` matches the last element.

Part 3: Pattern-Matching Variables

We will now extend function `match` so that certain pattern atoms get values if a match is successful. We will replace symbols '?' and '!' with *pattern variables*. A pattern variable is written as either `(? v)` or `(! v)`. The pattern variable `(? v)` is bound to an atomic value while the pattern variable `(! v)` is bound to a list of values.

Define a function `match-var` that extends `match`. The function `match-var` takes patterns containing pattern variables and matches them against assertions and assigns values to variables.

```
> (match-var '(plus (? a) (? b)) '(plus 2 3))
T
> a
2
> b
3
```

In this example, the pattern variable `(? a)` matches `2`. This results in assigning `2` to a variable called `a`. Similarly `b` is assigned `3`.

```
> (match-var '((! u) a (? v)) '(b c a d))
T
> u
(b c)
> v
d
```

Here, the pattern variable `(! u)` matches `b c`. Hence, `u` is assigned a value `(b c)`.

Notes

- The command to use Common LISP is `clisp`.
- Appendix A of LISPcraft summarizes LISP's built-in functions. Each function is explained briefly. You will find this a very useful reference as you write and debug your programs. Also, you can get help about clisp by typing:

  ```
  man clisp
  ```

- The test program will be provided. It exercises the functions that you write; hence there is no test data. If "testl.l" is the name of the test file in your directory, then, within LISP, you need only type

  ```
  > (load "test.l")
  ```

  Details regarding the test program will be posted on the class web page.

- You may define additional helper functions that your main functions use. Be sure, though, to name the main functions as specified since the test program uses those names.
- If you place a `init.lsp` file in the directory in which you execute LISP (or your home directory), LISP will load that file automatically when it starts execution. Such a file is useful to define your own environment. For instance, you will probably want to put the command

  ```
  (setq *print-case* :downcase)
  ```

  in that file.

- When developing your program, you might find it easier to test your functions first interactively before using the test program. You might find `trace`, `step`, `print`, etc. functions useful in debugging your functions.
- You must develop your program in parts as outlined above. Grading will be divided as follows:

Part	Percentage
1	35
2	30
3	35

If your program does not fully work, hand in a listing of the last working part along with your attempt at the next part, and indicate clearly which is which. No credit will be given if the last working part is not turned in. Points will be deducted for not following instructions, such as the above.

- A message giving exact details of what to turn in, where the provided test files are, etc, will be posted to the newsgroup and the class web page.
- A few points to help the novice LISP programmer:
  - Watch your use of (,), ", and '. Be sure to quote things that need to be quoted.
  - To see how lisp reads your function, use pretty printing. For example,

    ```
    > (pprint (symbol-function foo))
    ```

    It will print out the definition of the function `foo`, using indentation to show nesting. This is useful to locate logically incorrect nesting due to, e.g., wrong parenthesizing.
  - If you cause an error, Common Lisp places you into a mode in which debugging can be performed (LISPcraft section 11.2). To exit any level, except the top level, type `:q`. To exit the top level, type

    ```
    > (bye)
    ```

- **Get started now to avoid the last minute rush.**