ECS 140A Programming Languages

Winter 2016

Homework #4

Due 11:59pm Thursday February 25th

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 second part require you to build a pattern-matching program. The third part is related to operations on matrices - addition, multiplication, and transpose.

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. Define a function `cycle` that takes a list and an integer `n` as input and returns the same list, but with the first element cycled to the end of the list `n` times.

```
> (cycle 1 '(1 2 3 4))
(2 3 4 1)

> (cycle 2 '(1 2 3 4 5)))
(3 4 5 1 2)
```

2. Write a function `split-list` that takes a list and an atom and splits the list using the atom as the delimiter. If the array doesn't contain the delimiter, then it returns the entire list as it is.

```
> (split-list '1 '(a b 1 c d 1 e f))
((a b) (c d) (e f))

> (split-list 'a '(b c d e))
((b c d e))
```

3. Define a function `range` that takes a list of numbers (with at least one element) and returns a list of length 3 that consists of the smallest number, the mean (reduced to the simplest fraction) of all numbers and the largest number.

```
> (range '(2 5 11 15 7 1 8))
(1 7 15)

> (range '(6 6 5 -4 3 2 1 1))
(-4 5/2 6)
```

4. Define a function `search-count` that takes an atom and a list and returns the number of occurrences of that atom in the list. Searching in the first level in the list will suffice.

```
> (search-count 'h '(2 d t h 3 h 6))
2

> (search-count 'hi '(good morning hello))
0
```

5. Define a function `pivot` that takes a list `l` and a number `n` and splits it into two lists, one containing all the numbers in `l` less than `n` and the other containing all numbers in `l` greater than or equal to `n`. Note: preserve the relative order of elements inside the list.

```
> (pivot 3 '(3 2 5 1 4)))
((2 1) (3 5 4))

> (pivot 3 nil))
(NIL NIL)
```

6. Write a `quicksort` function that sorts a list. (Review of the quicksort algorithm: First pick an element and call it the pivot. The head of the list is an easy option for pivot. Partition the rest of the list into two sublists, one with all the elements less than the pivot and the other with all the elements not less than the pivot. Recursively sort the sublists. Combine the two sublists and the pivot into a final sorted list). You can reuse the idea of a pivot from the previous question.

```
> (quicksort '(2 9 5 3 8))
(2 3 5 8 9)
```

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: Matrix Operations

Suppose we represent a matrix in LISP as a list of lists. For example, ((a b) (c d)) would represent a 2*2 matrix whose first row contains the elements a and b, and whose second row contains the elements a and d. You may assume that the matrices are well-formed and compatible.

1. Write a function `matrix-add` that takes two matrices as input and outputs the sum of the two matrices.

   ```
   > (matrix-add '((1 2) (2 1)) '((1 2) (3 4)))
   ((2 4) (5 5))
   ```

2. Write a function `matrix-multiply` that takes two matrices as input and multiplies them and outputs the resultant.

   ```
   > (matrix-multiply ((1 2) (2 1)) '((3 1) (1 3)))
   ((5 7) (7 5))
   ```

3. Write a function `matrix-transpose` that takes a matrix as input, and outputs its transpose.

   ```
   > (matrix-transpose '((1 2 3) (4 5 6)))
   ((1 4) (2 5) (3 6))
   ```

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 can be found in the attached materials for this assignment.
- The test program may be executed by calling ./test.sh.
- 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	35
3	30

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 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.**