CMSC 421, Spring 2010: Project 1

Near-final draft - last updated February 18, 2010

- Due date/time: Noon on Tuesday, March 2.
- Late date/time (5-point penalty): Noon on Thursday, March 4.

The main purpose of this project is to get you acquainted with Lisp. It will only count for 50 points (subsequent projects will be 100 points each).

Implement the Lisp functions described below. You can assume that they will never be called with incorrect arguments. For example, in Problem 8, the argument of the `fib` function will always be a positive integer.

Comment your code, indent it appropriately, and use good programming style (e.g., as in Norvig's *Tutorial on Good Lisp Programming Style*).

Don't use any operators that make destructive modifications to their arguments (e.g., `NCONC`, `RPLACA`, `DELETE`, `SETF`), and don't use the `GO` operator.

I don't intend to give you test data (other than what's already provided in the examples below), but it should be pretty easy for you to write your own.

1. Write a function `(infix-eval` s`)` that takes a single argument s, and evaluates s as an expression in infix notation instead of prefix notation. In s, every expression or subexpression will always be either an atom or a list of length 3 in which the 2nd element is a binary operator and the other two elements are the operator's arguments. For example:

```
(infix-eval '(2.0 expt 3))                         ⟹  8.0
(infix-eval '((2 * 6.5) >= ((5 + 8) - 2)))         ⟹  T
(infix-eval 33.2)                                  ⟹  33.2

(defun foo (x y)
  (expt (abs x) (abs y)))
(infix-eval '(-2 foo -3))                          ⟹  8

(defun andp (x y)
  (and x y))
(infix-eval '((3 > 2) andp (2 > 3)))               ⟹  NIL
```

NOTE: In each of the problems below, you might want to write a helper function to compute the distance between a pair of n-dimensional points. If you do this, your helper function is subject the same restriction as your main function (e.g., no iteration or mapping in problem 2, no recursion or mapping or `do` operators in problem 3, etc.).

2. Write a function `(pathlength-r` p`)` that computes the length of an n-dimensional Euclidean path p, where p is represented in the following form:

$$((x_{11}\ x_{12}\ \ldots\ x_{1n})\ (x_{11}\ x_{12}\ \ldots\ x_{1n})\ \ldots\ (x_{11}\ x_{12}\ \ldots\ x_{1n}))$$

For example:

```
(pathlength-r '((2 0 5) (2 1 6)))  ⟹  1.4142135
(pathlength-r '((1 2 3 4 5 6)))    ⟹  0.0
(pathlength-r '((1.0 2.0) (1 2) (1 4) (0 4)))  ⟹  3.0
(pathlength-r '((0 0) (0 1) (0 2) (0 1) (0 0) (0 1))) ⟹  5.0
```

Don't use any iteration or mapping operators; use recursion instead.

3. Write a function `(pathlength-d` p`)` that returns the same result as `pathlength-r`. This time, use the `do` operator; don't use `loop`, mapping operators or recursion.

4. Write a function `(pathlength-l` p`)` that returns the same result as `pathlength-r`. This time, use the `loop` operator; don't use recursion or any of the `do` operators or mapping operators.

5. Write a function `(pathlength-m` p`)` that returns the same result as `pathlength-r`. This time, use one of the mapping operators (e.g., `mapcar` or `maplist`), rather than recursion or iteration.

HINT: below are two ways to do the problem.

- Probably the easiest approach is to write a helper function that takes two arguments that are points, and computes the distance between them. Then you could use `mapcar` to map your helper function over two lists at once, namely p and `cdr` p.
- Another way to do it is to use `maplist` to map a *unary* helper function over the list p. The helper function's argument would be a list of points, and it would compute the distance between the 1st and 2nd points in the list.

6. Write a predicate `(lexical<` x y`)` that takes any two arguments x and y, converts them to character strings, and invokes `string<` on them. For example:

```
(lexical< "abc" "abd")    ⟹  2
(lexical< "abd" "abc")    ⟹  NIL
(lexical< 2.0 "abc")      ⟹  0
(lexical< #(1 2 3) #(1 2 3.0))   ⟹  7
(lexical< "ABC" 'ABC)     ⟹  NIL
(lexical< 'abc "ABC")     ⟹  NIL
(lexical< 'abc "abc")     ⟹  0
```

HINT: Lisp's `string` and `coerce` functions won't do what you want, because many expressions cannot be coerced to strings. But there's a very easy way to do it using `format`.

7. Write a function `(pow b n)` that computes b^n, where n is a nonnegative integer. Do not use any of Lisp's built-in exponentiation functions. The time complexity of your function should be $O(\lg n)$.

HINT: think about the following recursive formula:

$$b^n = \begin{cases} (b^{n/2})(b^{n/2}), & \text{if } n \text{ is even,} \\ b(b^{(n-1)/2})(b^{(n-1)/2}), & \text{if } n \text{ is odd.} \end{cases}$$

8. Write a function `(fib` n`)` that computes the n'th Fibonacci number, F_n. I've seen conflicting descriptions of whether the Fibonacci sequence starts with 0 or 1; you should use $F_1 = 1$, $F_2 = 1$, and $F_i = F_{i-1} + F_{i-2}$ for $i \geq 3$.

Your function should run in linear time rather (not in exponential time like the example I used in my slides). Don't use any of Lisp's looping, iterating, or mapping functions; you need to use recursion instead.

HINT: consider building a helper function that recursively computes F_{n-1} and F_n and returns a list containing both of them.

9. Write a function `(ids` s_0 *goalp* *children*`)` that does an iterative-deepening search, where s_0 is the initial state, *goalp* is the goal predicate (a function of one argument), and *children* is a function of one argument that returns a list of a state's children. `ids` should return a list of two values: a path from s_0 to a state that satisfies *goalp*, and the total number of times *children* was called. See the example on the next page.

```
;;; Function for creating goal predicates
;;; Returns a predicate that's satisfied when its arg is X
;;; e.g., (funcall (is 3) y)  ==> T if y=3, else NIL
(defun is (x)
  (lambda (y) (eql y x)))

;;; Function to generate children: returns 2x and 2x+1
(defun successors (x)
  (list (* 2 x)
        (+ (* 2 x) 1)))

;;; Verbose version of SUCCESSORS
(defun successors-verbose (x)
  (format t "~%(successors ~s) ==> " x)
  (prin1 (successors x)))
```

```
(ids 1 (is 1) #'successors)  ⟹  ((1) 0)
(ids 1 (is 3) #'successors)  ⟹  ((1 3) 1)
(ids 1 (is 6) #'successors)  ⟹  ((1 3 6) 4)
(ids 1 (is 9) #'successors)  ⟹  ((1 2 5 10) 8)
```

```
(ids 1 (is 12) #'successors-verbose)
```

⟹ *prints out the following lines*

```
(successors 1) ==> (2 3)
(successors 1) ==> (2 3)
(successors 2) ==> (4 5)
(successors 3) ==> (6 7)
(successors 1) ==> (2 3)
(successors 2) ==> (4 5)
(successors 4) ==> (8 9)
(successors 5) ==> (10 11)
(successors 3) ==> (6 7)
(successors 6) ==> (12 13)
```

⟹ *and then returns* `((1 3 6 12) 10)`