

CMSC 421, Fall 2008: Project 1

50 points

- Due date/time: noon on October 9.
- Late date/time (5-point penalty): noon on October 11.

This purpose of this project is to get you acquainted with Lisp. It will only count for 50 points (Projects 2 and 3 will be 100 points each).

Implement the Lisp functions described below. You can assume that the functions will never be called with incorrect arguments: for example, the arguments to `subsets3-looping` will always be sets that are represented as lists.

Use good programming style, and comment your code. Except where I say otherwise, don't use operators (e.g., `NCONC`, `RPLACA`, `DELETE`, `SETF`) that destructively modify their arguments, and don't use the `GO` operator.

I don't intend to give you test data for this project, because it should be pretty easy for you to write your own.

1. Write a function `(subsets3-looping s)` that takes a set s as an argument, and returns the set of all subsets of s of size 3. You may use any of the looping functions that you want (`do`, `do*`, `dolist`, `dotimes`, `loop`), but don't use the mapping functions and don't use recursion.

Notes:

- If s contains fewer than 3 elements then it has no subsets of size 3. In such a case, `subset3-looping` should return `NIL`.
- Since we're talking about sets, the ordering of the elements doesn't matter. For example, if I haven't made a typing error, it would be OK for `(subsets3-looping '(1 2 3 4))` to return any of the following lists (as well as many others):

```
((1 2 3) (1 2 4) (1 3 4) (2 3 4))
((1 2 3) (1 2 4) (2 3 4) (1 3 4))
((1 2 3) (2 3 4) (1 2 4) (1 4 3))
((1 2 3) (1 2 4) (1 3 4) (4 3 2))
((2 3 1) (1 2 4) (1 3 4) (2 3 4))
((3 2 1) (1 2 4) (2 4 3) (1 3 4))
```

2. Write a function `(subsets3-mapping s)` that does the same thing as `(subsets3-looping s)`, with one difference: you may use the mapping functions (`mapcar`, `mapcan`, `maplist`), but don't use the looping functions or recursion.

3. Write a function `(subsets3-recursive s)` that does the same thing as `(subsets3-looping s)`, with one difference: you may use recursion, but don't use the looping functions or the mapping functions.

4. Write a function `(subsets s n)` that returns all subsets of s of size n . You may use any combination of recursion, mapping functions, and looping functions that you wish.

5. Write a function `(merge-sort sequence predicate num1 num2)`, which does a merge-sort of a subsequence of $sequence$ using $predicate$ as the comparison predicate. The subsequence to be sorted is the part that starts at position $num1$ and ends just before position $num2$, where the positions are indexed starting at 0.

You may assume that $predicate$ will take two arguments, and that it will return `non-nil` if and only if the first argument is strictly less than the second (in some appropriate sense).

`merge-sort` should take an optional keyword argument `:key` that does the same kind of thing that it does in the built-in Lisp functions.

For this function only, you should use the `setf` operator to do destructive operations on the argument s . (Otherwise, your function would be awkward to implement and would create lots of garbage.)

6. Write a function `(mathprint expr)` that prints $expr$ in mathematical format instead of Lisp format, e.g., $f(x_1, x_2, \dots, x_n)$ instead of `(f x1 ... xn)`. For example,

```
(mathprint '(g (concat a b) (plus 2 3 (f 4 "init") 5)))
```

should print

```
g(concat(a, b), plus(2, 3, f(4, "init"), 5))
```

Notice that there are blanks after the commas but no blanks elsewhere.

`mathprint` should return `nil`.