EXERCISES 01-29

This handout presents a series of exercises for CS 352. Each will be assigned when appropriate.

In general, each requires the following deliverables:

- a printout of the code file — document as appropriate
- hardcopy of assigned & appropriate test runs

Problems may also specify other or further requirements.

WARNING — for the moment

Do not use global variables unless they are truly required.
(Don't use `setf` inside functions.)

Do not use local variables. (Don't use `let` or `do`.)

Do not use `if`. Use `cond` instead.

Do not use nested `if`s. Use `cond` instead.

Regarding the Prolog exercises: I may have inadvertently selected function names which are reserved words. You can recover from my error by slightly changing the name.

Exercise 1.

Create a file containing the `QUAD1` function from class:

```
(defun QUAD1 (A B C)
  (/ (+ (- B) (sqrt (- (* B B) (* 4 A C))))
     (* 2 A))
)
```

Use the following as test cases:

```
(QUAD1 1 2 1)

(QUAD1 1 -2 1)

(QUAD1 1 -2 -3)
```

Exercise 2.

Write the Common Lisp function `QUAD2` which takes in the A, B, C of the quadratic formula and returns a *list* of the two roots given by the standard formula.

Use the following as test cases:

```
(QUAD2 1 2 -8) → (2 -4)
(QUAD2 1 -2 1) → (1 1)
```

Exercise 3.

Write a Common Lisp function `QUAD3` which takes in the A, B, C of a quadratic equation and returns a list of all its real roots.

Use the following as test cases:

```
(QUAD3 1 -1 -6) → (3 -2)
(QUAD3 1 2 1) → (-1)
(QUAD3 5 1 3) → nil
(QUAD3 0 4 5) → (-1.25)[1]
(QUAD3 0 0 3) → nil[2]
(QUAD3 0 0 0) → nil[3]
```

Assume that all the arguments are numeric.

Note that the superscripts (1, 2, 3) are references to footnotes. They are not meant to be part of your output.

Document any helper functions that you use.

---

[1] Note that this set of coefficients corresponds to the *linear* equation 4X+5=0, which has solution `X=-1.25`. Hence, `QUAD3` returns `(-1.25)`

[2] Note that this set of coefficients corresponds to the impossible equation 3=0. Clearly, no value of X can make this equation true, so `QUAD3` returns nil, a list of the values of X which make the equation true.

[3] Note that this set of coefficients corresponds to the tautology 0=0. Technically, this means that X is indeterminate, since *any* value of X makes the tautology true. Let's just have `QUAD3` return `NIL` in this case.

Exercise 4.

Consider polynomials in the variable `x`.

Write and demonstrate the function `write-term`, which takes the coefficient and exponent of a term and writes it out appropriately. For example,

```
(write-term –3 5)
```

results in the printing (side-effect) of

```
   5
-3x
```

Obviously, you need to print this with a fixed-width font.

There may also be a *result* from the function `write-term`, but that's ok.

Test your function using the driver supplied by the instructor.

Turn in:

- printout of your code file
- hardcopy of the test run

Exercise 4a.

We can represent a polynomial in **`X`** by giving a list of its terms in the form **`(`*`coef exp`*`)`**.

For example, $3x^2 - 2x + 5$ would be represented as **`((3 2)(-2 1)(5 0))`**.

Polynomial Normal Form — PNF — obeys the following rules:

(1) Terms are sorted from lowest to highest exponent. For example, we want **`((5 0)(-2 1)(3 2))`** instead of **`((3 2)(-2 1)(5 0))`**.

(2) Terms with **`0`** coefficient are omitted.
We want **`((1 2))`** instead of **`((0 1)(1 2))`**.
We want **`nil`** instead of **`((0 1)(0 2))`**.

(3) Only one term occurs for any exponent.
We want **`((5 1))`** instead of **`((2 1)(3 1))`**.

Write a function **`PNF`** which takes a polynomial and returns its representation in Polynomial Normal Form.

Test your code by running the driver given on the website.

Exercise 5.

Consider polynomials in one variable, x. These will be lists of the monomials we explored in an earlier exercise. Thus, if (3 2) represents the term $3x^2$ and `(-1 0)` represents the term $-1x^0 = -1$, then the list ((3 2)(-1 0)) stands for the polynomial $3x^2 - 1$.

Here are some examples:

math notation	internal representation	printed output form
0	`nil`	`0`
0	`((0 0))` or `((0 1))` or …	`0`
$2x+1$	`((2 1)(1 0))`	`2 x + 1`

$3x^2 - 1$	`((3 2)(-1 0))`	`      2`
		`3 x  - 1`

$5x^2 - 4x + 1$	`((5 2)(-4 1)(1 0))`	`      2`
		`5 x  – 4 x + 1`

$7x^{14} + 9x^3 - 3x^2 + 7x$	`((7 14)(9 3)(-3 2)(7 1))`	`    14       3       2`
		`7 x    + 9 x   - 3 x   + 7`

Write a Common Lisp function `write-poly`, which takes an internal representation of a polynomial and produces the appropriate printed output form.

The following Lisp functions may be useful: `format`, `reverse`, …

Beyond general instructions, turn in hardcopy of at least the test cases shown above.

Note: In a future exercise, we may continue using polynomials. We haven't yet examined canonical forms, simplification, or arithmetic.

Put my test cases first in your printout.

Exercise 6.

Write the recursive function `AL1` which takes two lists of integers and returns a list of integers in which each element is the sum of the corresponding elements of the incoming lists. For example,

```
(AL1 ’(3 1 4) ’(8 2 5)) → (11 3 9)
```

The lists need not be equal in length; the shorter list should be treated as though it were padded out with zeros at the end. For example,

```
(AL1 ’(3 1 4) ’(2 2)) → (5 3 4)

(AL1 ’(3 1 4) nil) → (3 1 4)
```

Exercise 7.

Write the recursive function `INTERSECT` which takes two *sets* of integers

and returns the intersection of the two sets. By *set*, I mean a list which contains no duplicates. For example,

```
(intersect ’(4 1 2) ’(2 5 3 1 7)) → (1 2)

(intersect ’(4 1 2) nil) → nil
```

Exercise 8.

Write a recursive function `FIBP` which takes a list of integers and determines whether or not the numbers in the list have the Fibonacci property:

> L_1 is arbitrary.
>
> L_2 is arbitrary.
>
> $L_N = L_{N-1} + L_{N-2}$.

`FIBP` should return nil only if its argument violates the Fibonacci property. Thus, any list of length 0 or 1 or 2 possesses the Fibonacci property.

For example,

```
(fibp nil) → t
(fibp ’(7)) → t
(fibp ’(7 3)) → t
(fibp ’(1 1 2 3 5 8)) → t
(fibp ’(7 3 10 13 23)) → t
(fibp ’(1 2 3 4 5)) → nil
```

Exercise 9.

Write a recursive function `SETIFY` which takes a list and converts it into a set. For example,

```
(setify nil) → nil
(setify ’(18)) → (18)
(setify ’(3 1 4 1)) → (3 4 1)
(setify ’(a b b c a b b)) → (c a b)
```

Exercise 10.

Write a recursive function `COUNT#` which takes an arbitrary S-expression and returns the number of numbers which are found in the expression, at any depth or nesting. For example,

```
(count# nil) → 0
(count# 3) → 1
(count# 'a) → 0
(count# '(5 1 a 4)) → 3
(count# '((1 1 (a)) 7 (b (3)))) → 4
```

If the argument is non-atomic, then your function will need to examine both the `car` and `cdr` of the argument.

Exercise 11.

We have seen that every iteration can be written as a tail recursion. Use tail recursion to implement a function `NONE` (that's N-O-N-E) which takes a list of integers and returns a 2-element list: the first element is the number of odd elements and the second element is the number of even elements in the argument. For example,

```
(none nil) → (0 0)
(none '(3)) → (1 0)
(none '(2)) → (0 1)
(none '(3 1 4 1 5)) → (4 1)
```

Draw the flowchart for the iterative function, and show me where you've broken it into a main function and a helper function.

Exercise 12.

Write a Lisp function `TABLE` which takes two arguments, `LO` and `HI`, and prints out a three-column table with the following properties:

- column 1 contains the integers `LO` through `HI`;
- column 2 contains the square of the integer in column 1;
- column 3 contains the cube of the integer in column 1.

Exercise 13.

Consider the function **`NONE`** which you wrote recursively in Exercise 6.

Write a non-recursive version of **`NONE`** which uses `do`.

Exercise 14.

Write a Lisp macro `zero` which takes a variable name and sets that variable to be zero.

For example,

```
x → error:  variable X has no value
(zero x)
x → 0
(setq y 5)
y → 5
(zero y)
y → 0
```

Before giving me the code, you must show me some examples of input code and the code into which it is transformed. See Exercise 16 for an example.

Exercise 15.

Write a Lisp macro `zero*` which takes *zero or more* variable names and sets those variables to zero.

For example,

```
x → error:  variable X has no value
(zero* x)
x → 0
(setq x 3)
(setq y 5)
x → 3
y → 5
(zero* x y)
x → 0
y → 0
```

Before giving me the code, you must show me some examples of input code and the code into which it is transformed. See Exercise 16 for an example.

Exercise 15a.

Write a Lisp macro **`INC*`** which takes *zero or more* variable names and increments those variables by **`1`**.

(1) Before giving me the code, you must show me some hand examples of input code and the code into which it is transformed.

(2) Give the code for your macro.

(3) Prove that your macro works by using **`macroexpand-1`** on each of your i/o examples.

Exercise 16.

Suppose we want to create a `for` macro in Lisp with the syntax

```
    (for <var> <lo> <hi> <exp1> ... <expN>)
```

A typical use might be

```
    > (for I 1 4 (format t "~%~s ~s" I (* I I)))
    1 1
    2 4
    3 9
    4 16
    nil
```

A `for` expression always returns `NIL`, although it may have many side effects.

Note that a `for` expression may contain any number of statements in its body.

Show the code you want produced when each of the following `for` expressions is macroexpanded:

1. `(for I 1 4 (print I))`
2. `(for I 1 100 (print I) (setq X I) (bar))`

Write the Lisp code which implements the `for` macro.

Test your code on the following examples:

```
    (for I 1 4 (print I))
    (for X 2 7 (print X) (print (* X X)))
    (for I 1 10 (format t "~%~s ~s" I (* I I)))
    (defun SUM (N)
      (let ((OUT 0))
        (for J 1 N (setq OUT (+ OUT J)))
        OUT
    ))
    (SUM 100)
```

Turn in:

your written answers to questions 1 and 2 above;

hardcopy of your code working on the examples described above;

hardcopy of your code.

Exercise 17.

Write a Lisp function `op-some` which uses functional arguments to implement the `op-some` schema. The form of a call to `op-some` is this:

`(op-some` *condition operation arg*`)`.

For example, we might use `op-some` to implement the function `sq-odd`, which takes a list of integers and returns another list in which only the odd numbers have been *squared*:

```
(op-some #'oddp #'sq nil) → nil

(op-some #'oddp #'sq '(2)) → (2)

(op-some #'oddp #'sq '(3)) → (9)

(op-some #'oddp #'sq '(3 6 4 5 2)) → (9 6 4 25 2)
```

Of course, we will have to write the function `sq` which takes a number and returns its square.

Exercise 18.

In this exercise, we will write a small helper function which might be used by an expert system to match patterns against facts. The function is called `MPF`, which stands for Match Pattern & Fact.

Here are some definitions you need:

- A **variable** is a symbol whose print-name begins with a question mark ("`?`"). For example: `?X`, `?AMT`, `?2`. You should write a predicate `varp` which tells if something is a variable.
- A **constant** is any other atom.
- A **fact** is a list of constants.
- A **pattern** is a list of constants and variables.

MPF takes two arguments, a pattern and a fact, both of which are lists. The function begins like this:

```
(defun MPF (P F) ...)
```

It returns one of three things:

(1) `NIL` means there was *no match* between the pattern and the fact. For example,

```
(mpf '(mammal ?x) '(john is a student)) → nil
(mpf '(mammal ?x) '(mammal rodent octopus)) → nil
(mpf '(mammal ?x) '(eats-meat leopard)) → nil
(mpf '(a b c) '(a e c)) → nil
```

(2) A list of pairs, meaning that the fact *matches* the pattern with the indicated correspondence of variables & values. Each pair is a two-element list: the first element is a variable; the second is its value (the thing it matches in the fact). For example,

```
(mpf '(mammal ?x) (mammal bozo)) → ((?x bozo))
(mpf '(?x is a ?y) '(john is a student))
                        → ((?x john) (?y student))
(mpf '(?x ?y) '(a b)) → ((?x a) (?y b))
(mpf '(?x ?x) '(a a)) → ((?x a))
```

(3) `T` means there *was* a match between the pattern and the fact, but *no variables were involved.*

```
(mpf '(a b c) '(a b c)) → t
(mpf '(method 1) '(method 1)) → t
```

Write the function `MPF` and any helper functions you require.

Document any helper functions you write: use a comment to tell what the function does, how it does it, and give some examples. You will not receive full credit if I can't understand your helper functions.

Put all your code into a single file. Put your name in a comment at the beginning of the file.

Test the completed function on the following test cases:

```
(MPF '(MAMMAL ?X) '(MAMMAL BOZO))
(MPF '(MAMMAL ?X) '(JOHN OWES MARY 10))
(MPF '(MAMMAL ?X) '(BIRD TWEETY))
(MPF '(COLOR ?X BROWN) '(COLOR SPOT BROWN))
(MPF '(COLOR ?X BROWN) '(COLOR FIDO RED))
(MPF '(?X IS A ?Y) '(JOHN IS A STUDENT))
(MPF '(A) '(A))
(MPF '(A) '(B))
(MPF '(A) '(MAMMAL BOZO))
(MPF '(A B C) '(A B C))
(MPF '(LOVES ?X ?Y) '(LOVES JOHN PHYSICS))
(MPF '(LOVES ?X ?Y) '(LOVES JOHN JOHN))
(MPF '(LOVES ?X ?Y) '(LOVES JOHN NIL))
(MPF '(?X OWES ?Y ?AMT) '(JOHN OWES MARY 10))
(MPF '(?X OWES ?Y ?AMT) '(MAMMAL BOZO))
(MPF '(?X ?X) '(A A))
(MPF '(?X ?X) '(A B))
(MPF '(?X ?Y ?X) '(A B A))
(MPF '(?X ?Y ?X) '(A B C))
```

I may have sent you an email containing these test cases.

Turn in:

a printout of the file containing your function definitions;

a printout showing its performance on the required test cases.

Exercise 19.

Write the Prolog code which computes the `member` function. For example:

```
member(a,[b,a,c]). → yes
member(a,[b,b,c]). → no
member(a,[]). → no
```

Does it work for the following?

```
member(X,[a,b,c]).
```

Exercise 20.

Write the Prolog code which determines if a list consists entirely of atoms.

For example, if the predicate were called aa (for "all atoms"), we might expect behavior like this:

```
aa([]) → yes
aa([a,b,c]) → yes
aa([a,2,c]) → yes
aa([a,[b],c]) → no
```

Exercise 21.

Write the Prolog code which returns the last element of a list.

For example:

```
last(X,[a,b,c]). → X = c
last(X,[a,[b,c]]). → X = [b,c]
```

Exercise 22.

Write a Prolog function `inc/2` which computes or verifies that two lists are related because the elements of the second list are all incremented by one from the elements of the first list. For example:

```
inc([1,7,5],[2,8,6]). → yes
inc([1,7,5],X). → X = [2,8,6]
inc([],X). → X = []
```

Can it solve the following?

```
inc(X,[2,8,6]).
```

Exercise 23.

Write the Prolog code which generates the squares of the natural numbers. For example:

```
sq(X). → X=0; X=1; X=4; X=9; X=16; ...
```

Exercise 24.

Write the Prolog code which computes factorials. For example:

```
fac(0,N). → N=1
fac(5,N). → N=120
```

What does it do in the following cases?

```
fac(0,1).
fac(0,5).
fac(4,18).
fac(X,1).
fac(X,6).
```

Exercise 25.

Write the Prolog code which adds two equal-length lists, element by element. For example:

```
addl([],[],L). → L=[]
addl([3,1,4],[2,7,1],L). → L=[5,8,5]
```

Can it solve the following problems?

```
addl(X,[2,7,1],[5,8,5]).
addl([3,1,4],X,[5,8,5]).
```

Exercise 26.

Write the Prolog code which computes the `mem1` function, which determines whether a given atom occurs *at any level* of a given expression. For example:

```
mem1(a,[b,a,c]) → yes
mem1(a,[b,[a],c]) → yes
mem1(a,[d,e]) → no
```

Exercise 27.

Write `sose/3` which computes the sum of the odd and even elements in a list of integers. For example:

```
sose([3,1,4,6,3,2],SO,SE) → SO=7 , SE=12
```

Exercise 28.

Write `rev/2` which reverses a list. For example:

```
rev([a,b,c],X) → X=[c,b,a]
rev([],X) → X=[]
```

Does your function work in the other direction?

```
rev(X,[a,b,c]) → X=[c,b,a]
rev(X,[]) → X=[]
```

Exercise 29.

Consider the inverted pyramid below.

Write the Prolog code which places the numbers 1 through 10 into the blanks such that any number is the absolute value of the difference of the two numbers “directly” above it.

For example,

```
1     9
   8
```

or

```
7     5
   2
```

are legal *sub*-pyramids because $8 = |1 - 9|$ and $2 = |7 - 5|$, but

```
1     9
   5
```

is not, because $5 \neq |1 - 9|$.

Remember, each of the numbers {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} should occur once and only once in the pyramid.

```
______    ______    ______    ______

     ______    ______    ______

          ______    ______

               ______
```