# Functional Programming Problems

*Colin Runciman*
*Department of Computer Science*
*University of York*

*October 1987 (revised September 1989)*

**Introduction**

Each of the problems given below is graded on the following scale so that you have some idea how long it can be expected to take, and what level of skill it demands.

Grade I — The exercise is elementary and may only take a few minutes. Perhaps there is a solution by straightforward adaptation of an example you have seen before.

Grade II — Allow half an hour for the exercise. It is intended to stretch understanding and requires something a little more than straightforward application of previously illustrated technique.

Grade III — An exercise with this grade could easily take an hour or more. There are probably lots of ways to do it, and discovering some of them may require careful thought. So it's the kind of problem that can be looked at again and again.

Inevitably there are all sorts of hidden assumptions behind such a grading system. In some cases it is not easy to assign an appropriate grade to an exercise. So please don't be insulted if you find what seems an obvious solution to a Grade III problem in two minutes flat (but do read the description of Grade III again), and don't be unduly alarmed if you end up wrestling with a Grade II problem for half a day (but you're probably making it too complicated).

0 Define a function `sum` that expects as its single argument a list of numbers and yields as result their sum.

```
sum [1, 2, 3] ⇒ 6
```

(Grade I)

1 Define a function `append x y` whose result is a list containing exactly the items of list `x` followed by those of list `y`.

```
append ["To", "be", "or"] ["not", "to", "be"]
       ⇒ ["To", "be", "or", "not", "to", "be"]
```

Notice that since there are *two* list arguments there are at least two possible ways of approaching the list recursion.

(Grade I)

2 Define the function `take n x` so that its result is the first `n` items of the list `x`.

```
take 2 ["To", "be", "or", "not", "to", "be"]  ⇒  ["To", "be"]
```

If `x` has less than `n` items the result should be the whole of `x`.

(Grade I)

3 Define the function `drop n x` so that its result is *all but* the first `n` items of the list `x`.

```
drop 4 ["To", "be", "or", "not", "to", "be"]  ⇒  ["to", "be"]
```

If `x` has less than `n` items the result should be `[]`.

(Grade I)

4 Define a function `contains x i`. It should expect `x` to be a list and give the result `True` if `i` is one of the items in `x` but `False` if it is not.

```
contains ["To", "be", "or", "not", "to", "be"] "that" ⇒ False
```

(Grade I)

5 Define a function `set x` to have result `True` if the list `x` has no duplicate items and `False` otherwise.

```
set ["To", "be", "or", "not", "to", "be"] ⇒ False
```

(Grade I)

6 Define a function `positions x i` whose result is the list of numeric indices at which the item `i` occurs in the list `x`.

```
positions ["To", "be", "or", "not", "to", "be"] "be" ⇒ [2, 6]
```

Expect to need an explicitly defined auxiliary.

(Grade II)

7 Define a function `duplicates x` expecting `x` to be a list of data values. The result of `duplicates` should be a list of those items that occur more than once in `x`.

```
duplicates ["To", "be", "or", "not", "to", "be"] ⇒ ["be"]
```

No item should occur in the *result* more than once, no matter how often it occurs in the argument.

(Grade II)

8 A pair of lists of equal length can be zipped together to make a list of pairs.

```
zip (["York","Durham","Leeds"] ^ [904,385,532])

  ⇒ ["York"^904, "Durham"^385, "Leeds"^532]
```

Define the function `zip`, and also its inverse `unzip`.

(Grade II)

9 Define a function `prefixes` to compute a list of all the non-empty prefixes of its argument (a list), including the entire argument if this is non-empty.

```
prefixes ["not", "to","be"] ⇒
         [["not"], ["not", "to"], ["not", "to", "be"]]
```

(Grade II)

10 Define a function `suffixes` to compute a list of all the non-empty suffixes of its argument (a list), including the entire argument if this is non-empty.

```
suffixes ["not", "to","be"] ⇒
         [["not", "to", "be"], ["to", "be"], ["be"]]
```

(Grade II)

11 The `powerset` of a set is the set of all its subsets.

```
powerset ["egg", "bacon"] ⇒
         [[], ["egg"], ["bacon"], ["egg", "bacon"]]
```

Define the function `powerset`. The order in which the subsets are listed need not be the same as that shown in the example; different definitions may give different orders.

(Grade II)

12 When there is an ordering relation defined on the items of a list a `sort` can be applied to obtain a new list containing the same items but arranged in non-decreasing order.

```
sort "quick brown fox"  ⇒  "  bcfiknooqruwx"
```

Define the function `sort`. There are many ways to do this. To start with, at least, avoid obviously procedural methods such as *bubblesort* as these only make the problem artificially difficult.

(Grade III)

13 Let the `segments` of a list `x` be all the (non-empty) sequences of items that occur consecutively somewhere within `x`.

```
segments [1, 2, 3] ⇒
         [ [1], [2], [3], [1, 2], [2, 3], [1, 2, 3] ]
```

Define the function `segments`. The segments need not necessarily be arranged in the same order as in the above example. You may wish to use the `prefixes` and `suffixes` functions as auxiliaries.

(Grade III)

14 Define a function `perms` that produces as result a list containing all the permutations of its list argument.

```
perms [0,1,2] ⇒
      [ [0,1,2], [0,2,1], [1,0,2], [1,2,0], [2,0,1], [2,1,0] ]
```

The order in which the permutations appear in the result is not important. Order of items within each permutation is vital, of course.

(Grade III)

15 Define the function `pam fs x` to compute the list of results obtained by applying each of the list of functions `fs` to the argument `x`.

```
pam [min, max, sum, product] [4, 5, 6]  ⇒  [4, 6, 15, 120]
```

Annotate your definition of `pam` with a suitable polymorphic type.

(Grade I)

16 Define the function `mapcat` in such a way that

```
mapcat f x = fold (::) [] (map f x)
```

but do not use `map` or `fold` in your definition. Annotate your definition of `mapcat` with a suitable polymorphic type.

(Grade I)

17 Define the function `allplayall` to take as argument a list of items (perhaps names of players in a sporting event) and yield a tabulated list of all possible pairings of those items.

```
allplayall ["ethel", "edna", "edith"] ⇒
    ethel  v. edna
    ethel  v. edith
    edna   v. edith
```

Note that players do not play themselves, and that pairings are not repeated in the opposite order.

(Grade II)

18 Define a function `sums p` that for a positive numeric argument `p` gives as result all the different addition sums (allowing only positive numbers) whose answer is `p`. The result should be a string that, when printed, presents an orderly list of the sums, one per line.

```
sums 3  ⇒
1 + 1 + 1
1 + 2
2 + 1
3
```

Each sum should be listed exactly once.

(Grade III)

19 Develop a "pretty printer" for N-queens boards. A function `board rs` should expect `rs` to be list of N rank numbers (each in the range 1..N), the ith number giving the rank on which a queen is placed in file i. The result of `board` should be a string which, when printed, depicts an NxN board with the positioning of queens suitably marked.

```
board [2, 4, 1, 3]  ⇒

     .....o o o.....
     ..... \|/ .....
     ..... === .....
          .....     o o o
          .....      \|/
          .....      ===
     o o o     .....
      \|/      .....
      ===      .....
          .....o o o.....
          ..... \|/ .....
          ..... === .....
```

The detailed format of the output is a matter of taste!

(Grade II)

20 Let a triangle of order `n` be a list of length `n` whose `i`th member is a list of length `i`. Define a function `trout` that accepts a triangle as its argument and produces (as a string) a suitably formatted diagram.

```
trout [[0], [1,1], [1,2,1]]  ⇒
  0
 1 1
1 2 1
```

(Grade II)

21 Define a function `trol` that, in terms of the layout produced by `trout` (see above), rotates triangles through 120 degrees anticlockwise.

```
trout (trol [[0], [1,1], [1,2,1]]) ⇒
  1
 1 2
0 1 1
```

Define also `tror`, the inverse of `trol` that rotates triangles clockwise. Of course

```
tror = trol o trol
```

but this is not necessarily the best definition.

(Grade II)

22 One triangle of particular interest is that due to Pascal (the man, not the language!). The first and last element in each row is `1`; any other element can be obtained by adding the pair of elements immediately above-left and above-right. Define the function `pascal n` whose result is the triangle of order `n` constituting the first `n` rows of Pascal's triangle.

```
trout (pascal 5) ⇒
      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
```

(Grade II)

23 Pascal's is really a triangle of infinite order. So formulate a definition for it as an infinite data structure `pas`. In terms of the `pascal` function (see above), the equation

```
take n pas = pascal n
```

should hold for any number `n`.

(Grade II)

24 Develop an interactive program to report the number of syllables in words. Here is an example dialogue, in which *italicised* text is that entered by the user.

```
word: a
1 syllable
word: functional
3 syllables
word: program
2 syllables
word: (eof)
```

It is part of the problem to devise a (rough) rule for the number of syllables in a word. Start with the distinction between vowels and consonants.

(Grade II)

25 To motivate trainee typists a program is required to count the number of words typed by its user and report the running total after each line of input.

```
Half a pound of tuppenny rice;
6
half a pound of treacle;
11
that's the way the money goes:
17
pop goes the weasel!
21
```

A rough rule for the number of words on a line, such as the number of blanks plus one, is adequate.

(Grade II)

26 Prove carefully by list induction that append is associative. That is, prove that the following is true for any lists `a`, `b`, `c`.

```
append a (append b c) == append (append a b) c
```

(Grade I)

27 The following equivalence involving `append` and `reverse` does *not* hold in general. Why not?

```
reverse (append x y) == append (reverse y) (reverse x)
```

(Grade I)

28 Prove that

```
map2 f (map g xs) ys == map2 f xs (map h ys)
```

whenever `f`, `g` and `h` satsify the following law.

```
f (g x) y == f x (h y)
```

(Grade I)

29 Do your definitions of the functions `take` and `drop` (see above), satisfy the following equation?

```
append (take n x) (drop n x) == x
```

Give a proof or a counter-example.

(Grade II)

30 Assuming the following definition of a function `all`

```
all [] -> True
all (h:t) -> h & all t
```

map f [] → []
map f (x:xs) → f x : map f xs

and the usual definition of the function `map`, consider the following definition of `subset`.

```
subset x y -> all ( map ( member y ) x )
```

Apply fold/unfold transformation to obtain another definition of subset involving neither `all` nor `map`.

(Grade II)

31 Consider the problem of defining a function `eval` to evaluate an integer polynomial of the following form.

$$c_0x^n + c_1x^{n-1} + \ldots + c_n$$

Simple recurrence analysis leads to one solution.

```
eval [] x -> 0
eval (c:cs) x -> c * pow x (length cs) + eval cs x
```

But the following generalisation can be obtained by considering how evaluation for an entire coefficient list is related to the evaluations of an arbitrary prefix-suffix pair.

```
eval' cs x e -> eval cs x + e * pow x (length cs)
```

Use tranformation to derive a more efficient `eval'` (and hence `eval`) eliminating all auxiliaries other than primitive arithmetic.

(Grade II)

32 A function `diag`, extracting the diagonal of a matrix represented as a list of lists, could be defined as follows.

```
diag -> map2 item nats

item 0      (x:xs) -> x
item (n+1)  (x:xs) -> item n xs

nats -> 0 : map (1+) nats
```

Use fold/unfold transformation to derive an alternative definition of `diag` with `map` as the sole non-primitive auxiliary. (*Hint:* you will need the law involving `map` and `map2` proved as an earlier exercise.)

(Grade II)

33 Here is a family of three functions on sets, computing set difference (elements in first argument but not in second), set union (elements in one argument or the other) and set intersection (elements in both arguments).

```
diff [] y -> []
diff (h:t) y -> if (member h y) (diff t y) (h : diff t y)

union [] y -> y
union (h:t) y -> if (member h y) (union t y) (h : union t y)

inter [] y -> []
inter (h:t) y -> if (member h y) (h : inter t y) (inter t y)
```

Consider the problem of defining a function to compute the *exclusive union* of its set arguments, that is the set of elements occurring in one argument or the other but *not* in both. An "obviously correct" definition is as follows.

```
excl x y -> diff (union x y) (inter x y)
```

Derive a more efficient definition by transformation.

(Grade III)

—♦—