Extra Practice Problems

Here are some extra programming problems that can be done using the material in this module. Many are similar in difficulty and content to the homework, but they are not the homework, so you are free to discuss solutions, etc. on the discussion forum. Thanks to Pavel Lepin and Charilaos Skiadas for contributing most of these.

0. Consider any of the extra Practice Problems from Section 1 and redo them using pattern matching.

Problems 1-4 use these type definitions:

```
type student_id = int
type grade = int (* must be in 0 to 100 range *)
type final_grade = { id : student_id, grade : grade option }
datatype pass_fail = pass | fail
```

Note that the grade might be absent (presumably because the student unregistered from the course).

1. Write a function `pass_or_fail` of type `{grade : int option, id : 'a} -> pass_fail` that takes a `final_grade` (or, as the type indicates, a more general type) and returns `pass` if the grade field contains `SOME` i for an $i \geq 75$ (else `fail`).
2. Using `pass_or_fail` as a helper function, write a function `has_passed` of type `{grade : int option, id : 'a} -> bool` that returns `true` if and only if the the grade field contains `SOME` i for an $i \geq 75$.
3. Using `has_passed` as a helper function, write a function `number_passed` that takes a list of type `final_grade`(or a more general type) and returns how many list elements have passing (again, ≥ 75) grades.
4. Write a function `number_misgraded` of type `(pass_fail * final_grade) list -> int` that indicates how many list elements are "mislabeled" where mislabeling means a pair `(pass,x)` where `has_passed x` is `false` or `(fail,x)` where `has_passed x` is `true`.

Problems 5-7 use these type definitions:

```
datatype 'a tree = leaf
                 | node of { value : 'a, left : 'a tree, right : 'a tree }
datatype flag = leave_me_alone | prune_me
```

5. Write a function `tree_height` that accepts an `'a tree`and evaluates to a height of this tree. The height of a tree is the length of the longest path to a leaf. Thus the height of a leaf is `0`.

6. Write a function `sum_tree` that takes an `int tree` and evaluates to the sum of all values in the nodes.

7. Write a function `gardener` of type `flag tree -> flag tree` such that its structure is identical to the original tree except all nodes of the input containing `prune_me` are (along with all their descendants) replaced with a leaf.

8. Re-implement various functions provided in the SML standard libraries for lists and options. See http://sml-family.org/Basis/list.html and http://sml-family.org/Basis/option.html. Good examples include `last`, `take`, `drop`, `concat`, `getOpt`, and `join`.

Problems 9-16 use this type definition for natural numbers:

```
datatype nat = ZERO | SUCC of nat
```

A "natural" number is either zero, or the "successor" of a another integer. So for example the number 1 is just `SUCC ZERO`, the number 2 is `SUCC (SUCC ZERO)`, and so on.

9. Write `is_positive : nat -> bool`, which given a "natural number" returns whether that number is positive (i.e. not zero).

10. Write `pred : nat -> nat`, which given a "natural number" returns its predecessor. Since 0 does not have a predecessor in the natural numbers, throw an exception `Negative` (will need to define it first).

11. Write `nat_to_int : nat -> int`, which given a "natural number" returns the corresponding `int`. For example, `nat_to_int (SUCC (SUCC ZERO)) = 2`. (Do not use this function for problems 13-16 -- it makes them too easy.)

12. Write `int_to_nat : int -> nat` which given an integer returns a "natural number" representation for it, or throws a `Negative` exception if the integer was negative. (Again, do not use this function in the next few problems.)

13. Write `add : nat * nat -> nat` to perform addition.

14. Write `sub : nat * nat -> nat` to perform subtraction. (Hint: Use `pred`.)

15. Write `mult : nat * nat -> nat` to perform multiplication. (Hint: Use `add`.)

16. Write `less_than : nat * nat -> bool` to return `true` when the first argument is less than the second.

The remaining problems use this datatype, which represents sets of integers:

```
datatype intSet =
  Elems of int list (*list of integers, possibly with duplicates to be ignored*)
| Range of { from : int, to : int }  (* integers from one number to another *)
| Union of intSet * intSet (* union of the two sets *)
| Intersection of intSet * intSet (* intersection of the two sets *)

```

17. Write `isEmpty : intSet -> bool` that determines if the set is empty or not.

18. Write `contains: intSet * int -> bool` that returns whether the set contains a certain element or not.

19. Write `toList : intSet -> int list` that returns a list with the set's elements, without duplicates.