Practice Problems for Section 2

38 Cons Cells (*)

Write a function `length_of_a_list` that takes an arbitrary list (`'a list`) and evaluates to its length as an integer number. Use pattern matching instead of list functions. Your function should be tail-recursive.

SIGNATURE: `val length_of_a_list = fn : 'a list -> int`

EXAMPLE: `length_of_a_list [1] = 1`

Pass/Fail

We'll do some of the old and boring "students and their grades" stuff. We'll use the following definitions to represent students and their grades:

```
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 did not take the final exam).

Pass/Fail – 1

Write a function `pass_or_fail` that takes a `final_grade` and determines whether this particular student has passed or failed the class. The passing grade is 75 (inclusive) – everyone with a grade equal to or above 75 passes. Students with no grade are considered to have failed the class. Your function should evaluate to a value of type `pass_fail`. You may assume that the grade is correct (that is, in the specified range), and you do not need to validate the input. Use pattern matching instead of option functions.

NOTE: Your signature may be a little different.

SIGNATURE: `val pass_or_fail = fn : {grade:int option, id:'a} -> pass_fail`

EXAMPLE: `pass_or_fail { id = 1023, grade = SOME 73 } = fail`

Pass/Fail – 2

Write a function `has_passed` that takes a `final_grade` and evaluates to a `bool` indicating whether the given student has passed the class. The rules for determining that are the same as in the previous problem. Use pattern matching instead of option functions.

HINT: You might want to use `pass_or_fail` here.

NOTE: Your signature may be a little different.

SIGNATURE: `val has_passed = fn : {grade:int option, id:'a} -> bool`

EXAMPLE: `has_passed { id = 1023, grade = SOME 73 } = false`

Pass/Fail – 3

Write a function `number_passed` that takes a list of `final_grade`s and evaluates to a number of students on the list who have passed the class. The rules for determining that are still the same. You do not need to check whether all of the student IDs on the list are unique (though you may treat adding that to your implementation as a challenge problem). Use pattern matching instead of list functions.

NOTE: Your signature may be a little different.

SIGNATURE: `val number_passed = fn : {grade:int option, id:'a} list -> int`

EXAMPLE: `number_passed [{ id = 1, grade = SOME 65 }, { id = 2, grade = SOME 82 }, { id = 3, grade = NONE }, { id = 5, grade = SOME 96 }] = 2`

Pass/Fail – 4

Write a function `group_by_outcome` that takes a list of `final_grade`s and evaluates to a list of tuples consisting of no more than two elements. The resulting list should be empty if the original list was empty. If there are any students with a passing grade on the original list, the resulting list should contain a tuple with `pass` as the first element and the list of IDs of passing students as the second element. The IDs must be in the same order as they appear in the original list. Similarly, if there are any failing students on the list, the result should contain a tuple with `fail` for its first element and the list of IDs of failing students as its second element. Once again, the IDs must be in the same relative order as in the original list. Additionally, if there are both passing and failing students on the original list, the resulting list should contain `(pass, ...)` before `(fail, ...)`. As usual, use pattern matching.

NOTE: Your signature may be a little different.

SIGNATURE: `val group_by_outcome = fn : {grade:int option, id:'a} list -> (pass_fail * 'a list) list`

EXAMPLE: `group_by_outcome [{ id = 1025, grade = NONE }, { id = 4, grade = SOME 99 }] = [(pass, [4]), (fail, [1025])]`

Forest For The Trees

To flex our mental muscles with algebraic data types and records a little, we'll work with binary trees. We'll use the following definition:

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

Thus a binary tree is a recusive data structure that is either a plain `leaf`, or a `node` containing a value and two other trees of the same type. This parallels the definition of lists, but with two references to "following" elements.

Note that while this datatype can be used to build binary search trees, we're not assuming the existence of any invariant – values in the nodes may be of any type and may be distributed arbitrarily.

Forest For The Trees – 1

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

SIGNATURE: `val tree_height = fn : 'a tree -> int`

EXAMPLE: `tree_height (node { value = 0, left = node { value = 0, left = node { value = 0, left = leaf, right = leaf }, right = leaf }, right = node { value = 0, left = leaf, right = leaf } }) = 3`

Forest For The Trees – 2

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

SIGNATURE: `val sum_tree = fn : int tree -> int`

EXAMPLE: `sum_tree (node { value = 1, left = node { value = 2, left = node { value = 3, left = leaf, right = leaf }, right = leaf }, right = node { value = 4, left = leaf, right = leaf } }) = 10`

Forest For The Trees – 3

We'll define a simple datatype for this problem:

```
datatype flag = leave_me_alone | prune_me
```

Write a function `gardener` takes takes `flag tree` and evaluates to a new tree of the same type, such that its structure is identical to the original tree, but all nodes marked with `prune_me` have been removed together with their descendant nodes and replaced with leaves.

SIGNATURE: `val gardener = fn : flag tree -> flag tree`

EXAMPLE: `gardener (node { value = leave_me_alone, left = node { value = prune_me, left = node { value = leave_me_alone, left = leaf, right = leaf }, right = leaf }, right = node { value = leave_me_alone, left = leaf, right = leaf } }) = node { value = leave_me_alone, left = leaf, right = node { value = leave_me_alone, left = leaf, right = leaf } }`

Back To The Future!

A few of the practice problems from Section 1 can be rewritten more elegantly using the material from Section 2. All problem statements, **SIGNATURES** and **EXAMPLES** remain the same, if there are any additional considerations, these will be mentioned below. Only some of the potentially eligible problems are included – naturally, you're welcome to rewrite the rest on your own, using similar approaches.

GCD – Redux

Write a function `gcd_list` following the specification from Section 1's **Greated Common Divisor – Continued** problem. Use pattern matching instead of list functions. Use the following implementation of `gcd` as a helper function:

```
fun gcd (a : int, b : int) =
    if a = b
    then a
    else
        if a < b
        then gcd (a, b - a)
        else gcd (a - b, b)
```

Element Of A List – Redux

Write a function `any_divisible_by` following the specification from Section 1's **Element Of A List** problem. Use pattern matching instead of list functions. Use the following implementation of `is_divisible_by` as a helper function:

```
fun is_divisible_by (a : int, b : int) = a mod b = 0
```

Quirky Addition – Redux (**)

Write a function `add_opt` following the specification from Section 1's **Quirky Addition** problem. Use pattern matching instead of option functions. You may and should use option constructors.

Quirky Addition – Continued – Redux (**)

Write a function `add_all_opt` following the specification from Section 1's **Quirky Addition – Continued** problem. Use pattern matching instead of list and option functions.

Flip Flop – Redux (**)

Write a function `alternate` following the specification from Section 1's **Flip Flop** problem. Use pattern matching instead of list functions.

Minimum/Maximum – Redux (**)

Write a function `min_max` following the specification from Section 1's **Minimum/Maximum** problem. Use pattern matching instead of list functions.

Lists And Tuples, Oh My! - Redux

Write a function `unzip` following the specification from Section 1's **Lists And Tuples, Oh My!** problem. Use pattern matching instead of list functions.

NOTE: The type of your function is probably going to be more general that the one specified in the original problem. That's totally fine – awesome, actually!

BBCA – Redux (**)

Write a function `repeats_list` following the specification from Section 1's **BananaBanana – Continued (Again)** problem. Use pattern matching instead of list functions.

NOTE: The type of your function is probably going to be more general that the one specified in the original problem. That's totally fine – awesome, actually!

(*) The author of this problem apologizes for using an obscure cultural reference that is unlikely to be understood by anyone who does not speak author's native language as a title for this problem. Couldn't resist the temptation.

(**) Problems contributed by Charilaos Skiadas.