Practice Problems for Section 3

High-Order Fun

There Can Be Only One

Write functions `fold_map` and `fold_filter` that have the same signatures and behavior as `List.map` and `List.filter` correspondingly. Use `List.foldr`. Do not use pattern matching or any other list functions.

SIGNATURE: `val fold_map = fn : ('a -> 'b) -> 'a list -> 'b list`

EXAMPLE: `fold_map (fn x => x + 1) [1, 2, 3, 4, 5] = [2, 3, 4, 5, 6]`

SIGNATURE: `val fold_filter = fn : ('a -> bool) -> 'a list -> 'a list`

EXAMPLE: `fold_filter (fn x => x mod 2 = 0) [1, 2, 3, 4, 5] = [2, 4]`

The Evil Twin

Write a function `unfold` that takes a state transition function and an initial state and produces a list. On each step the current state is fed into the state transition function, which evaluates either to `NONE`, indicating that the result should contain no more elements, or to `SOME`

$$pair$$

, where

$$pair$$

contains the next state and the next list element.

SIGNATURE: `val unfold = fn : ('a -> ('a * 'b) option) -> 'a -> 'b list`

EXAMPLE: `unfold (fn x => if x > 3 then NONE else SOME (x + 1, x)) 0 = [0, 1, 2, 3]`

A Novel Approach

Write a function `factorial` that takes an integer number

$$n$$

and evaluates to

$$n!$$

. Your function should be a composition of `unfold` and `List.foldl`. You should not use any other list functions, recursion or pattern matching.

BONUS QUESTION: Is this function as good as a simple tail-recursive factorial implementation?

SIGNATURE: `val factorial = fn : int -> int`

EXAMPLE: `factorial 4 = 24`

Unforeseen Developments

Write a function `unfold_map`, that behaves exactly as `List.map` and `fold_map`, but that would be implemented in terms of `unfold`.

SIGNATURE: `val unfold_map = fn : ('a -> 'b) -> 'a list -> 'b list`

EXAMPLE: `unfold_map (fn x => x + 1) [1, 2, 3, 4, 5] = [2, 3, 4, 5, 6]`

So Imperative (*)

Write a function `do_until` that takes three arguments, `f`, `p` and `x`, and keeps applying `f` to `x` until `p x` evaluates to `true`. Upon reaching that condition, `f (f (f ... (f x) ...))` is returned.

SIGNATURE: `val do_until = fn : ('a -> 'a) -> ('a -> bool) -> 'a -> 'a`

EXAMPLE: `do_until (fn x => x div 2) (fn x => x mod 2 <> 0) 48 = 3`

Yet Another Factorial

Write a function `imp_factorial` that has the same behavior as the `factorial` function described above, but is defined in terms of `do_until`.

NOTE: There is a deep relationship between these two versions of `factorial` function, with `imp_factorial` eliminating the building of an intermediate list.

SIGNATURE: `val imp_factorial = fn : int -> int`

EXAMPLE: `imp_factorial 4 = 24`

Fixed Point (*)

Write a function `fixed_point` that accepts some function `f` and an initial value `x`, and keeps applying `f` to `x` until an `x` is found such that `f x = x`. Note that the function must have the same domain and codomain, and that the values must be comparable for equality.

SIGNATURE: `val fixed_point = fn : (''a -> ''a) -> ''a -> ''a`

EXAMPLE: `fixed_point (fn x => x div 2) 17 = 0`

Newton's Method

Square root of a real number

$$n$$

is a fixed point of function

$$f_n(x) = \frac{1}{2}(x + \frac{n}{x})$$

. Unfortunately, for reasons rooted in the arcane art of numerical analysis, `reals` are not comparable for equality in Standard ML. Write a function `my_sqrt` that takes a real number and evaluates to an approximation of its square root. You will probably need to write a version of `fixed_point` that uses "difference in absolute value less than

$$\epsilon$$

" as a test for equality. Use

$$\epsilon = 0.0001$$

. Use the number itself as an initial guess.

SIGNATURE: `val my_sqrt = fn : real -> real`

EXAMPLE: `abs (my_sqrt 2.0 - Math.sqrt 2.0) < 0.01`

Deeper Into The Woods

Let's reuse the binary tree data structure from practice problems for Section 2:

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

Write functions `tree_fold` and `tree_unfold` that would serve as equivalents of `fold` and `unfold` on lists for this data structure.

HINT: This is a hard problem, but consider this: the initial value for `fold` corresponds to the base case of recursion on lists (i.e., matching `[]`), while the function passed to the `fold` corresponds to the case when we match on `::`.

[] and :: correspond to `leaf` and `node` data constructors. Similar reasoning applies to `unfold`. You might also want to meditate over the signatures below if this does not provide sufficient insight.

SIGNATURE: `val tree_fold = fn : ('a * 'b * 'a -> 'a) -> 'a -> 'b tree -> 'a`

EXAMPLE: `tree_fold (fn (l, v, r) => l ^ v ^ r) "!" (node { value = "foo", left = node { value = "bar", left = leaf, right = leaf }, right = node { value = "baz", left = leaf, right = leaf }}) = "!bar!foo!baz!"`

SIGNATURE: `val tree_unfold = fn : ('a -> ('a * 'b * 'a) option) -> 'a -> 'b tree`

EXAMPLE: `tree_unfold (fn x => if x = 0 then NONE else SOME (x - 1, x, x - 1)) 2 = node { value = 2, left = node { value = 1, left = leaf, right = leaf }, right = node { value = 1, left = leaf, right = leaf }}`

A Grand Challenge

Let's try to write a simple type inference algorithm for a very simple expression language. We won't deal with functions, variables or polymorphism.

The expressions will be represented by the following data type:

```
datatype expr = literal_bool | literal_int | binary_bool_op of expr * expr | binary_int_op o
```

The data constructors represent literal booleans, literal integers, binary operators on booleans, binary operators on integers, comparison operators and conditionals. Since we're only interested in types, and not in actually evaluating our expressions, we're omitting immaterial details, such as whether a literal boolean is "true" or "false", or whether an operator on integers is addition, subtraction or something else entirely.

The types will be represented by the following simple datatype:

```
datatype expr_type = type_bool | type_int
```

The typing rules for our expression language are simple:

1. Literal booleans are of type `type_bool`.
2. Literal integers have type `type_int`.
3. Boolean operators have type `type_bool` provided that both of their operands also have type `type_bool`.

4. Integer operators have type `type_int` provided that both operands also have type `type_int`.
5. Comparison operators have type `type_bool` provided that both operands have type `type_int`.
6. Conditionals have the same type as the first branch, provided that the second branch has the same type, and the condition has type `type_bool`.

Write a function `infer_type` that accepts an `expr` and evaluates to the type of the given expression. If the type cannot be determined according to the rules above, raise `TypeError` exception.

SIGNATURE: `val infer_type = fn :  expr -> expr_type`

EXAMPLE: `infer_type (conditional (literal_bool, literal_int, binary_int_op (literal_int, literal_int))) = type_int`

Back To The Future! 2

A few of the practice problems from Sections 1 and 2 can be rewritten more elegantly using the material from Section 3. 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 – Final Redux

Write a function `gcd_list` following the specification from Section 1's **Greated Common Divisor – Continued** problem. Use folds. 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 – Final Redux

Write a function `any_divisible_by` following the specification from Section 1's **Element Of A List** problem. Use folds or other high-order 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 – Continued – Final Redux (*)

Write a function `add_all_opt` following the specification from Section 1's **Quirky Addition – Continued** problem. Use folds.

Flip Flop – Final Redux (*)

Write a function `alternate` following the specification from Section 1's **Flip Flop** problem. Use folds.

Minimum/Maximum – Final Redux (*)

Write a function `min_max` following the specification from Section 1's **Minimum/Maximum** problem. Use folds.

Lists And Tuples, Oh My! - Final Redux

Write a function `unzip` following the specification from Section 1's **Lists And Tuples, Oh My!** problem. Use folds.

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!

Lists And Tuples, Oh My! – Continued (1) – Final Redux (*) (**)

Write a function `zip` following the specification from Section 1's **Lists And Tuples, Oh My! – Continued (1)** problem. Use `unfold` that you wrote in **The Evil Twin** problem.

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 – Final Redux (*)

Write a function `repeats_list` following the specification from Section 1's **BananaBanana – Continued (Again)** problem. Use folds.

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!

38 Cons Cells – Final Redux

Write a function `length_of_a_list` following the specification from Section 2's **38 Cons Cells** problem. Use folds.

Forest For The Trees – Final Redux

Write functions `tree_height`, `sum_tree` and `gardener` following specifications from Section 2's **Forest For The Trees** series of problems. Use `tree_fold` and/or `tree_unfold`.

(*) Problems contributed by Charilaos Skiadas.

(**) And yes, that's a stupid title for a problem. Charilaos had nothing to do with *that* part of it.