Practice Problems for Section 1

Positive Numbers

Write a function `is_positive` that takes an integer number and evaluates to `true` or `false`. The function should evaluate to `true` if its argument is a positive number, and to `false` otherwise.

SIGNATURE: `val is_positive = fn : int -> bool`

EXAMPLE: `is_positive 1 = true`

Divisibility

Write a function `is_divisible_by` that takes two integer numbers and evaluates to `true` or `false`. It should evaluate to `true` if its first argument is divisible by its second argument, and to `false` otherwise. You may assume that the second argument will be non-zero.

SIGNATURE: `val is_divisible_by = fn : int * int -> bool`

EXAMPLE: `is_divisible_by (6, 3) = true`

Integer Division

Write a function `divide_by` that takes two integer numbers and evaluates to the result of the integer division of the first one by the second one. You may assume that the first argument is non-negative and the second one is strictly positive. You should not use the `div` operator.

HINT: Recall the `pow` function in the lectures.

SIGNATURE: `val divide_by = fn : int * int -> int`

EXAMPLE: `divide_by (7, 3) = 2`

Greatest Common Divisor

Write a function `gcd` that takes two integer numbers and evaluates to their greatest common divisor. Use the Euclidean algorithm:

http://en.wikipedia.org/wiki/Greatest_common_divisor

You may assume that both numbers are positive.

SIGNATURE: `val gcd = fn : int * int -> int`

EXAMPLE: `gcd (18, 12) = 6`

Least Common Multiple

Write a function `lcm` that takes two integer numbers and evaluates to their least common multiple. LCM can be defined in terms of GCD:

http://en.wikipedia.org/wiki/Least_common_multiple

You may assume that both numbers are positive.

SIGNATURE: `val lcm = fn :  int * int -> int`

EXAMPLE: `lcm (18, 12) = 36`

Greatest Common Divisor – Continued

Write a function `gcd_list` that takes a list of integers and evaluates to their GCD. GCD of a set of numbers can be defined in terms of binary GCD:

$$\gcd(a_1, \ldots, a_n) = \gcd(a_1, \gcd(a_2, \gcd(a_3, \ldots)))$$

You may assume that the list is non-empty and all the numbers on the list are positive.

SIGNATURE: `val gcd_list = fn :  int list -> int`

EXAMPLE: `gcd_list [18, 12, 3] = 3`

Element Of A List

Write a function `any_divisible_by` that takes a list of integers and a divisor (an integer number) and evaluates to either `true` or `false`. The function should evaluate to `true` if and only if there exists an element of the list that is divisible by the function's second argument.

SIGNATURE: `val any_divisible_by = fn :  int list * int -> bool`

EXAMPLE: `any_divisible_by ([13, 1, 20], 5) = true`

Integer Division – Continued

Write a function `safe_divide_by` that takes two integer numbers and evaluates to an `int option`. If the second argument is non-zero, the function should evaluate to `SOME`

$$x$$

where

$$x$$

is the result of the integer division of the first argument by the second one, otherwise it should evaluate to `NONE`. You may and should use the `div` operator for this problem.

SIGNATURE: `val safe_divide_by = fn : int * int -> int option`

EXAMPLE: `safe_divide_by (7, 3) = SOME 2`

Quirky Addition (*)

Write a function `add_opt` that given two "optional" integers, adds them if they are both present, or evaluates to `NONE` if at least one of the two arguments is `NONE`.

SIGNATURE: `val add_opt = fn : int option * int option -> int option`

EXAMPLE: `add_opt (SOME 1, SOME 2) = SOME 3`

Quirky Addition – Continued (*)

Write a function `add_all_opt` that given a list of "optional" integers, adds those integers that are there (i.e. adds all the `SOME i`). If the list does not contain any `SOME` in it, i.e. they are all `NONE` or the list is empty, the function should evaluate to NONE.

HINT: It probably wouldn't make sense to use `add_opt` for this.

SIGNATURE: `val add_all_opt = fn : int option list -> int option`

EXAMPLE: `add_all_opt [SOME 1, NONE, SOME 3] = SOME 4`

Flip Flop (*)

Write a function `alternate` that takes a list of numbers and adds them with alternating sign. The result of applying this function to `[1, 2, 3, 4]` should be `1 - 2 + 3 - 4 = ~`.

SIGNATURE: `val alternate = fn : int list -> int`

EXAMPLE: `alternate [1, 2, 3, 4] = ~`

Minimum/Maximum (*)

Write a function `min_max` that takes a non-empty list of numbers, and evaluates to a tuple `(min, max)` of the minimum and maximum of the numbers in the list.

SIGNATURE: `val min_max = fn : int list -> int * int`

EXAMPLE: `min_max [3, 1, 2, 5, 4] = (1, 5)`

Lists And Tuples, Oh My!

Write a function `unzip` that takes an `(int * int) list` and evaluates to `int list * int list` such that the first element of the resulting tuple is a list consisting of all first elements of the argument (in order), and the second element of the result consists of all second elements of the tuples in the original list.

HINT: There are several approaches to this, some of which could be directly based on the code in the lectures.

SIGNATURE: `val unzip = fn : (int * int) list -> int list * int list`

EXAMPLE: `unzip [(1, 2), (3, 4), (5, 6)] = ([1, 3, 5], [2, 4, 6])`

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

Write a function `zip` that given two lists of integers evaluates to a list of corresponding consecutive pairs, stopping when one of the lists is empty.

SIGNATURE: `val zip = fn : int list * int list -> (int * int) list`

EXAMPLE: `zip ([1, 2, 3], [4, 6]) = [(1, 4), (2, 6)]`

Lists And Tuples, Oh My! – Continued (2) (*)

Write a version `zip_recycle` of `zip`, where when one list is empty it starts recycling from its start until the other list completes.

SIGNATURE: `val zip_recycle = fn : int list * int list -> (int * int) list`

EXAMPLE: `zip_recycle ([1, 2, 3], [4, 6]) = [(1, 4), (2, 6), (3, 4)]`

Lists And Tuples, Oh My! – Continued (3) (*)

Write a version `zip_opt` of `zip` that should evaluate to `SOME` list when the original lists have the same length, and to `NONE` if they do not.

SIGNATURE: `val zip_opt = fn : int list * int list -> (int * int) list option`

EXAMPLE: `zip_opt ([1, 2, 3], [4, 6]) = NONE`

BananaBanana

Write a function `duplicate` that takes a `string list` and evaluates to another `string list`, consisting of the elements of the original list, in the same order, but with each one repeated twice.

SIGNATURE: `val duplicate = fn : string list -> string list`

EXAMPLE: `duplicate ["a", "bc", "def"] = ["a", "a", "bc", "bc", "def", "def"]`

Greetings, Earthlings! (*)

Write a function `greeting` that given an (optional) name string evaluates to the string `"Hello there, ...!"` where the dots would be replaced by the name. Note that the name is given as an option, so if it is `NONE` then replace the dots with `"you"`.

SIGNATURE: `val greeting = fn : string option -> string`

EXAMPLE: `greeting (SOME "Charilaos") = "Hello there, Charilaos!"`

BananaBanana – Continued

Write a function `repeats` that takes a string and an integer number

$$n$$

and evaluates to a `string list` consisting of

$$n$$

elements. The first element must be equal to the first argument of the function, the second one must be equal to the first argument repeated twice, the third one will be the argument repeated three times etc. You will need to use the string concatenation operator `^`. You may assume that the second argument is non-negative.

SIGNATURE: `val repeats = fn : string * int -> string list`

EXAMPLE: `repeats ("banana", 2) = ["banana", "bananabanana"]`

BananaBanana – Continued (Again) (*)

Write a function `repeats_list` that given a list of strings and a list of non-negative integers, repeats the strings in the first list according to the numbers indicated by the second list. You may assume that both lists have the same length.

SIGNATURE: `val repeats_list = fn : string list * int list -> string list`

EXAMPLE: `repeats_list (["abc", "def", "ghi"], [4, 0, 3]) = ["abc", "abc", "abc", "abc", "ghi", "ghi", "ghi"]`

(*) Problems contributed by Charilaos Skiadas.