CSE 341, Autumn 2020, Assignment 1
Due: Octobor 14, 5:59:59PM

You will write 12 OCaml functions (and tests for them) related to calendar dates. In all problems, a "date" is an OCaml value of type `int*(int*int)`, where the first part is the day, the second part is the month, and the third part is the year. A "reasonable" date has a positive year, a month between 1 and 12, and a day no greater than 31 (or less depending on the month). Your solutions need to work correctly only for reasonable dates, but do not check for reasonable dates (that is a challenge problem) and many of your functions will naturally work correctly for some/all non-reasonable dates. A "day of year" is a number from 1 to 365 where, for example, 33 represents February 2. (We ignore leap years except in one challenge problem.)

When writing you solution, feel free to refer to the library functions listed at the end of the homework. If you encountered an error saying "Unbound module Option", you probably have an old version of OCaml (4.07 or older). Upgrade your installation, or see the course website for instructions about how to upgrade on attu.

1. Write a function `is_older` that takes two dates and evaluates to true or false. It evaluates to true if the first argument is a date that comes before the second argument. (If the two dates are the same, the result is false.)

2. Write a function `number_in_month` that takes a list of dates and a month (i.e., an `int`) and returns how many dates in the list are in the given month.

3. Write a function `number_in_months` that takes a list of dates and a list of months (i.e., an `int list`) and returns the number of dates in the list of dates that are in any of the months in the list of months. *Assume the list of months has no number repeated.* Hint: Use your answer to the previous problem.

4. Write a function `dates_in_month` that takes a list of dates and a month (i.e., an `int`) and returns a list holding the dates from the argument list of dates that are in the month. The returned list should contain dates in the order they were originally given.

5. Write a function `dates_in_months` that takes a list of dates and a list of months (i.e., an `int list`) and returns a list holding the dates from the argument list of dates that are in any of the months in the list of months. *Assume the list of months has no number repeated.* Hint: Use your answer to the previous problem and OCaml's list-append operator (`@`).

6. Write a function `get_nth` that takes a list of strings and a positive `int` n and returns the n^{th} element of the list where the head of the list is 1^{st}. Do not worry about the case where the list has too few elements: your function may apply `List.hd` or `List.tl` to the empty list in this case, which is okay.

7. Write a function `string_of_date` that takes a date and returns a `string` of the form `September-10-2015` (for example). Use the operator `^` for concatenating strings and the library function `string_of_int` for converting an `int` to a `string`. For producing the month part, do *not* use a bunch of conditionals. Instead, use a list holding 12 strings and your answer to the previous problem. For consistency, use hyphens exactly as in the example and use English month names: January, February, March, April, May, June, July, August, September, October, November, December.

8. Write a function `number_before_reaching_sum` that takes an `int` called *sum*, which you can assume is positive, and an `int list`, which you can assume contains all positive numbers, and returns an `int`. You should return an int n such that the first n elements of the list add to less than sum, but the first $n+1$ elements of the list add to *sum* or more. Assume the entire list sums to more than the passed in value; it is okay for an exception to occur if this is not the case.

9. Write a function `what_month` that takes a day of year (i.e., an `int` between 1 and 365) and returns what month that day is in (1 for January, 2 for February, etc.). Use a list holding 12 integers and your answer to the previous problem.

10. Write a function `month_range` that takes two days of the year `day1` and `day2` and returns an `int list` `[m1;m2;...;mn]` where `m1` is the month of `day1`, `m2` is the month of `day1+1`, ..., and `mn` is the month of day `day2`. Note the result will have length `day2 - day1 + 1` or length 0 if `day1>day2`.

11. Write a function `oldest` that takes a list of dates and evaluates to an `(int*(int*int)) option`. It evaluates to `None` if the list has no dates else `Some` d where the date d is the oldest date in the list.

12. Write a function `cumulative_sum` that takes a list of numbers and returns a list of the partial sums of these numbers. For example, `cumulative_sum [12;27;13] = [12;39;52]`. Hint: Use a helper function that takes two arguments.

13. **Challenge Problem:** Write functions `number_in_months_challenge` and `dates_in_months_challenge` that are like your solutions to problems 3 and 5 except having a month in the second argument multiple times has no more effect than having it once. (Hint: Remove duplicates, then use previous work.)

14. **Challenge Problem:** Write a function `reasonable_date` that takes a date and determines if it describes a real date in the common era. A "real date" has a positive year (year 0 did not exist), a month between 1 and 12, and a day appropriate for the month. Solutions should properly handle leap years. Leap years are years that are either divisible by 400 or divisible by 4 but not divisible by 100. (Do not worry about days possibly lost in the conversion to the Gregorian calendar in the Late 1500s.)

Note: Remember that challenge problems are worth very few points, especially as compared to their difficulty. Do them only after you've solved the other problems!

Note: There may be problems that have a corresponding functions with exactly functionality. In that case, don't copy the standard library implementation, which will give you no points, because it uses the wrong language features that are not covered in class.

Note: The sample solution contains *roughly* 90–100 lines of code, not including challenge problems.

Summary

Evaluating a correct homework solution should generate these bindings:

```
val is_older : (int * (int * int)) * (int * (int * int)) -> bool = <fun>
val number_in_month : (int * (int * int)) list * int -> int = <fun>
val number_in_months : (int * (int * int)) list * int list -> int = <fun>
val dates_in_month :
  (int * (int * int)) list * int -> (int * (int * int)) list = <fun>
val dates_in_months :
  (int * (int * int)) list * int list -> (int * (int * int)) list = <fun>
val get_nth : string list * int -> string = <fun>
val string_of_date : int * (int * int) -> string = <fun>
val number_before_reaching_sum : int * int list -> int = <fun>
val what_month : int -> int = <fun>
val month_range : int * int -> int list = <fun>
val oldest : (int * (int * int)) list -> (int * (int * int)) option = <fun>
val cumulative_sum : int list -> int list = <fun>
```

Of course, generating these bindings does not guarantee that your solutions are correct. *Test your functions: Put your testing code in a separate file. We will not grade the testing file, but you must turn it in.*

Assessment

Solutions should be:

- Correct
- In good style, including indentation and line breaks
- Written using features discussed in class. In particular, you must *not* use OCaml's mutable references or arrays. (Why would you?) Also do *not* use pattern-matching; it is the focus of the next assignment.

Turn-in Instructions

Put all your solutions in a file named `hw1.ml` and your tests in `hw1_test.ml`, then submit both of them to the CSE 341 Autumn 2020 Gradescope Homework 1 Assignment page.

Syntax Hints

Small syntax errors can lead to strange error messages. Here are 3 examples for function definitions:

1. `int * (int * int) list` means `int * ((int * int) list)`, not `(int * (int * int)) list`.
2. `let f x : t` means the *result type* of `f` is `t`, whereas `let f(x : t)` means the *argument type* of `f` is `t`. There is no need to write result types (and in later assignments, no need to write argument types).
3. `let f(x t)`, `let f(t x)`, or `let f(t : x)` are all wrong, but the error message suggests you are trying to do something much more advanced than you actually are (which is trying to write `let f(x : t)`).
4. `let f(x : t, y : t)` will cause a syntax error in OCaml. The correct format is `let f((x : t), (y : t))`.

Library Functions

Here are a cheatsheet of library functions and operations that you may find help when completing your solution.

- `fst (* get the first value of a pair *)`
- `snd (* get the second value of a pair *)`
- `List.hd (* get the head of the list *)`
- `List.tl (* get the tail of the list *)`
- `@ (* infix operator to append two list together *)`
- `^ (* infix operator to concat two string together *)`
- `string_of_int (* convert an integer into a string *)`
- `Option.get (* if the argument = Some value, return value, otherwise raise an exception *)`
- `Option.is_some (* return a bool indicating whether opt = Some value *)`
- `Option.is_none opt (* return a bool indicating whether opt = None *)`