COMP 360, Spring 2013, Assignment 1

You will write 11 Racket functions related to calendar dates. In all problems, a "date" is an Racket list containing three values: the first part is the year, the second part is the month, and the third part is the day. For example, January 16, 2013 would be represented by the list '(2013 1 16). A "reasonable" date would have a positive year, a month between 1 and 12, and a day no greater than 31 (or less depending on the month). However, most problems do *not* assume "reasonable" dates; solutions should work for any date except where noted. A "day of year" is a number from 1 to 365 where, for example, 33 represents February 2. (We ignore leap years entirely.)

For the examples, we use ==> to mean "evaluates to."

The sample solution is *roughly* 60–65 lines. See the last page for additional instructions.

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

   Example: `(before?  '(2013 4 2) '(2013 5 1)) ==> #t`

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

   Example:

   `(number-in-month '((2013 1 2) (2012 2 1) (2015 2 3) (2013 12 1)) 2) ==> 2`

3. Write a function `number-in-months` that takes a list of dates and a list of months (i.e., a list of integers) 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 (or if a number is repeated then dates in that month are counted multiple times).* Use your answer to the previous problem.

   Example:

   `(number-in-months '((2013 1 2) (2012 2 1) (2015 2 3) (2013 12 1)) '(12 2))`
   `  ==> 3`

4. Write a function `dates-in-month` that takes a list of dates and a month (i.e., an integer) and returns a list holding the dates from the argument list of dates that are in the month.

   Example:

   `(dates-in-month '((2013 1 2) (2012 2 1) (2015 2 3) (2013 12 1)) 2)`
   `  ==> '((2012 2 1) (2015 2 3))`

5. Write a function `dates-in-months` that takes a list of dates and a list of months (i.e., a list of integers) 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 (or if a number is repeated then dates in that month are in the result list multiple times).* Use your answer to the previous problem and `append`.

   Example:

   `(dates-in-months '((2013 1 2) (2012 2 1) (2015 2 3) (2013 12 1)) '(12 2))`
   `  ==> '((2013 12 1) (2012 2 1) (2015 2 3))`

6. Write a function `get-nth` that takes a list and an integer n and returns the nth element of the list where the `car` of the list is 1st. If the list has too few elements, your function should apply `car` to the empty list, which will raise an exception.

   Example:

   ```
   (get-nth '(7 5 3 8 1) 2) ==> 5
   ```

7. Racket contains a `string` data type that is has similar functionality to strings in Python or C++. For instance, a literal string in Racket is a sequence of characters enclosed by double quotes. Write a function `date->string` that takes a date and returns a `string` of the form `April 11, 2011` (for example). Use the function `string-append` for concatenating strings and the function `number->string` for converting an integer 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.

   Example:

   ```
   (date->string '(2012 7 29)) ==> "July 29, 2012"
   ```

8. Write a function `number-before-reaching-sum` that takes an integer called `sum` (which you can assume is non-negative) and a list of integers and returns an integer. It returns n if `sum` is greater than or equal to the sum of the first n elements of the list, but not greater than or equal to the sum of the first $n + 1$ elements. If `sum` is greater than the sum of all numbers in the list, your function should apply `car` to the empty list, which will raise an exception.

   Example:

   ```
   (number-before-reaching-sum 130 '(40 30 50 10 90)) ==> 4
   ```

9. Write a function `what-month` that takes a day of year (i.e., an integer between 1 and 365) and returns an integer representing the 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.

   Example:

   ```
   (what-month 138) ==> 5
   ```

10. Write a function `month-range` that takes two days of the year `day1` and `day2` (integers) and returns a list of integers $(m_1\ m_2\ \dots\ m_n)$ where m_1 is the month of `day1`, m_2 is the month of `day1+1`, ..., and m_n is the month of day `day2`. Note the result will have length `day2 - day1 + 1` or length 0 if `day1 > day2`.

    Example:

    ```
    (month-range 30 34) ==> '(1 1 2 2 2)
    ```

11. Write a function `earliest` that takes a list of dates and returns the earliest date in the list. The list is guaranteed to contain at least one date.

    Example:

    ```
    (earliest '((2013 1 2) (2012 2 1) (2015 2 3) (2013 12 1))) ==> '(2012 2 1)
    ```

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 any mutation operations nor arrays (even though Racket has them).

Turn-in Instructions

- Put all your solutions in one file, `hw1_lastname_firstname.rkt`, where `lastname` is replaced with your last name, and `firstname` is replaced with your first name.

- Upload your file to Moodle before the project deadline.

Notes

- Yes, we already wrote `get-nth` in class, but this version starts the indices at 1, not 0. This makes the `date->string` function simpler because we usually consider January to be month 1, not month 0.

- To write a literal list that contains (literal) sub-lists, do not use additional quotes:

  ```
  Right: '(1 2 (3 4))
  Wrong: '(1 2 '(3 4))
  ```

- To write a literal list that contains string literals, you should use double quotes around each individual string:

  ```
  Right: '("Hello" "World")
  Wrong: '(Hello World)
  ```

  The easy way to remember this is that you still always need double quotes around every string literal, just like in Python or C++. All the single quote does before a list is stops evaluating the individual terms inside the list, so you can write '(1 2 3) without the number 1 being interpreted as the name of a function.