CSci 555-01: Functional Programming
Assignment #1, Spring 2019

H. Conrad Cunningham

1 February 2019

Assignment #1

Due Wednesday, 13 February, 11:59 p.m.

General Instructions

All homework and programming exercises must be prepared in accordance with the instructions given in the Syllabus. Each assignment must be submitted to your instructor by its stated deadline.

Citations: In accordance with expected scholarly and academic standards, if you reference outside textbooks, reference books, articles, websites, etc., or discuss an assignment with individuals inside or outside the class, you must document these by including appropriate citations or comments at prominent places in your submission such as in the header of the primary source file.

Identification: Put your name, course name, and assignment number as comments in each file you submit.

Assignment Description

1. This is an individual assignment.
2. When complete, submit your Scala source code files to the course Blackboard site for Assignment #1.
3. Be sure to document your source code appropriately using program comments. Give attention to the general instructions given above and in the Syllabus.

   Consider documenting the source code with preconditions, postconditions, termination arguments, and time and space complexity.

4. **Do exercises 14, 15, 16, 19, 20, and 24** from the notes on *Recursion Styles, Correctness, and Efficiency*: [HTML] [PDF]

   Exercise 20 is optional for undergraduates.

5. Test and documen

Exercises

14. Develop a backward recursive Scala function `sumFromTo` such that `sumFromTo(m,n)` computes the sum of the integers from `m` to `n` for `n >= m`.

15. Develop a Scala function `sumFromTo2` such that `sumFromTo2(m,n)` computes the sum of the integers from `m` to `n n >= m`. Use a tail recursive auxilliary function.

16. Suppose we have Scala functions `succ` (successor) and `pred` (predecessor) defined as follows:

    ```
    def succ(n: Int): Int = n + 1
    def pred(n: Int): Int = n - 1
    ```

    Develop a recursive Scala function `add` such that `add(m,n)` computes `m + n` for two integers `m` and `n`. Function `add` *cannot* use addition or subtraction operators but *can* use unary negation, comparisons between integers, and the `succ` and `pred` functions defined above.

17. OMITTED

18. OMITTED

19. Develop a recursive Scala function `hailstone` to implement the following function:

$hailstone(n)$	$=$	$1,$	if $n = 1$
$hailstone(n)$	$=$	$hailstone(n/2),$	if $n > 1$, even n
$hailstone(n)$	$=$	$hailstone(3 * n + 1),$	if $n > 1$, odd n

    Note that an application of the `hailstone` function to the argument `3` would result in the following "sequence" of "calls" and would ultimately return the result `1`.

    ```
    hailstone(3)
      hailstone(10)
        hailstone(5)
          hailstone(16)
            hailstone(8)
              hailstone(4)
                hailstone(2)
    ```

```
hailstone(1)
```

What is the domain of the *hailstone* function? How do we know the function terminates?

20. Develop a Scala exponentiation function `expt4` that is similar to `expt3` but is tail recursive as well as logarithmic recursive.

21. OMITTED

22. OMITTED

23. OMITTED

24. A date on the *proleptic Gregorian calendar* (see note below) can be represented in Scala by the definition

```
case class PGDate(year: Int, month: Int, day: Int)
```

with the following constraints on *valid* objects:

- `year` is any integer
- `1 <= month <= 12`
- `1 <= day <= days_in_month(year,month)`

Here `days_in_month(year,month)` represents the number of days in the the given `month` (i.e. 28, 29, 30, or 31) for the given `year`. Remember that the number of days in February varies between regular and leap years.

For the items below, write your own Scala functions. Do not use a date library.

a. Extend class `PGdate`{scala} to implement trait `Ord` as defined below (and in the *Notes on Scala for Java Programmers*):

```
trait Ord {
    def < (that: Any): Boolean
    def <=(that: Any): Boolean =
        (this < that) || (this == that)
    def > (that: Any): Boolean = !(this <= that)
    def >=(that: Any): Boolean = !(this < that)
}
```

If needed, redefine the method `equals`.

The interpretation of `d1 < d2` is that `d1` is an earlier date than `d2`.

b. Redefine method `toString` appropriately for `PGDate`.

c. Develop a Scala function `validPGDate(d)` that takes a `PGDate` object `d` and returns `true` if and only if `d` satisfies the constraints given above.

For example:

- `validPGDate(PGDate(2019,2,1)) == true`
- `validPGDate(PGDate(2016,2,29)) == true`
- `validPGDate(PGDate(2017,2,29)) == false`
- `validPGDate(PGDate(0,0,0)) == false`

You may need to develop one or more other functions to implement the `validPGDate` function.

d. For any `PGDate` beginning with (i.e. `>=`) `PGDate(-4712,1,1)`, develop Scala functions:

- `daysBetween(d1,d2)` that takes two valid `PGDate` objects `d1` and `d2` and returns the number of days between them. The difference value is positive if `d1 < d2` and negative if `d1 > d2`.
- `addDays(d,days)` takes a `PGDate` object `d` and an integer number of days and returns a valid `PGDate` object that is offset by that number of days. A positive offset results in a later date.

Note: The Gregorian calendar [Wikipedia 2019] was introduced by Pope Gregory of the Roman Catholic Church in October 1582. It replaced the Julian calendar system, which had been instituted in the Roman Empire by Julius Caesar in 46 BC. The goal of the change was to align the calendar year with the astronomical year.

Some countries adopted the Gregorian calendar at that time. Other countries adopted it later. Some countries may never have adopted it officially.

However, the Gregorian calendar system became the common calendar used worldwide for most civil matters. The *proleptic Gregorian calendar* [Wikipedia 2019] extends the calendar backward in time from 1582. The year 1 BC becomes year 0, 2 BC becomes year -1, etc. The proleptic Gregorian calendar underlies the ISO 8601 standard used for dates and times in software systems [Wikipedia 2019].

Arithmetic on calendar dates is often done by converting a date to the Julian Day Number (JDN), doing the arithmetic on those values, and then converting back to the calendar date [Wikipedia 2019].