Exam of Programming Languages

27 January 2016

Disclaimer. Note that to have a running solution for an exercise is not enough: you need a well-cooked solution that proves your ability to use what explained during the classes. All the exercises have the same value: 11; Exercise are valued separately and it is necessary to get at least 6 points each exercise to pass the exams, i.e., 18 achieved with only 2 exercises means to fail the exam.

Exercise OCaML: Hooray! It's Prime!!!

To verify that a number is prime is considered a trivial task for a computer scientist. This is true when the number to check is pretty small (let's say if it is below or equal to 10000) but when the number to check grows larger the naïve approach (namely trial division) tends to become slower and slower up to be intractable.

Fortunately in the years, several mathematicians proposed different algorithms to check the primality of big numbers. Just to cite a few *Lucas-Lehmer* and *Fermat's Little Theorem*.

The **Lucas–Lehmer** test works as follows. Let $M^p = 2^p-1$ be the *Mersenne number* to test with *p* an odd prime. The primality of *p* can be efficiently checked with a simple algorithm like trial division since *p* is exponentially smaller than M^p. Define a sequence $\{ s_i \}$ for all $i \geq 0$ by

$$s_i = \begin{cases} 4 & \text{if } i = 0; \\ s_{i-1}^2 - 2 & \text{otherwise.} \end{cases}$$

Then M^p is prime if and only if $s_{p-2} \equiv 0 \pmod{M^p}$. The number $s_{p-2} \pmod{M^p}$ is called the Lucas–Lehmer residue of *p*.

Fermat's little theorem states that if *p* is prime and $0<a<p$, then $a^{p-1} \equiv 1 \pmod{p}$. If we want to test whether *p* is prime, then we can pick random *a*'s in the interval and see whether the equality holds. If the equality does not hold for a value of *a*, then *p* is not prime. If the equality does hold for many values of *a*, then we can say that *p* is **probably** prime.

As you can notice, neither the Lucas-Lehmer's primality test nor the Fermat's Little Theorem provide a certain for all numbers. But in an case we can use them to define the number as *probably* prime or *surely* not prime.

The exercise consists of implementing a module `Primality` with (at least) 4 functions: `trialdivision`, `lucaslehmer`, `littlefermat` and `is_prime`. Each of them is a predicate over an integer number (the one that should be tested as prime). Thje first three functions should implement the described algorithms. The last one is testing the primality of the input by applying one of the other algorithms according to the following rules:

- if the input is smaller or equal to 10000 it will use the trial division algorithm.
- if the input is between 10001 and 524287 (extremes included) it will use the Lucas-Lehmar's algorithm (no check if the input is a Marsenne number).
- it will use the Fermat's little theorem otherwise (please use a reasonable (both in size and values) set of values for *a*).

Note that the given interfaces must be respected and your module should be compliant with the following main and its execution. Note that the last value takes more or less ten minutes to be validated.

```
open Primality ;;

let main () =
  let rec test_primes = function
  |  hd::tl ->  Printf.printf "%10d :- %b\n" hd (is_prime hd);
                flush stdout; test_primes tl
  |  []     ->  Printf.printf "\n"
  in test_primes [25; 127; 8191; 131071;
     524286; 524287; 524288; 2147483647] ;;

let() = main() ;;
```

```
[18:08]cazzola@hymir:~/ocaml>main
Trial-Division's Primality Test         25 :- false
Trial-Division's Primality Test        127 :- true
Trial-Division's Primality Test       8191 :- true
```

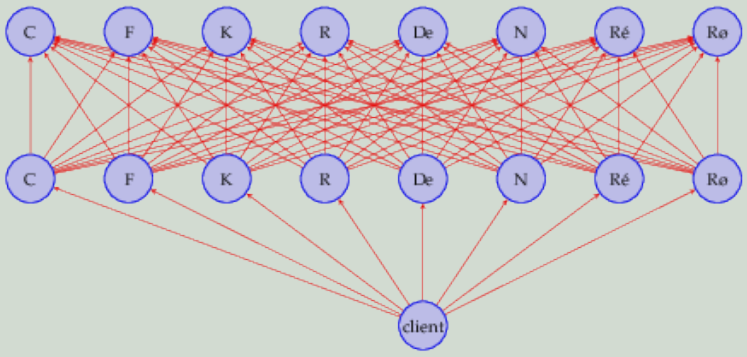
```
Lucas-Lehmer's Primality Test       131071 :- true
Lucas-Lehmer's Primality Test       524286 :- false
Lucas-Lehmer's Primality Test       524287 :- true
Little Fermat's Primality Test      524288 :- false
Little Fermat's Primality Test  2147483647 :- true
```

Exercise Erlang: You Are Hot!

Beyond the well-known Celsius and Fahrenheit, there are other six temperature scales: Kelvin, Rankine, Delisle, Newton, Réaumur, and Rømer. For your convenience, the conversion formulas from Celsius degrees (°C) are:

- Fahrenheit [°F] = [°C] × 9/5 + 32
- Kelvin [K] = [°C] + 273.15
- Rankine [°R] = ([°C] + 273.15) × 9/5
- Delisle [°De] = (100 − [°C]) × 3/2
- Newton [°N] = [°C] × 33/100
- Réaumur [°Ré] = [°C] × 4/5
- Rømer [°Rø] = [°C] × 21/40 + 7.5

To pass from one scale to another and vice versa is pretty simple but it would be interesting to have a distributed service that transform a temperature in any scale to another scale. Basically its structure should be compliant with this schema:



Where the nodes on the second row are those that from a given temperature scale translate the value to the Celsius scale and pass the result to the appropriate actor on the first row that translates it to the target scale. Each actor translates from or to a single scale; they all interact with the Celsius scale. Note that each actor has a different behavior but this **doesn't** mean that you have to implement so many different actors: be smart. Data go from the client to **one** actor in the second row then to **one** in the first row; the result follows the same route backward.

The exercise consists of implementing the whole structure in the module `tempsys` with a function `startsys/0` that bootstraps the whole system. The client is implemented in the module `client` that provides a single function `convert/5`. First and third parameters are always the atoms `from` and `to` respectively; the second and fourth arguments represent the source and target scale respectively as an atom representing the scale's symbol without the degree symbol.

The following is an example of the expected behavior:

```
[15:07]cazzola@hymir:~/erlang>erl
1> tempsys:startsys().
2> client:convert(from, 'Re', to, 'De', 25).
25°Re are equivalent to 103.125°De
3> client:convert(from, 'K', to, 'N', -25).
-25°K are equivalent to -98.38949999999998°N
4> client:convert(from, 'C', to, 'F', 3.5).
3.5°C are equivalent to 38.3°F
5> Conv = fun(X) -> client:convert(from, X, to, 'C', 32) end.
6> lists:map(Conv, ['C','De', 'F', 'K', 'N', 'R', 'Re', 'Ro']).
32°C are equivalent to 32°C
32°De are equivalent to 78.66666666666667°C
32°F are equivalent to 0.0°C
32°K are equivalent to -241.14999999999998°C
32°N are equivalent to 96.96969696969697°C
32°R are equivalent to -255.3722222222222°C
32°Re are equivalent to 40.0°C
32°Ro are equivalent to 46.666666666666664°C
7> Conv2 = fun(X) -> client:convert(from, 'C', to, X, 32) end.
8> lists:map(Conv2, ['C','De', 'F', 'K', 'N', 'R', 'Re', 'Ro']).
32°C are equivalent to 32°C
32°C are equivalent to 102.0°De
32°C are equivalent to 89.6°F
32°C are equivalent to 305.15°K
32°C are equivalent to 10.56°N
32°C are equivalent to 549.27°R
```

```
32°C are equivalent to 25.6°Re
32°C are equivalent to 24.3°Ro
```

All the solutions not conform to the specification will be considered wrong.

LogLang.

LogLang is a trivial DSL that permits to play with log files. The permitted operations on log files are:

- `remove` «filename», that remove the file associated to the given filename
- `rename` «filename.old» «filename.new», that changes the name to the file according with the passed data
- `merge` «file1» «file2» «file3», that copies the content of «file1» and «file2» in «file3»
- `backup` «file1» «file2», that copies the content of «file1» in «file2»

All the filenames are enclosed in quotes; quotes are not part of the filename.

The operations are grouped in tasks: `task «task name» { «operations» }`. Operations inside a task and the tasks are separated by a newline.

The exercise consists of implementing a DSL by using parser combinator techniques that can parse and execute LongLang scripts. Note that the operations must really affect the files and that these can fail due to many reasons, e.g., file missing. The interpreter should report on the failure or success of each single command as grouped in the script.

The following is an example of loglang script with the expected behavior (yielded using this set of logs).

```
task TaskOne {
  remove "application.debug.old"
  rename "application.debug" "application.debug.old"
}

task TaskTwo {
  backup "access.error" "security.logs"
  backup "system.error" "system.logs"
}

task TaskThree {
  merge "security.logs" "system.logs" "security+system.logs"
}
```

```
[18:13]cazzola@surtur:~/lp/scala>scala LogLangEvaluator test.ll
Task TaskOne
 [op1] true
 [op2] false
Task TaskTwo
 [op1] true
 [op2] true
Task TaskThree
 [op1] true
```