Principles of Programming Languages, 2018.01.16

Notes
- Total available time: 1h 30'
- You may use any written material you need, and write in Italian, if you prefer.
- You cannot use electronic devices during the exam.

Exercise 1, Scheme (11 pts)

Consider a procedure p that receives as input a list. Elements in the list are either numbers or strings, together with the two special separators * and $.

For instance, *L = (* 1 2 3 * $ "hello" * 1 * 7 "my" * 1 2 * "world" $).*

Implement p as a tail recursive function that sums all the numbers found between two occurrences of * symbols, and concatenates all the strings between occurrences of $, then returns the list of the resulting elements. Numbers and strings that are not between the correct pair of separators are discarded.

E.g., in the case of L, the result should be *(6 1 3 "hellomyworld")*.

Exercise 2, Haskell (11 pts)

Implement and state all the types of the following functions:

1) Define a fixpoint operator, which takes a function *f :: a -> a*, and an initial value of type *a*, and returns its fixed point, i.e. a value x such that f(x) = x. E.g. *fixpoint sqrt 12* should return 1.0.

2) We want to use our fixpoint operator to sets of values, but we also want to implement sets as regular lists (not a good idea in practice). What kind of data declaration we need, if any? We need to make it an instance of standard classes? If the answer is yes, do it. If it is no, motivate your answer.

3) Define *setminus*, *intersection*, and *union* for such sets.

Exercise 3, Erlang (10 pts)

Consider the following Erlang Code:

```
main() ->
    X = spawn(fun() -> logger() end),
    [spawn(fun() -> loggee(X, Y) end) || Y <- ["A","B","A","B","A","B"]].

logger() ->
    receive
        {log, K, Y} ->
            io:format("log: ~w ~p~n", [K, Y]),
            logger()
    end.

loggee(X, Y) ->
        receive
        after rand:uniform(100) ->
                X ! {log, self(), Y}
        end.
```

1) Describe what the the code does, and give more descriptive names to the variables called X,Y,K.

(see back)

2) Are the following output traces possible? Why?

(1)
```
log: <0.87.0> "B"
log: <0.86.0> "A"
log: <0.88.0> "A"
log: <0.89.0> "B"
log: <0.84.0> "A"
log: <0.85.0> "B"
```

(2)
```
log: <0.88.0> "A"
log: <0.89.0> "B"
log: <0.84.0> "A"
log: <0.87.0> "B"
log: <0.86.0> "A"
log: <0.85.0> "B"
```

3) Can we modify the code to guarantee (2) to happen? (Of course, excluding actual PID values.) Do it.

Solutions

```
Es 1
(define (p lst)
  (define (loc-p ls res cint cstr)
    (if (null? ls)
      res
      (let ((cur (car ls)))
        (cond
          ((eq? cur '*)
           (if cint
             (loc-p (cdr ls) (append res (list cint)) #f cstr)
             (loc-p (cdr ls) res 0 cstr)))
          ((eq? cur '$)
           (if cstr
             (loc-p (cdr ls) (append res (list cstr)) cint #f)
             (loc-p (cdr ls) res cint "")))
          ((and cint (number? cur))
           (loc-p (cdr ls) res (+ cur cint) cstr))
          ((and cstr (string? cur))
           (loc-p (cdr ls) res cint (string-append cstr cur)))
          (else
            (loc-p (cdr ls) res cint cstr))
          ))))
  (loc-p lst '() #f #f))

Es 2
fixpoint :: Eq t => (t -> t) -> t -> t
fixpoint f i = let v = f i
               in if v == i then i else fixpoint f v

-- the simplest way is to make lists a set, with a newtype:
newtype Lset a = Lset [a] deriving Show

setminus :: Eq a => Lset a -> Lset a -> Lset a
setminus (Lset x) (Lset y) = Lset [t | t <- x, not (elem t y)]

intersection :: Eq a => Lset a -> Lset a -> Lset a
intersection (Lset x) (Lset y) = Lset [t | t <- x, elem t y]

union :: Lset a -> Lset a -> Lset a
union (Lset x) (Lset y) = Lset $ x ++ y

-- to use fixpoint, we need a special version of == (of course the standard eq of lists does not work)
instance (Eq a) => Eq (Lset a) where
  x == y = setminus x y == Lset [] && setminus y x == Lset []
```

Es 3

The code implements a master logging actor and a set of other actors which send to the master their logging messages.

Variable naming X=Pid/Logger, Y=Type, K=Pid/From

Both the output traces are possible, since there is no guarantees regarding the ordering, as Erlang maintains a FIFO ordering of messages only between two processes.

With the current code is not possible to guarantee any order of execution.
A possible one extends the logger to alternatively receive an A and the a B.

```
logger2(Next) ->
    receive
        {log, K, Next} ->
            io:format("log: ~w ~p~n", [K, Next]),
            if
                Next =:= "B" -> logger2("A");
                true -> logger2("B")
            end
    end.
```