Walter Cazzola

Home Page
ADAPT Lab.
Curriculum Vitae
Research Topic

Exam of Programming Languages

15 June 2016

Exercise ML/OCaML: A «Continued» QuickSort.

```
let cqsort  (>:) l =
   let rec cqsort l k = match l with
     ([] | [_]) -> k l
   | h :: t -> cqsort
       (List.filter (fun x -> h >: x) t)
       (fun ll -> k (cqsort
            (List.filter (fun x -> x >: h) t)
            (fun gl -> (ll @ [h]) @ gl)))
   in cqsort l (fun x -> x)
```

Exercise Erlang: Joseph's Problem.

```
-module(hebrew).
-export([start/3]).

start(Label, N, K) ->
   receive
     {neighbor, PID, master, M} -> loop(Label, PID, M, N, K);
     Other -> io:format("### error ~p~n", [Other])
   end.

loop(Label,Next,Master,N,K) ->
   receive
     % Your neighbor is died! Long live to the new neightbor!
     {newn, NewNext} -> loop(Label, NewNext, Master, N, K);
     % You are the last standing! Communicate it to the master!
     {msg, _, from, _, stepn, N, stepk, 1} ->
         Master ! {msg, "I'm the survivor!", from, self(), label, Label};
     % This is not your lucky day! You drawn the shortest match.
     {msg, MSG, from, Prev, stepn, Sn, stepk, K} ->
         Prev!{newn,Next}, Next!{msg,MSG,from,Prev,stepn,Sn+1,stepk,1};
     % Phwee!
     {msg, MSG, from, _, stepn, Sn, stepk, Sk} ->
         Next! {msg, MSG, from, self(), stepn, Sn, stepk, Sk+1},
         loop(Label, Next, Master, N, K)
   end.
```

```
-module(joseph).
-export([joseph/2]).

joseph(N,K) ->
  PIDS = [spawn(hebrew, start, [G, N, K]) || G <- lists:seq(1,N)],
  [P! {neighbor, Next, master, self()} ||
      {P, Next} <- lists:zip(PIDS, tl(PIDS)++[hd(PIDS)])],
  hd(PIDS) ! {msg, "Who will survive?", from,
            lists:last(PIDS), stepn, 1, stepk, 1},
  receive
     {msg, "I'm the survivor!", from, _, label, L} ->
         io:format("In a circle of ~p people, killing number ~p~n
             Joseph is the Hebrew in position ~p~n", [N, K, L]);
     Other -> io:format("### error ~p~n", [Other])
  end.
```

Exercise Scala: ArnoldC.

```
import scala.util.parsing.combinator._
import scala.collection.mutable._

class ArnoldCCombinators(var the_stack: Stack[Int],
      var the_table: HashMap[String,Int]) extends JavaTokenParsers {
   def arnoldc_program = "IT'S" ~> "SHOW" ~> "TIME" ~> arnoldc_body <~
        "YOU" <~ "HAVE" <~ "BEEN" <~ "TERMINATED"
            ^^ {_ => (the_stack,the_table)}
   def arnoldc_body:Parser[Any] = rep(statement)
   def statement = var_def | assignment | printing | ifthenelse | loop
   def printing = "TALK" ~> "TO" ~> "THE" ~> "HAND" ~>
```

```
        (idvalue|stringLiteral) ^^ { println(_) }
    def var_def = "HEY" ~> "CHRISTMAS" ~> "TREE" ~> ident ~
        ("YOU" ~> "SET" ~> "US" ~> "UP"~> intvalue)
        ^^ { case s~n => the_table(s) = n; }
    def intvalue = wholeNumber  ^^ { n => n.toInt }
    def initial_value = (intvalue|idvalue) ^^ {n => the_stack.push(n)}
    def a_value:Parser[Int] = intvalue|idvalue
    def block = """(?s)\[.*?\]""".r ^^ {s => s.substring(1,s.length-1)}
    def ifthenelse =
          "BECAUSE" ~> "I'M" ~> "GOING" ~> "TO" ~> "SAY" ~> "PLEASE"
          ~ idvalue ~ block ~
          ("BULLSHIT" ~> block <~ "YOU" <~ "HAVE" <~ "NO" <~ "RESPECT"
          <~ "FOR" <~ "LOGIC") ^^ {
             case e~b1~b2 =>
               if (e != 0) parseAll(arnoldc_body, b1)
               else parseAll(arnoldc_body, b2)
          }
    def loop = "STICK" ~> "AROUND" ~> ident ~ block <~ "CHILL" ^^ {
       case id ~ b1 => while (the_table(id) !=0) parseAll(arnoldc_body, b1)
    }
    def exprs = expr ~ rep(expr)
    def expr = (arithmetic|logic)
       ^^ { (f:(Int => Int)) => the_stack.push(f(the_stack.pop)) }
    def arithmetic = (
         "GET" ~> "UP" ~> a_value ^^ { n => (a: Int) => a+n }
       | "GET" ~> "DOWN" ~> a_value ^^ { n => (a: Int) => a-n }
       | "YOU'RE" ~> "FIRED" ~> a_value ^^ { n => (a: Int) => a*n }
       | "HE" ~> "HAD" ~> "TO" ~> "SPLIT" ~> a_value ^^ { n => (a: Int) => a/n }
    )
    def logic = (
         "YOU" ~> "ARE" ~> "NOT" ~> "YOU" ~> "YOU" ~> "ARE" ~> "ME" ~> a_value
             ^^ {n => (a:Int) => if (n==a) 1 else 0}
       | "LET" ~> "OFF" ~> "SOME" ~> "STEAM" ~> "BENNET" ~> a_value
             ^^ {n => (a:Int) => if (a>n) 1 else 0}
       | "CONSIDER" ~> "THAT" ~> "A" ~> "DIVORCE" ~> a_value
             ^^ {n => (a:Int) => a*n}
       | "KNOCK" ~> "KNOCK" ~> a_value
             ^^ {n => (a:Int) => a+n}
    )
    def idvalue = ident ^^ { the_table(_) }
    def assignment = "GET" ~> "TO" ~>"THE" ~> "CHOPPER" ~> ident <~
        ("HERE" ~> "IS" ~> "MY" ~> "INVITATION" ~> initial_value)
        <~ exprs <~ "ENOUGH" <~ "TALK" ^^ {
       case s => the_table(s) = the_stack.pop
    }
}

object ArnoldCEvaluator {
  def main(args: Array[String]) = {
     val p = new ArnoldCCombinators(new Stack[Int](), new HashMap[String, Int]())

     args.foreach { filename =>
        val src = scala.io.Source.fromFile(filename)
        val lines = src.mkString
        p.parseAll(p.arnoldc_program, lines) match {
            case p.Success((s,t),_) =>
               println(s)
               println("Symbol Table :-")
               t foreach { m => println("\t"+m) }
            case x => print(x.toString)
        }
        src.close()
     }
  }
}
```