Walter Cazzola

Home Page
ADAPT Lab.
Curriculum Vitae
Research Topic

Exam of Programming Languages

29 January 2015

Exercise OCaML/ML: Playing with Numbers.

```
module Natural =
  struct
    type natural = Zero | Succ of natural
    exception NegativeNumber
    exception DivisionByZero

    let rec ( + ) n = function
       Zero    -> n
    |  Succ(m) -> (+) (Succ(n)) m

    let (>) a b =
      match a, b with
         Succ(_), Zero -> true
      |  Zero, Succ(_)
      |  Zero, Zero    -> false
      |  Succ(n), Succ(m) -> (>) n m

    let rec ( - ) n m =
      if not (m > n) then
        match n, m with
           n', Zero           -> n'
        |  Succ(n'), Succ(m') -> (-) n' m'
      else raise NegativeNumber

    let ( * ) n m =
      if n > Zero then
        let rec ( * ) r n = function
          Zero       -> Zero
        | Succ(Zero) -> r
        | Succ(m)    -> ( * ) ((+) r n) n m
        in ( * ) n n m
      else Zero

    let ( / ) n m =
      if n > Zero then (
        if not (m > Zero) then raise DivisionByZero
        else
          let rec ( / ) r n m =
            if not (m > n) then ( / ) (Succ r) ((-) n m) m
            else r
          in ( / ) Zero n m )
      else Zero

    let rec eval = function
       Zero       -> 0
    |  Succ(n)    -> succ (eval n)

    let convert n =
      let rec convert r n =
        if (0  < n ) then convert (Succ(r)) ( pred n )
        else r
      in convert Zero n
  end;;

module N = (Natural: NaturalI.NaturalI) ;;
```

Exercise Erlang: Distributed Combinatorics.

```
-module(combinator).
-export([start/2]).

start(N, M) ->
  register(entrypoint, spawn(fun() -> init_slaves(N,M) end)).

init_slaves(N, M) ->
  [spawn_link(generator, init, [P, N, M]) || P <- lists:seq(1,N)],
```

```
  collect(N,M,1,1,trunc(math:pow(M, N))).

collect(N, M, P, Seq, Max) ->
   receive
     {seq, Seq, val, Value, pos, P} when (Seq==Max) and ((P+1)>N) ->
         io:format("~p~n", [Value]), exit(entrypoint), unregister(entrypoint);
     {seq, Seq, val, Value, pos, P} when ((P+1)>N) ->
         io:format("~p~n", [Value]), collect(N,M,1,Seq+1,Max);
     {seq, Seq, val, Value, pos, P} ->
         io:format("~p,", [Value]), collect(N,M,P+1,Seq,Max)
   end.
```

```
-module(generator).
-export([init/3]).

% P is the processor position in the sequence,
% N is the number of processors in the system and
% M the number of possible values
init(P, N, M) ->
  counter(1, trunc(math:pow(M, (N-P))), trunc(math:pow(M, N)), 1, P, N, M).

% The counter advances after M^(N-P) ticks
% Seq is the iteration number
% Delay downcounts from the number of ticks needed before increasing the counter
% Value is the current value of the counter, if greater than M the counter stops
% T is the total number of iterations
counter(Seq, _, T, _, _, _, _) when Seq > T -> stop;
counter(Seq, _, T, Value, P, N, M) when (Value > M) ->
   counter(Seq, trunc(math:pow(M, (N-P))), T, 1, P, N, M);
counter(Seq, 1, T, Value, P, N, M) ->
   entrypoint ! {seq, Seq, val, Value, pos, P},
   counter(Seq+1, trunc(math:pow(M, (N-P))), T, Value+1, P, N, M);
counter(Seq, Delay, T, Value, P, N, M) ->
   entrypoint ! {seq, Seq, val, Value, pos, P},
   counter(Seq+1, Delay-1, T, Value, P, N, M).
```

Tables' Pretty Printing.

```
import util.parsing.combinator.RegexParsers

trait CSVParser extends RegexParsers {
  override val skipWhitespace = false
  override val whiteSpace = """[ \t]""".r

  var row_sizes: List[Int] = Nil
  var tmp_sizes: List[Int] = Nil
  var firstTime = true
  def file: Parser[String] = hdr ~! rep1(row) ^^ {
    case header ~ rows =>
      val row_len = (3*row_sizes.length+1+row_sizes./:(0)(_ + _))
      val str_format = row_sizes.map(n =>
            "| %%-%ds ".format(n)).:\[String]("|\n")(_+_)
            "-"*row_len+"\n"+str_format.format(header.toSeq:_*)+"-"*row_len+"\n"+
            rows.map(r => str_format.format(r.toSeq:_*)).:\[String]("")(_+_)+
            "-"*row_len+"\n"
  }
  def hdr: Parser[List[String]] = row
  def row: Parser[List[String]] = repsep(field, ",") <~ """\r""".? <~ "\n" ^^
    {s =>
       if (firstTime) {
           row_sizes = tmp_sizes
           firstTime = false
       } else row_sizes = (row_sizes, tmp_sizes).zipped map (_ max _);
       tmp_sizes = Nil
       s
     }

  def field: Parser[String] = (TEXT ||| STRING | EMPTY) ^^ {
     s => tmp_sizes = tmp_sizes :+ s.length(); s }

  lazy val TEXT: Parser[String] = rep1("""[^,\n\r\"]""".r) ^^ makeText
  lazy val STRING: Parser[String] =
     whiteSpace.* ~> "\"" ~> rep("\"\"" | """[^\"]""".r)
        <~ "\"" <~ whiteSpace.* ^^ makeString
  lazy val EMPTY: Parser[String] = "" ^^ makeEmpty

  def makeText: (List[String]) => String
  def makeString: (List[String]) => String
  def makeEmpty: String => String
}
```

```
trait CSVParserAction {
  // remove leading and trailing blanks
  def makeText = (text: List[String]) => text.mkString("").trim
  // remove embracing quotation marks
  // replace double quotes by single quotes
  def makeString =
    (string: List[String]) => string.mkString("").replaceAll("\"\"", "\"")
  // modify result of EMPTY token if required
  def makeEmpty = (string: String) => ""
}

object CSVParserCLI {
  def main(args: Array[String]) {

    args.foreach { filename =>
       val p = new CSVParser with CSVParserAction
       val src = scala.io.Source.fromFile(filename)
       val lines = src.mkString

       p.parseAll(p.file, lines) match {
           case p.Success(t,_) =>
              println(t)
           case x => print(x.toString)
       }
       src.close()
     }
  }
}
```