Walter Cazzola

Home Page
ADAPT Lab.
Curriculum Vitae
Research Topic

Exam of Programming Languages

26 February 2015

Exercise ML/OCaML: Expressions Solved Step by Step.

```
module ArithExpr =
  struct
    type expr = Sum of expr * expr | Minus of expr * expr | Mul of expr * expr |
                Div of expr *expr | Number of (float) ;;

    let rec tostring = function
      Number(n) -> string_of_float n
    | Sum(n1,n2) -> String.concat " " ["("; (tostring n1); "+"; (tostring n2); ")"]
    | Minus(n1,n2) -> String.concat " " ["("; (tostring n1); "-"; (tostring n2); ")"]
    | Mul(n1,n2) -> String.concat " " ["("; (tostring n1); "*"; (tostring n2); ")"]
    | Div(n1,n2) -> String.concat " " ["("; (tostring n1); "/"; (tostring n2); ")"] ;;

    let rec combine = function
      Number(n) -> Number(n)
    | Sum(Number(n1), Number(n2)) -> Number(n1+.n2)
    | Sum(n1,n2) -> Sum((combine n1), (combine n2))
    | Minus(Number(n1), Number(n2)) -> Number(n1-.n2)
    | Minus(n1,n2) -> Minus((combine n1), (combine n2))
    | Mul(Number(n1), Number(n2)) -> Number(n1*.n2)
    | Mul(n1,n2) -> Mul((combine n1), (combine n2))
    | Div(Number(n1), Number(n2)) -> Number(n1/.n2)
    | Div(n1,n2) -> Div((combine n1), (combine n2))

    let rec print_reduction e =
      print_endline (tostring e);
      match e with
        Number(n) ->  ()
      | e -> print_reduction (combine e) ;;

    let parse str =
      let rec parse str n =
        match str.[n] with
          '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9' ->
            Number(float_of_string  ((String.make 1 str.[n]) ^".")), n
        | '+' ->
          let op1,n1 = (parse str (n+1)) in let op2,n2 = (parse str (n1+1)) in Sum(op1 op2   n2
        | '-' ->
          let op1,n1 = (parse str (n+1)) in let op2,n2 = (parse str (n1+1)) in Minus(op1 op2   n2
        | '*' ->
          let op1,n1 = (parse str (n+1)) in let op2,n2 = (parse str (n1+1)) in Mul(op1 op2   n2
        | '/' ->
          let op1,n1 = (parse str (n+1)) in let op2,n2 = (parse str (n1+1)) in Div(op1 op2   n2
      in fst (parse str 0);;

    let print_evaluation str = print_reduction (parse str) ;;

  end;;
```

Exercise Erlang: Hamiltonian Path on an Hypercube.

```
-module(hypercube).
-export([create/0,hamilton/2,gray/1]).

create() ->
  PIDs = [{G, spawn(node, start, [G])} || G <- gray(4)],
  lists:foreach(
    fun({G,Ns}) ->
      {_,P} = lists:keyfind(G, 1, PIDs),
      P! {neighbors, pair_pids(PIDs, Ns), src, self()}
    end, grayneighbors(gray(4))),
  {_,P} =lists:keyfind("0000", 1, PIDs),
  register(zero, P).

hamilton(M, [HG|TG]) ->
   zero ! {msg, {src, HG, msg, M}, path, TG},
   receive
     Other -> io:format("~p~n", [Other])
   end.

gray(0) -> [""];
gray(N) -> G = gray(N-1), [ "0"++L || L <- G ]++[ "1"++L || L <- lists:reverse(G) ].

strxor([],[]) -> [];
strxor([H1|T1],[H2|T2]) -> [(H1 bxor H2)+$0 |strxor(T1,T2)].
```

```
neighbors(Lab) -> [strxor(X,N) || {X,N} <-
   lists:zip([X || X <- [Lab], _ <-lists:seq(0,3)],
      ["1000", "0100", "0010", "0001"])] .

grayneighbors([]) -> [];
grayneighbors([H|T]) -> [{H,neighbors(H)}]++grayneighbors(T).

pair_pids(_, []) -> [];
pair_pids(PIDs, [H|T]) -> {H,P} = lists:keyfind(H,1,PIDs),[{H,P}]++pair_pids(PIDs, T).
```

```
-module(node).
-export([start/1]).

start(G) ->
   io:format("The process labeled ~p just started~n", [G]),
   receive
     {neighbors, PIDs, src, S} ->
        loop(G, PIDs, S);
     Other -> io:format("### error ~p~n", [Other])
   end.

loop(G,Ns,S) ->
   receive
     {msg, M, path, [HP|[]]} -> S ! {msg, {src, HP, msg, M}}, loop(G,Ns,S);
     {msg, M, path, [HP|TP]} ->
        case lists:keyfind(HP, 1, Ns) of
          {HP, P} -> P! {msg, {src, HP, msg, M}, path, TP}, loop(G,Ns,S);
          Other -> io:format("~p~n", [Other])
        end
   end.
```

Exercise Scala: Expressions Solved Step by Step.

```
import scala.util.parsing.combinator._
import  scala.util.matching.Regex._
import  scala.util.matching._

trait Expr {
  def eval: Int
}

case class IntLeaf(n: Int) extends Expr {
  def eval = n
  override def toString = "%d".format(n)
}

case class Sum(a: Expr, b: Expr) extends Expr {
  def eval = a.eval + b.eval
  override def toString = "(%s + %s)".format(a, b)
}

case class Minus(a: Expr, b: Expr) extends Expr {
  def eval = a.eval - b.eval
  override def toString = "(%s - %s)".format(a, b)
}

case class Mul(a: Expr, b: Expr) extends Expr {
  def eval = a.eval * b.eval
  override def toString = "(%s * %s)".format(a, b)
}

case class Div(a: Expr, b: Expr) extends Expr {
  def eval = a.eval / b.eval
  override def toString = "(%s / %s)".format(a, b)
}

object ReductionUtil {
   def combineLeaves(e: Expr): Expr = {
     e match {
       case IntLeaf(n) => IntLeaf(n)
       case Sum(IntLeaf(a), IntLeaf(b)) => IntLeaf(a + b)
       case Sum(a, b) => Sum(combineLeaves(a), combineLeaves(b))
       case Minus(IntLeaf(a), IntLeaf(b)) => IntLeaf(a - b)
       case Minus(a, b) => Minus(combineLeaves(a), combineLeaves(b))
       case Mul(IntLeaf(a), IntLeaf(b)) => IntLeaf(a * b)
       case Mul(a, b) => Mul(combineLeaves(a), combineLeaves(b))
       case Div(IntLeaf(a), IntLeaf(b)) => IntLeaf((a/b).toInt)
       case Div(a, b) => Div(combineLeaves(a), combineLeaves(b))
     }
   }

   def printReduction(e: Expr) {
     println(e)
     e match {
       case IntLeaf(n) =>
       case _ => printReduction(combineLeaves(e))
     }
```

```
    }
}

class ArithmeticParser extends RegexParsers {
  def int: Parser[IntLeaf] = regex(new Regex("""\d+""")).map(s => IntLeaf(s.toInt))
  def sum: Parser[Sum] =
        ("(" ~> expr ~ "+" ~ expr <~ ")").map { case (a ~ _ ~ b) => Sum(a, b) }
  def minus: Parser[Minus] =
      ("(" ~> expr ~ "-" ~ expr <~ ")").map { case (a ~ _ ~ b) => Minus(a, b) }
  def mul: Parser[Mul] =
        ("(" ~> expr ~ "*" ~ expr <~ ")").map { case (a ~ _ ~ b) => Mul(a, b) }
  def div: Parser[Div] =
        ("(" ~> expr ~ "/" ~ expr <~ ")").map { case (a ~ _ ~ b) => Div(a, b) }
  def expr = int | sum | minus | mul | div
}

object StepByStepEvaluator {
  def main(args: Array[String]) = {
     val p = new ArithmeticParser

     args.foreach { expression =>
        p.parseAll(p.expr, expression) match {
            case p.Success(result: Expr, _) => ReductionUtil.printReduction(result);println
            case x => sys.error("Could not parse the input string: " + x.toString)
        }
     }
  }
}
```