```scala
package fr.istic.cal.interpreter


/**
 * définition d'une exception pour le cas des listes vides
 */
case object ExceptionListeVide extends Exception

/**
 * définition d'une exception pour le cas des listes de tailles
différentes
 */
case object ExceptionListesDeLongueursDifferentes extends Exception

object Interpreter {

  /**
   * UN INTERPRETER POUR LE LANGAGE WHILE
   *
   */

  /**
   *  GESTION DE LA MEMOIRE DE L'INTERPRETEUR
   */

  /**
   *  définition d'un type Memory pour représenter une mémoire
   */
  type Memory = List[(Variable, Value)]

  /**
   * @param v : une variable
   * @param mem : une mémoire
   * @return m(v), c'est-à-dire la valeur de la variable v dans la
mémoire mem,
   * la valeur par défaut si la variable v n'est pas présente dans la
mémoire mem
   */
  // TODO TP2
  def lookUp(v: Variable, mem: Memory): Value = {
    (mem) match {
      case (Nil) => NlValue
      case (hd :: tl) => if (hd._1 == v) hd._2
      else (lookUp(v, tl))
    }
```

```
  }

  /*
   * @param v : une variable
   * @param d : une valeur
   * @param mem : une mémoire
   * @return la mémoire modifiée par l'affectation [v->d]
   */
  // TODO TP2
  def assign(v: Variable, d: Value, mem: Memory): Memory = {
    mem match {
      case Nil => List((v, d))
      case (v1, d1) :: tl =>
        if (v1 == v) (v, d) :: tl
        else (v1, d1) :: assign(v, d, tl)
    }
  }

  /**
   *  TRAITEMENT DES EXPRESSIONS DU LANGAGE WHILE
   */

  /**
   * @param expression : un AST décrivant une expression du langage
WHILE
   * @return la valeur de l'expression
   */
  // TODO TP2
  def interpreterExpr(expression: Expression, mem: Memory): Value = {
    expression match {
      case Nl                                    => NlValue
      case Cst(name: String)                     => CstValue(name)
      case VarExp(name: String)                  => lookUp(Var(name),
mem)
      case Cons(arg1: Expression, arg2: Expression) =>
ConsValue(interpreterExpr(arg1, mem), interpreterExpr(arg2, mem))
      case Hd(arg: Expression) => interpreterExpr(arg, mem) match {
        case ConsValue(val1, _) => val1
        case _                  => NlValue
      }
      case Tl(arg: Expression) => interpreterExpr(arg, mem) match {
        case ConsValue(_, val2) => val2
        case _                  => NlValue
      }
      case Eq(arg1: Expression, arg2: Expression) => {
        if (interpreterExpr(arg1, mem) == interpreterExpr(arg2, mem))
```

```
CstValue("true") else NlValue
      }
      // Td5 Ex2
      case Permute(arg) =>
        interpreterExpr(arg,mem) match{
          case ConsValue(a,b) => ConsValue(b,a)
          case x => x
      }
      case Miroir(arg)=>
        interpreterExpr(arg,mem) match{
          case ConsValue(a,b) =>
ConsValue(interpreterExpr(Miroir(Tl(arg)),mem),interpreterExpr(Miroir(Hd
(arg)),mem))
           case x => x
        }
    }
  }

  /**
   * la fonction interpreterExpr ci-dessus calcule la valeur associée à
une expression
   * il peut être utile de produire à l'inverse une expression associée
à une valeur
   * la fonction valueToExpression ci-dessous construira l'expression la
plus simple associée à une valeur
   *
   * @param value : une valeur du langage WHILE
   * @return l'AST décrivant l'expression de cette valeur
   */
  // TODO TP2
  def valueToExpression(value: Value): Expression = {
    value match {
      case NlValue                              => Nl
      case CstValue(name: String)               => Cst(name)
      case ConsValue(arg1: Value, arg2: Value) =>
Cons(valueToExpression(arg1), valueToExpression(arg2))
    }
  }

  /**
   *
   *  TRAITEMENT DES COMMANDES DU LANGAGE WHILE
   */

  /**
   * @param command : un AST décrivant une commande du langage WHILE
```

```
   * @param memory : une mémoire
   * @return la mémoire après l'interprétation de command
   */
  // TODO TP2
  def interpreterCommand(command: Command, memory: Memory): Memory = {
    command match {
      case Nop => memory
      case Set(variable: Variable, expression: Expression) =>
assign(variable, interpreterExpr(expression, memory), memory)
      case While(condition: Expression, body: List[Command]) =>
        interpreterExpr(condition, memory) match {
          case NlValue => memory
          case _       => interpreterCommand(command,
interpreterCommands(body, memory))
        }
      /*Version 2 :
       * val value = interpreterExpr(condition, memory)
        condition match {
          case VarExp(name) =>
            for (c <- body) {
              c match {
                case Set(Var(n), expression) =>
                  if (n == name) {
                    val newMemory = assign(Var(n),
interpreterExpr(condition, memory), memory)
                    assign(Var(name), NlValue, newMemory)
                  }
              }
            }
            memory
        }*/
      /*val value = interpreterExpr(condition,memory)
        value match {
           case  NlValue => memory // quand condition est faux,ne passe
pas au boucle
           case CstValue("true") =>// quand condition est vrai

                for (c <- body){
                  c match {
                    case Set(Var(n), expression) =>
                      if(n==name){
                      val newMemory =
assign(Var(n),interpreterExpr(condition,memory),memory)
                      assign(Var(name),NlValue,newMemory)
                      }
                  }
```

```
              }
            memory}
            *
            */
      //Version 0:
assign(Var(name),interpreterExpr(condition,memory),newMemory)
      //FIXME case when commands in body change value of Var in
condition
      /*Version 1:
                * val newMemory = interpreterCommands(body, memory)
            condition match {
              case VarExp(name) =>
                  for (c <- body){
                  c match {
                    case Set(Var(n), expression) =>
                      if(n==name){
                      val condiMemory =
assign(Var(n),interpreterExpr(condition,memory),memory)

assign(Var(name),interpreterExpr(expression,memory),condiMemory)
                      }
                  }


                }
                memory
            }
        } */
      case For(count: Expression, body: List[Command]) =>
        val value = interpreterExpr(count, memory)
        value match {
          case NlValue     => memory //passe pas au boucle
          case CstValue(_) => interpreterCommands(body, memory) //pas
sur,reste 1 boucle
          case ConsValue(arg1, arg2) =>
            interpreterCommand(
              For(valueToExpression(arg2), body),
              interpreterCommand(For(valueToExpression(arg1), body),
interpreterCommands(body, memory)))
        }

      case If(condition: Expression, then_commands: List[Command],
else_commands: List[Command]) => {
        val value = interpreterExpr(condition, memory)
        value match {
          case NlValue => interpreterCommands(else_commands, memory) //
```

```
quand condition est faux
          case _       => interpreterCommands(then_commands, memory) //
tout le reste c'est quand condition est vrai
        }
      }
      // Td5 Ex1
      case Repeat(body,expr) =>
        val memAfterBody = interpreterCommands(body,memory)
        interpreterExpr(expr,memAfterBody) match {
          case NlValue => interpreterCommand(command,memAfterBody)
          case _ => memAfterBody
        }
    }
  }

  /**
   * @param commands : une liste non vide d'AST décrivant une liste non
vide de commandes du langage WHILE
   * @param memory : une mémoire
   * @return la mémoire après l'interprétation de la liste de commandes
   */
  // TODO TP2
  def interpreterCommands(commands: List[Command], memory: Memory):
Memory = {
    commands match {
      case command1 :: Nil   => interpreterCommand(command1, memory)
      case command1 :: reste => interpreterCommands(reste,
interpreterCommand(command1, memory))
    }
  }

  /**
   *
   *  TRAITEMENT DES PROGRAMMES DU LANGAGE WHILE
   */

  /**
   * @param vars : une liste non vide décrivant les variables d'entrée
d'un programme du langage WHILE
   * @param vals : une liste non vide de valeurs
   * @return une mémoire associant chaque valeur à la variable d'entrée
correspondant
   */
  // TODO TP2
  def interpreterMemorySet(vars: List[Variable], vals: List[Value]):
Memory = {
```

```
    (vars, vals) match {
      case (var1 :: Nil, val1 :: Nil)           => (var1, val1) :: Nil
      case (var1 :: resteVar, val1 :: resteVal) => (var1, val1) ::
interpreterMemorySet(resteVar, resteVal)
    }
  }
  /**
   * @param vars : une liste non vide décrivant les variables de sortie
d'un programme du langage WHILE
   * @param memory : une mémoire
   * @return la liste des valeurs des variables de sortie
   */
  // TODO TP2
  def interpreterMemoryGet(vars: List[Variable], memory: Memory):
List[Value] = {
    if (vars.isEmpty) throw ExceptionListeVide
    vars.flatMap(v => memory.collectFirst { case (`v`, value) => value
})
  }
    // Version ultra : vars.flatMap(v => memory.collectFirst { case
(`v`, value) => value })
    /*vars match {
      case var1 :: Nil => {
        for (element <- memory) {
          if (element._1 == var1) {
            return List(element._2)
          }
        }
        Nil
      }
      case var1 :: resteVar => {
        for (element <- memory) {
          if (element._1 == var1) {
            return element._2 :: interpreterMemoryGet(resteVar, memory)
// return pour arrêter le boucle and passer au récursion quand
//l'élement est trouvé
          }
        }
        Nil // retourne une lsite vide si l'élément n'est pas dans le
memory
      }
    }
  }*/
  /* vars match {
      case var1 :: Nil => {
        for (element <- memory) {
```

```
          element match {
            case var1 => element._2::Nil
          }
        }
       Nil
      }
      case var1 :: resteVar =>
        {
          for (element <- memory) {
            element match {
              case var1 => element._2 :: interpreterMemoryGet(resteVar,
memory) //l'élement est trouvé
            }
          }
        }
        Nil // retourne une lsite vide si l'élément n'est pas dans le
memory
    }*/
  /**
   * @param program : un AST décrivant un programme du langage WHILE
   * @param vals : une liste de valeurs
   * @return la liste des valeurs des variables de sortie
   */
  // TODO TP2
  def interpreter(program: Program, vals: List[Value]): List[Value] = {
    program match {
      case Progr(in: List[Variable], body: List[Command], out:
List[Variable]) =>
        val memory = interpreterMemorySet(in, vals: List[Value]) //
création d'un memory à partir d'entrée / initialisation
        //de memory
        val memoire = interpreterCommands(body, memory) // nouvelle
mémoire aprés l'interprétation
        interpreterMemoryGet(out, memoire) // récupération de nouvelle
list de valeurs à
      //partir de nouvelle mémoire
    }
  }
  /**
   * UTILISATION D'UN ANALYSEUR SYNTAXIQUE POUR LE LANGAGE WHILE
   *
   * les 3 fonctions suivantes permettent de construire un arbre de
syntaxe abstraite
   * respectivement pour une expression, une commande, un programme
   */
```

```
  /**
   * @param s : une chaine de caractères représentant la syntaxe
concrète d'une expression du langage WHILE
   * @return un arbre de syntaxe abstraite pour cette expression
   */
  def readWhileExpression(s: String): Expression = {
WhileParser.analyserexpression(s) }

  /**
   * @param s : une chaine de caractères représentant la syntaxe
concrète d'une commande du langage WHILE
   * @return un arbre de syntaxe abstraite pour cette commande
   */
  def readWhileCommand(s: String): Command = {
WhileParser.analysercommand(s) }

  /**
   * @param s : une chaine de caractères représentant la syntaxe
concrète d'un programme du langage WHILE
   * @return un arbre de syntaxe abstraite pour ce programme
   */
  def readWhileProgram(s: String): Program = {
WhileParser.analyserprogram(s) }

}
```