```scala
package fr.istic.cal.prettyprinter

import scala.util.Try

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

object Prettyprinter {

  /**
   * UN PRETTY-PRINTER POUR LE LANGAGE WHILE
   *
   */

  /**
   * définition d'un type pour les spécifications d'indentation
   */
  type IndentSpec = List[(String, Int)]

  /**
   * définition d'une valeur d'indentation par défaut
   */
  val indentDefault: Int = 1

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

  /**
   * @param expression : un AST décrivant une expression du langage
WHILE
   * @return une chaîne représentant la syntaxe concrète de l'expression
   */

  def prettyPrintExpr(expression: Expression): String = {
    expression match {
      case Nl => "nil"
      case Cst(name) => name
      case VarExp(name) =>   name
      case Cons(arg1, arg2) => "(cons " + prettyPrintExpr(arg1) + " " +
prettyPrintExpr(arg2) + ")"
      case Hd(darg) => "(hd " + prettyPrintExpr(darg) + ")"
      case Tl(arg) => "(tl " + prettyPrintExpr(arg) + ")"
```

```
      case Eq(arg1, arg2) =>  prettyPrintExpr(arg1) + " =? " +
prettyPrintExpr(arg2)
    }
  }

/*
    def prettyPrintExpr(expression: Expression): String = {
    expression match {
      case Nl => "nil"
      case Cst(name: String) => name
      case VarExp(name: String) =>   name
      case Cons(arg1: Expression, arg2: Expression) => "(cons " +
prettyPrintExpr(arg1) + " " + prettyPrintExpr(arg2) + ")"
      case Hd(darg: Expression) => "(hd " + prettyPrintExpr(darg) + ")"
      case Tl(arg: Expression) => "(tl " + prettyPrintExpr(arg) + ")"
      case Eq(arg1: Expression, arg2: Expression) =>
prettyPrintExpr(arg1) + " =? " +  prettyPrintExpr(arg2)

    }
  }
  */

  /**
   *  FONCTIONS AUXILIAIRES DE TRAITEMENT DE CHAINES POUR L'INDENTATION
DES COMMANDES
   *  OU LA PRESENTATION DU PROGRAMME
   */

  /**
   * recherche d'une valeur d'indentation dans une liste de
spécifications d'indentation
   *
   * @param context une chaîne de caractères décrivant un contexte
d'indentation
   * @param is une liste de spécifications d'indentation, chaque
spécification étant un couple (un contexte,une indentation)
   * les contextes possibles seront, en majuscules, "WHILE", "FOR",
"IF", ou "PROGR".
   * @return l'indentation correspondant à context
   */


  def indentSearch(context: String, is: IndentSpec): Int = {
    is match  {
      case Nil => indentDefault
      case hd :: tl => if(hd._1.equals(context)){ hd._2}
```

```
                    else {indentSearch(context ,tl)}
    }
  }

  /**
   * création d'une indentation
   *
   * @param n un nombre d'espaces
   * @return une chaîne de n espaces
   */


  def makeIndent(n: Int): String = {
    n match{
      case 0 => ""
      case _ => " " + makeIndent(n-1)
    }
  }
  /*autre maniere a faire  avec for
   * def makeIndent(n: Int): String = {
    var chaineEspace = ""
    for(i <- 1 to n) {
      chaineEspace = chaineEspace + " "
    }
    chaineEspace
  }
  */

  /**
   * ajout d'une chaîne devant chaque élément d'une liste non vide de
chaînes
   *
   * @param pref une chaîne
   * @param strings une liste non vide de chaînes
   * @return une liste de chaînes obtenue par la concaténation de pref
devant chaque élément de strings
   */

//Ajouter une situation pour éviter l'apparition de situation imprévu-
liste vide
//autant la résolution de cette fonction n'est pas exclusive

 def appendStringBeforeAll(pref: String, strings: List[String]):
List[String] = {
   strings match {
     //case Nil => Nil
```

```
      case hd :: Nil => pref + hd :: Nil
      case hd :: tl => pref + hd :: appendStringBeforeAll(pref ,tl)
    }
}



  /**
   * ajout d'une chaîne après chaque élément d'une liste non vide de
chaînes
   *
   * @param suff une chaîne
   * @param strings une liste non vide de chaînes
   * @return une liste de chaînes obtenue par la concaténation de suff
après chaque élément de strings
   */


  def appendStringAfterAll(suff: String, strings: List[String]):
List[String] = {
    strings match {
      //case Nil => Nil
      case hd :: Nil =>  hd + suff :: Nil
      case hd :: tl =>  hd + suff :: appendStringAfterAll(suff ,tl)
    }
  }

  /**
   * ajout d'une chaîne après le dernier élément d'une liste non vide de
chaînes
   *
   * @param suff une chaîne
   * @param strings une liste non vide de chaînes
   * @return une liste de chaînes obtenue par la concaténation de suff
après le dernier élément de strings
   */


  def appendStringAfterLast(suff: String, strings: List[String]):
List[String] = {
    strings match {
      //case Nil => Nil
      case hd :: Nil => hd + suff::Nil
      case hd :: tl => hd :: appendStringAfterLast(suff , tl)
    }
  }
```

```
  /**
   * ajout d'une chaîne après chaque élément d'une liste non vide de
chaînes sauf le dernier
   *
   * @param suff une chaîne
   * @param strings une liste non vide de chaînes
   * @return une liste de chaînes obtenue par la concaténation de suff
après chaque élément de strings sauf le dernier
   */

  //TODO TP2
  def appendStringAfterAllButLast(suff: String, strings: List[String]):
List[String] = {
    strings match {
      //case Nil => Nil
      case hd :: Nil => hd :: Nil
      case hd :: tl => hd + suff :: appendStringAfterAllButLast(suff
,tl)
    }
  }

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

  /**
   * @param command : un AST décrivant une commande du langage WHILE
   * @param is : une liste de spécifications d'indentation
   * @return une liste de chaînes représentant la syntaxe concrète de la
commande
   */
  // Il faut utiliser fonction prettyPrintCommands pour decompose
List(command) à command pour Self-Reccu
  def prettyPrintCommand(command: Command, is: IndentSpec): List[String]
= {
    command match {
      case Nop => "nop" :: Nil
      case Set(Var(name), expression) => name + " := " +
prettyPrintExpr(expression) ::Nil

      //case While(condition, body)                    => List("while"+
prettyPrintExpr(condition) + "do")
     // ::: appendStringBeforeAll( makeIndent(indentSearch("WHILE" ,
is)),prettyPrintCommands( body , is))::: List("od")
```

```
      case While(condition, body) => "while " +
prettyPrintExpr(condition) + " do" ::
      appendStringBeforeAll(makeIndent(indentSearch("WHILE", is)),
prettyPrintCommands(body, is)) :::
      "od" :: Nil

      case For(count, body)                          => "for "
+prettyPrintExpr(count)+" do"::
      appendStringBeforeAll(makeIndent(indentSearch("FOR", is)),
prettyPrintCommands(body, is)) :::
      "od" :: Nil

      case If(condition, then_command, else_command) =>"if "
+prettyPrintExpr(condition)+" then"::
      appendStringBeforeAll(makeIndent(indentSearch("IF", is)),
prettyPrintCommands(then_command, is)) ::: "else"::Nil :::
      appendStringBeforeAll(makeIndent(indentSearch("IF", is)),
prettyPrintCommands(else_command, is)) ::: "fi"::Nil

    }
  }

  /*
  def prettyPrintCommand(command: Command, is: IndentSpec): List[String]
= {
    command match {
      case Nop                                       => ???
      case Set(Var(name), expression)                => ???
      case While(condition, body)                    => ???
      case For(count, body)                          => ???
      case If(condition, then_command, else_command) => ???
    }
  }
  */

  /**
   * @param commands : une liste non vide d'AST décrivant une liste non
vide de commandes du langage WHILE
   * @param is : une liste de spécifications d'indentation
   * @return une liste de chaînes représentant la syntaxe concrète de la
liste de commandes
   */
  def prettyPrintCommands(commands: List[Command], is: IndentSpec):
List[String] ={
    commands match {
```

```
      case c::Nil => prettyPrintCommand(c,is)
      case c::reste => appendStringAfterLast("
;",prettyPrintCommand(c,is))++prettyPrintCommands(reste,is)
    }
  }

  /*
  def prettyPrintCommands(commands: List[Command], is: IndentSpec):
List[String] = {
    commands match {
      case command :: Nil      => ???
      case command :: commands => ???
      case Nil                 => throw ExceptionListeVide
    }
  }
      */

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

  /**
   * @param vars : une liste non vide décrivant les paramètres d'entrée
d'un programme du langage WHILE
   * @return une chaîne représentant la syntaxe concrète des paramètres
d'entrée du programme
   */
  def prettyPrintIn(vars: List[Variable]): String = {
    vars match {
      //case Nil => ""
      case Var(varName)::Nil => varName
      case Var(varName)::reste => varName + ", "+ prettyPrintIn(reste)
    }
  }

  /**
   * @param vars : une liste non vide décrivant les paramètres de sortie
d'un programme du langage WHILE
   * @return une chaîne représentant la syntaxe concrète des paramètres
de sortie du programme
   */
  def prettyPrintOut(vars: List[Variable]): String ={
    vars match {
      //case Nil => ""
      case Var(varName)::Nil => varName
```

```
      case Var(varName)::reste => varName + ", "+ prettyPrintOut(reste)
    }
  }

  /**
   * @param program : un AST décrivant un programme du langage WHILE
   * @param is : une liste de spécifications d'indentation
   * @return une liste de chaînes représentant la syntaxe concrète du
programme
   */
  def prettyPrintProgram(program: Program, is: IndentSpec): List[String]
= {
    program match{
      case Progr(in,body,out) =>
        List("read " +prettyPrintIn(in),"%")++

appendStringBeforeAll(makeIndent(indentSearch("PROGR",is)),prettyPrintCo
mmands(body,is))++
      List("%","write "+prettyPrintOut(out))
    }
  }

  /**
   * @param program : un AST décrivant un programme du langage WHILE
   * @param is : une liste de spécifications d'indentation
   * @return une chaîne représentant la syntaxe concrète du programme
   */
  def prettyPrint(program: Program, is: IndentSpec): String ={
     prettyPrintProgram(program,is).mkString("\n")
     }

  val program: Program =
    Progr(
      List(Var("X")),
      List(
        Set(Var("Y"), Nl),
        While(
          VarExp("X"),
          List(
            Set(Var("Y"), Cons(Hd(VarExp("X")), VarExp("Y"))),
            Set(Var("X"), Tl(VarExp("X")))))),
      List(Var("Y")));
  val is: IndentSpec = List(("PROGR", 2), ("WHILE", 5));

  def main(args: Array[String]): Unit = {
    println(prettyPrint(program, is));
```

```
  }

  /**
   * 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 caractère 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 caractère 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 caractère 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)

}
```