```scala
package fr.istic.cal.while1cons

import scala.util.Try

/**
 * 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 While1cons {

  /**
   * UN ELIMINATEUR D'EXPRESSIONS COMPLEXES POUR LE LANGAGE WHILE
   *
   */

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

  /**
   * @param expression : un AST décrivant une expression du langage
WHILE
   * @return une paire constituée d'une liste d'affectations ayant le
même effet
   * que l'expression et de la variable qui contient le résultat
   */
  def while1ConsExprV(expression: Expression): (List[Command], Variable)
= {
    expression match {
      case Nl =>
        //La nouvelle variable créée par chaque expression,on l'appele
v.
        val v = NewVar.make()
        (List(Set(v, Nl)), v)

      case Cst(name) =>
        val v = NewVar.make()
        (List(Set(v, Cst(name))), v)

      case VarExp(name: String) => (Nil, Var(name))

      case Cons(arg1: Expression, arg2: Expression) =>
        //Stocker les resuletat des deux expressions
        val (listCmdArg1, Var(newArg1)) = while1ConsExprV(arg1)
        val (listCmdArg2, Var(newArg2)) = while1ConsExprV(arg2)
```

```
        val v = NewVar.make()
        (listCmdArg1 ++ listCmdArg2 ++ List(Set(v, Cons(VarExp(newArg1),
VarExp(newArg2)))), v)
      case Hd(arg: Expression) =>
        val (listCmdArg, Var(newArg)) = while1ConsExprV(arg)
        val v = NewVar.make()
        (listCmdArg ++ List(Set(v, Hd(VarExp(newArg)))), v)
      case Tl(arg: Expression) =>
        //Resembler case Hd
        val (listCmdArg, Var(newArg)) = while1ConsExprV(arg)
        val v = NewVar.make()
        (listCmdArg ++ List(Set(v, Tl(VarExp(newArg)))), v)
      case Eq(arg1: Expression, arg2: Expression) =>
        //Resembler case Cons
        val (listCmdArg1, Var(newArg1)) = while1ConsExprV(arg1)
        val (listCmdArg2, Var(newArg2)) = while1ConsExprV(arg2)
        val v = NewVar.make()
        (listCmdArg1 ++ listCmdArg2 ++ List(Set(v, Eq(VarExp(newArg1),
VarExp(newArg2)))), v)
    }
  }

  /**
   * @param expression : un AST décrivant une expression du langage
WHILE
   * @return une paire constituée d'une liste d'affectations et une
expression simple
   * qui, combinées, ont le même effet que l'expression initiale
   */
  def while1ConsExprSE(expression: Expression): (List[Command],
Expression) = {
    expression match {
      case Nl                   => (List(), Nl)
      case Cst(name)            => (List(), Cst(name))
      case VarExp(name: String) => (List(), VarExp(name))
      case Cons(arg1: Expression, arg2: Expression) =>
        val (listCmdArg1, Var(newArg1)) = while1ConsExprV(arg1)
        val (listCmdArg2, Var(newArg2)) = while1ConsExprV(arg2)
        val consExpr = Cons(VarExp(newArg1), VarExp(newArg2))
        (listCmdArg1 ++ listCmdArg2, consExpr)
      case Hd(arg: Expression) =>
        val (listCmdArg, Var(newArg)) = while1ConsExprV(arg)
        val hdExpr = Hd(VarExp(newArg))
        (listCmdArg, hdExpr)
      case Tl(arg: Expression) =>
        val (listCmdArg, Var(newArg)) = while1ConsExprV(arg)
        val tlExpr = Tl(VarExp(newArg))
        (listCmdArg, tlExpr)
      case Eq(arg1: Expression, arg2: Expression) =>
        val (listCmdArg1, Var(newArg1)) = while1ConsExprV(arg1)
        val (listCmdArg2, Var(newArg2)) = while1ConsExprV(arg2)
        val eqExpr = Eq(VarExp(newArg1), VarExp(newArg2))
        (listCmdArg1 ++ listCmdArg2, eqExpr)
```

```
    }
  }

  /**
   *
   *  TRAITEMENT DES COMMANDES DU LANGAGE WHILE
   */
  /**
   * @param command : un AST décrivant une commande du langage WHILE
   * @return une liste de commandes ayant un seul constructeur par
expression
   * et ayant le même effet que la commande initiale
   */
  // TODO TP4
  def while1ConsCommand(command: Command): List[Command] = {
    def convertVariableToExpression(v: Variable): VarExp = {
      v match {
        case Var(name) => VarExp(name)
      }
    }
    command match {
      case Nop => List(Nop)
      case Set(v, e) => {
        val (listCmdArg, newExpr) = while1ConsExprSE(e)
        listCmdArg ++ List(Set(v, newExpr))
      }
      case While(cond, body) => {
        val (listCmdArg, Var(newArg)) = while1ConsExprV(cond)
        listCmdArg ++ List(
          While(
            VarExp(newArg),
            while1ConsCommands(body) ++ listCmdArg))
      }
      case For(count, body) => {
        val (listCmdArg, Var(newArg)) = while1ConsExprV(count)
        listCmdArg ++ List(
          For(
            VarExp(newArg),
            while1ConsCommands(body)))
      }
      case If(cond, thenCmds, elseCmds) => {
        val (listCmdArg, Var(newArg)) = while1ConsExprV(cond)
        listCmdArg ++ List(
          If(
            VarExp(newArg),
            while1ConsCommands(thenCmds),
            while1ConsCommands(elseCmds)))
      }
    }
  }

  /**
   * @param commands : une liste non vide d'AST décrivant une liste non
```

```
vide de commandes du langage WHILE
   * @return une liste de commandes ayant un seul constructeur par
expression
   *         et ayant le même effet que les commandes initiales
   */
  def while1ConsCommands(commands: List[Command]): List[Command] = {
    commands match {
      case Nil          => throw ExceptionListeVide
      case cmd :: Nil   => while1ConsCommand(cmd) //éviter de passer à
Exception s'il y des cmd
      case cmd :: reste => while1ConsCommand(cmd) ++
while1ConsCommands(reste)
    }
  }

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

  /**
   * @param program : un AST décrivant un programme du langage WHILE
   * @return un AST décrivant un programme du langage WHILE
   *         de même sémantique que le programme initial mais ne
contenant que des expressions simples
   */
  def while1ConsProgr(program: Program): Program = {
    program match {
      case Progr(Nil, _, _)     => throw ExceptionListeVide
      case Progr(in, body, out) => Progr(in, while1ConsCommands(body),
out)
    }
  }

  def main(args: Array[String]): Unit = {

    // vous pouvez ici tester manuellement vos fonctions par des print

  }

  /**
   * 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)
  }
}
```