```
package patmat

import common._

/**
 * Assignment 4: Huffman coding
 *
 * !!!!  DISCLAIMER !!!!
 *       The skeleton code for this was provided as an assignment for the
 *       coursera course - Functional Programming in Scala by Martin Odersky
 *       I am providing you with this code snippet to show you my understanding
 *       of functional programming principles
 *
 * !!!!   DISCLAIMER !!!!
 */
object Huffman {

  /**
   * A huffman code is represented by a binary tree.
   *
   * Every `Leaf` node of the tree represents one character of the alphabet that
the tree can encode.
   * The weight of a `Leaf` is the frequency of appearance of the character.
   *
   * The branches of the huffman tree, the `Fork` nodes, represent a set
containing all the characters
   * present in the leaves below it. The weight of a `Fork` node is the sum of
the weights of these
   * leaves.
   */
  abstract class CodeTree
  case class Fork(left: CodeTree, right: CodeTree, chars: List[Char], weight:
Int) extends CodeTree
  case class Leaf(char: Char, weight: Int) extends CodeTree

  // Part 1: Basics

  def weight(tree: CodeTree): Int = {
    tree match {
      case tree : Leaf => tree.weight
      case tree : Fork => this.weight(tree.left) + this.weight(tree.right)
    }

  } // tree match ...

  def chars(tree: CodeTree): List[Char] = {
       tree match {
      case tree : Leaf => List(tree.char)
      case tree : Fork => this.chars(tree.left) ::: this.chars(tree.right)
        }
  }

  def makeCodeTree(left: CodeTree, right: CodeTree) =
    Fork(left, right, chars(left) ::: chars(right), weight(left) + weight(right))



  // Part 2: Generating Huffman trees

  /**
   * In this assignment, we are working with lists of characters. This function
```

```
allows
   * you to easily create a character list from a given string.
   */
  def string2Chars(str: String): List[Char] = str.toList

  /**
   * This function computes for each unique character in the list `chars` the
number of
   * times it occurs. For example, the invocation
   *
   *   times(List('a', 'b', 'a'))
   *
   * should return the following (the order of the resulting list is not
important):
   *
   *   List(('a', 2), ('b', 1))
   *
   * The type `List[(Char, Int)]` denotes a list of pairs, where each pair
consists of a
   * character and an integer. Pairs can be constructed easily using parentheses:
   *
   *   val pair: (Char, Int) = ('c', 1)
   *
   * In order to access the two elements of a pair, you can use the accessors
`_1` and `_2`:
   *
   *   val theChar = pair._1
   *   val theInt  = pair._2
   *
   * Another way to deconstruct a pair is using pattern matching:
   *
   *   pair match {
   *     case (theChar, theInt) =>
   *       println("character is: "+ theChar)
   *       println("integer is  : "+ theInt)
   *   }
   */
  def times(chars: List[Char]): List[(Char, Int)] = {
   chars.groupBy((x) => x).map(x => (x._1, x._2.length)).toList.sortWith((x, y)
=> x._1 < y._1)
  }

  /**
   * Returns a list of `Leaf` nodes for a given frequency table `freqs`.
   *
   * The returned list should be ordered by ascending weights (i.e. the
   * head of the list should have the smallest weight), where the weight
   * of a leaf is the frequency of the character.
   */
  def makeOrderedLeafList(freqs: List[(Char, Int)]): List[Leaf] = {
    freqs.sortWith((x,y) => x._2 < y._2).map(x => Leaf(x._1, x._2))
  }

  /**
   * Checks whether the list `trees` contains only one single code tree.
   */
  def singleton(trees: List[CodeTree]): Boolean = trees.length == 1

  /**
   * The parameter `trees` of this function is a list of code trees ordered
   * by ascending weights.
```

```
    *
    * This function takes the first two elements of the list `trees` and combines
    * them into a single `Fork` node. This node is then added back into the
    * remaining elements of `trees` at a position such that the ordering by
weights
    * is preserved.
    *
    * If `trees` is a list of less than two elements, that list should be returned
    * unchanged.
    */
  def combine(trees: List[CodeTree]): List[CodeTree] = {
    if (trees.length < 2) trees
    else
    {
       val f = Fork(trees(0), trees(1), chars(trees(0)) ::: chars(trees(1)),
weight(trees(0)) + weight(trees(1)));
       val n = f :: trees.drop(2)
       //combine(n.sortWith((x, y) => weight(x) < weight(y)))
       //n.sortWith((x, y) => weight(x) < weight(y))
       n.sortBy(weight)
    }
  }

  /**
    * This function will be called in the following way:
    *
    *   until(singleton, combine)(trees)
    *
    * where `trees` is of type `List[CodeTree]`, `singleton` and `combine` refer
to
    * the two functions defined above.
    *
    * In such an invocation, `until` should call the two functions until the list
of
    * code trees contains only one single tree, and then return that singleton
list.
    *
    * Hint: before writing the implementation,
    *  - start by defining the parameter types such that the above example
invocation
    *    is valid. The parameter types of `until` should match the argument types
of
    *    the example invocation. Also define the return type of the `until`
function.
    *  - try to find sensible parameter names for `xxx`, `yyy` and `zzz`.
    */
  def until(xxx: List[CodeTree] => Boolean ,
            yyy: List[CodeTree] => List[CodeTree])
                (zzz: List[CodeTree]): CodeTree = {
 if (xxx(zzz)) zzz(0)
 else
   until(xxx,yyy) (yyy(zzz))
  }

  /**
    * This function creates a code tree which is optimal to encode the text
`chars`.
    *
    * The parameter `chars` is an arbitrary text. This function extracts the
character
    * frequencies from that text and creates a code tree based on them.
```

```
   */
  def createCodeTree(chars: List[Char]): CodeTree = {
    until(singleton, combine) (makeOrderedLeafList(times(chars)))
  }



  // Part 3: Decoding

  type Bit = Int

  /**
   * This function decodes the bit sequence `bits` using the code tree `tree` and
returns
   * the resulting list of characters.
   */
  def decode(tree: CodeTree, bits: List[Bit]): List[Char] = {
    val top = tree
    def _decode(tree: CodeTree, bits: List[Bit], acc: List[Char]) : List[Char] =
{

      tree match {
      case tree: Leaf => { if (bits.isEmpty) acc ++ List(tree.char)
                           else
                            _decode(top, bits, acc ++ List(tree.char))
                         }
      case tree: Fork => {
                          if (bits.isEmpty) acc
                          else
                          {
                             if (bits.head == 0) _decode(tree.left, bits.tail,
acc)
                             else
                             _decode(tree.right, bits.tail, acc)
                          }
                          }

                 }
    }
    val acc : List[Char] = List()
    _decode(top, bits, acc)
  }

  /**
   * A Huffman coding tree for the French language.
   * Generated from the data given at
   *
http://fr.wikipedia.org/wiki/Fr%C3%A9quence_d%27apparition_des_lettres_en_fran%C3%
A7ais
   */
  val frenchCode: CodeTree =
Fork(Fork(Fork(Leaf('s',121895),Fork(Leaf('d',56269),Fork(Fork(Fork(Leaf('x',5928)
,Leaf('j',8351),List('x','j'),14279),Leaf('f',16351),List('x','j','f'),30630),For
k(Fork(Fork(Fork(Leaf('z',2093),Fork(Leaf('k',745),Leaf('w',1747),List('k','w'),2
492),List('z','k','w'),4585),Leaf('y',4725),List('z','k','w','y'),9310),Leaf('h',1
1298),List('z','k','w','y','h'),20608),Leaf('q',20889),List('z','k','w','y','h','
'),41497),List('x','j','f','z','k','w','y','h','q'),72127),List('d','x','j','f','
','k','w','y','h','q'),128396),List('s','d','x','j','f','z','k','w','y','h','q'),
50291),Fork(Fork(Leaf('o',82762),Leaf('l',83668),List('o','l'),166430),Fork(Fork(I
eaf('m',45521),Leaf('p',46335),List('m','p'),91856),Leaf('u',96785),List('m','p','
u'),188641),List('o','l','m','p','u'),355071),List('s','d','x','j','f','z','k','w'
```

```
,'y','h','q','o','l','m','p','u'),605362),Fork(Fork(Fork(Leaf('r',100500),Fork(Le
af('c',50003),Fork(Leaf('v',24975),Fork(Leaf('g',13288),Leaf('b',13822),List('g','
b'),27110),List('v','g','b'),52085),List('c','v','g','b'),102088),List('r','c','v'
,'g','b'),202588),Fork(Leaf('n',108812),Leaf('t',111103),List('n','t'),219915),Li
st('r','c','v','g','b','n','t'),422503),Fork(Leaf('e',225947),Fork(Leaf('i',11546
5),Leaf('a',117110),List('i','a'),232575),List('e','i','a'),458522),List('r','c','
v','g','b','n','t','e','i','a'),881025),List('s','d','x','j','f','z','k','w','y','
h','q','o','l','m','p','u','r','c','v','g','b','n','t','e','i','a'),1486387)

  /**
   * What does the secret message say? Can you decode it?
   * For the decoding use the `frenchCode' Huffman tree defined above.
   */
  val secret: List[Bit] =
List(0,0,1,1,1,0,1,0,1,1,1,0,0,1,1,0,1,0,0,1,1,0,1,0,1,1,0,0,1,1,1,1,1,1,0,1,0,1,1,0
,0,0,0,1,0,1,1,1,0,0,1,0,0,1,0,0,0,1,0,0,0,1,0,1)

  /**
   * Write a function that returns the decoded secret
   */
  def decodedSecret: List[Char] = decode(frenchCode, secret)



  // Part 4a: Encoding using Huffman tree

  /**
   * This function encodes `text` using the code tree `tree`
   * into a sequence of bits.
   */
  def encode(tree: CodeTree)(text: List[Char]): List[Bit] = {
    val top = tree
    def _encode(tree: CodeTree, text: List[Char], acc: List[Bit]) : List[Bit] = {

      tree match {
      case tree: Leaf => { if (text.isEmpty) acc
                           else
                            _encode(top, text.tail, acc)
                         }
      case tree: Fork => {
                          if (text.isEmpty) acc
                          else
                          {
                            if (this.chars(tree.left).contains(text.head))
                            {
                              _encode(tree.left, text, acc ++ List(0))
                            }
                            else
                            {
                              _encode(tree.right, text, acc ++ List(1))
                            }
                          }
                          }

                }
    }
    val acc : List[Bit] = List()
    _encode(top, text, acc)

  }
```

```
  // Part 4b: Encoding using code table

  type CodeTable = List[(Char, List[Bit])]

  /**
   * This function returns the bit sequence that represents the character `char` in
   * the code table `table`.
   */
  def codeBits(table: CodeTable)(char: Char): List[Bit] = {
    if (table.head._1 == char) table.head._2
    else
      codeBits(table.tail)(char)
  }

  /**
   * Given a code tree, create a code table which contains, for every character in the
   * code tree, the sequence of bits representing that character.
   *
   * Hint: think of a recursive solution: every sub-tree of the code tree `tree` is itself
   * a valid code tree that can be represented as a code table. Using the code tables of the
   * sub-trees, think of how to build the code table for the entire tree.
   */
  def convert(tree: CodeTree): CodeTable = {
    val top = tree
    def _convert(tree: CodeTree) : CodeTable = {

      tree match {
      case tree: Leaf => {
                                List((tree.char, List()))
                            }
      case tree: Fork => {
                                mergeCodeTables(
                                 _convert(tree.left),
                                 _convert(tree.right)
                                 )
                             }
                  }
    }
    _convert(top)
  }

  /**
   * This function takes two code tables and merges them into one. Depending on how you
   * use it in the `convert` method above, this merge method might also do some transformations
   * on the two parameter code tables.
   */
  def mergeCodeTables(a: CodeTable, b: CodeTable): CodeTable = {
    a.map(x => (x._1, 0 :: x._2)) ::: b.map(x => (x._1, 1 :: x._2))
  }

  /**
   * This function encodes `text` according to the code tree `tree`.
   *
   * To speed up the encoding process, it first converts the code tree to a code table
```

```
   * and then uses it to perform the actual encoding.
   */
  def quickEncode(tree: CodeTree)(text: List[Char]): List[Bit] = {
  val map = convert(tree).toMap
  def _quickEncode(text: List[Char], acc: List[Bit]): List[Bit] = {
     if (text.isEmpty) acc
     else
       _quickEncode(text.tail, acc ::: map(text.head))
  }
  _quickEncode(text, List())
  }
}
```