Putting the Pieces Together

Principles of Functional Programming

Task

Phone keys have mnemonics assigned to them.

```
val mnemonics = Map(
      '2' -> "ABC", '3' -> "DEF", '4' -> "GHI", '5' -> "JKL",
      '6' -> "MNO", '7' -> "PQRS", '8' -> "TUV", '9' -> "WXYZ")
```

Assume you are given a dictionary `words` as a list of words.

Design a method `encode` such that

```
encode(phoneNumber)
```

produces all phrases of words that can serve as mnemonics for the phone number.

Example: The phone number "7225247386" should have the mnemonic `Scala is fun` as one element of the set of solution phrases.

Outline

```
class Coder(words: List[String]) {
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  val charCode: Map[Char, Char] = ???

  /** Maps a word to the digit string it can represent */
  private def wordCode(word: String): String = ???

  /** Maps a digit string to all words in the dictionary that represent it */
  private val wordsForNum: Map[String, List[String]] = ???

  /** All ways to encode a number as a list of words */
  def encode(number: String): Set[List[String]] = ???
}
```

Implementation (1)

```
class Coder(words: List[String]) {
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  val charCode: Map[Char, Char] =
```

Implementation (1)

```
class Coder(words: List[String]) {
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  val charCode: Map[Char, Char] =
    for (digit, str) <- mnemonics; ltr <- str yield ltr -> digit
```

Implementation (1)

```
class Coder(words: List[String]) {
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  val charCode: Map[Char, Char] =
    for (digit, str) <- mnemonics; ltr <- str yield ltr -> digit

  /** Maps a word to the digit string it can represent */
  private def wordCode(word: String): String =
```

Implementation (1)

```
class Coder(words: List[String]) {
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  val charCode: Map[Char, Char] =
    for (digit, str) <- mnemonics; ltr <- str yield ltr -> digit

  /** Maps a word to the digit string it can represent */
  private def wordCode(word: String): String = word.toUpperCase.map(charCode)
```

Implementation (1)

```
class Coder(words: List[String]) {
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  val charCode: Map[Char, Char] =
    for (digit, str) <- mnemonics; ltr <- str yield ltr -> digit

  /** Maps a word to the digit string it can represent */
  private def wordCode(word: String): String = word.toUpperCase.map(charCode)

  /** Maps a digit string to all words in the dictionary that represent it */
  private val wordsForNum: Map[String, List[String]] =
```

Implementation (1)

```
class Coder(words: List[String]) {
  val mnemonics = Map(...)

  /** Maps a letter to the digit it represents */
  val charCode: Map[Char, Char] =
    for (digit, str) <- mnemonics; ltr <- str yield ltr -> digit

  /** Maps a word to the digit string it can represent */
  private def wordCode(word: String): String = word.toUpperCase.map(charCode)

  /** Maps a digit string to all words in the dictionary that represent it */
  private val wordsForNum: Map[String, List[String]] =
    words.groupBy(wordCode).withDefaultValue(Nil)
```

Implementation (2)

```
/** All ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] =
```

Idea: use divide and conquer

Implementation (2)

```
/** All ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] =
  if number.isEmpty then ???
  else ???
```

Implementation (2)

```
/** All ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] =
  if number.isEmpty then Set(Nil)
  else ???
```

Implementation (2)

```
/** All ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] =
  if number.isEmpty then Set(Nil)
  else
    for
      splitPoint <- (1 to number.length).toSet
      word <- ???
      rest <- ???
    yield word :: rest
```

Implementation (2)

```
/** All ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] =
  if number.isEmpty then Set(Nil)
  else
    for
      splitPoint <- (1 to number.length).toSet
      word <- wordsForNum(number.take(splitPoint))
      rest <- ???
    yield word :: rest
```

Implementation (2)

```
/** All ways to encode a number as a list of words */
def encode(number: String): Set[List[String]] =
  if number.isEmpty then Set(Nil)
  else
    for
      splitPoint <- (1 to number.length).toSet
      word <- wordsForNum(number.take(splitPoint))
      rest <- encode(number.drop(splitPoint))
    yield word :: rest
```

Testing It

A test program:

```
@main def code(number: String) =
  val coder = Coder(List(
    "Scala", "Python", "Ruby", "C",
    "rocks", "socks", "sucks", "works", "pack"))
  coder.encode(number).map(_.mkString(" "))
```

A sample run:

```
> scala code "7225276257"
HashSet(Scala rocks, pack C rocks, pack C socks, Scala socks)
```

Background

This example was taken from:

> *Lutz Prechelt: An Empirical Comparison of Seven Programming Languages. IEEE Computer 33(10): 23-29 (2000)*

Tested with Tcl, Python, Perl, Rexx, Java, C++, C.

Code size medians:

- 100 loc for scripting languages
- 200-300 loc for the others

Benefits

Scala's immutable collections are:

- *easy to use*: few steps to do the job.
- *concise*: one word replaces a whole loop.
- *safe*: type checker is really good at catching errors.
- *fast*: collection ops are tuned, can be parallelized.
- *universal*: one vocabulary to work on all kinds of collections.

This makes them an attractive tool for software development