Pairs and Tuples

Sorting Lists Faster

As a non-trivial example, let's design a function to sort lists that is more efficient than insertion sort.

A good algorithm for this is *merge sort*. The idea is as follows:

If the list consists of zero or one elements, it is already sorted.

Otherwise,

- Separate the list into two sub-lists, each containing around half of the elements of the original list.
- Sort the two sub-lists.
- Merge the two sorted sub-lists into a single sorted list.

First MergeSort Implementation

Here is the implementation of that algorithm in Scala:

```
def msort(xs: List[Int]): List[Int] = {
  val n = xs.length/2
  if (n == 0) xs
  else {
    def merge(xs: List[Int], ys: List[Int]) = ???
    val (fst, snd) = xs splitAt n
    merge(msort(fst), msort(snd))
  }
}
```

Definition of Merge

Here is a definition of the `merge` function:

```
def merge(xs: List[Int], ys: List[Int]) =
  xs match {
    case Nil =>
      ys
    case x :: xs1 =>
      ys match {
        case Nil =>
          xs
        case y :: ys1 =>
          if (x < y) x :: merge(xs1, ys)
          else y :: merge(xs, ys1)
      }
    }
```

The SplitAt Function

The `splitAt` function on lists returns two sublists

- the elements up the the given index
- the elements from that index

The lists are returned in a *pair*.

Detour: Pair and Tuples

The pair consisting of `x` and `y` is written `(x, y)` in Scala.

Example

```
val pair = ("answer", 42)   > pair : (String, Int) = (answer,42)
```

The type of `pair` above is `(String, Int)`.

Pairs can also be used as patterns:

```
val (label, value) = pair   > label : String = answer
                            | value : Int = 42
```

This works analogously for tuples with more than two elements.

Translation of Tuples

A tuple type $(T_1, ..., T_n)$ is an abbreviation of the parameterized type

$$\texttt{scala.Tuple}n[T_1, ..., T_n]$$

A tuple expression $(e_1, ..., e_n)$ is equivalent to the function application

$$\texttt{scala.Tuple}n(e_1, ..., e_n)$$

A tuple pattern $(p_1, ..., p_n)$ is equivalent to the constructor pattern

$$\texttt{scala.Tuple}n(p_1, ..., p_n)$$

The Tuple class

Here, all `Tuple`*n* classes are modeled after the following pattern:

```
case class Tuple2[T1, T2](_1: +T1, _2: +T2) {
  override def toString = "(" + _1 + "," + _2 +")"
}
```

The fields of a tuple can be accessed with names `_1`, `_2`, ...

So instead of the pattern binding

```
val (label, value) = pair
```

one could also have written:

```
val label = pair._1
val value = pair._2
```

But the pattern matching form is generally preferred.

Exercise

The `merge` function as given uses a nested pattern match.

This does not reflect the inherent symmetry of the merge algorithm.

Rewrite `merge` using a pattern matching over pairs.

```
def merge(xs: List[Int], ys: List[Int]): List[Int] =
  (xs, ys) match {
    ???
  }
```