Implicit Parameters

Making Sort more General

Problem: How to parameterize `msort` so that it can also be used for lists with elements other than `Int`?

```
def msort[T](xs: List[T]): List[T] = ...
```

does not work, because the comparison < in `merge` is not defined for arbitrary types `T`.

Idea: Parameterize `merge` with the necessary comparison function.

Parameterization of Sort

The most flexible design is to make the function `sort` polymorphic and to pass the comparison operation as an additional parameter:

```
def msort[T](xs: List[T])(lt: (T, T) => Boolean) = {
  ...
    merge(msort(fst)(lt), msort(snd)(lt))
}
```

Merge then needs to be adapted as follows:

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

Calling Parameterized Sort

We can now call `msort` as follows:

```
val xs = List(-5, 6, 3, 2, 7)
val fruit = List("apple", "pear", "orange", "pineapple")

merge(xs)((x: Int, y: Int) => x < y)
merge(fruit)((x: String, y: String) => x.compareTo(y) < 0)
```

Or, since parameter types can be inferred from the call `merge(xs)`:

```
merge(xs)((x, y) => x < y)
```

Parametrization with Ordered

There is already a class in the standard library that represents orderings.

```
scala.math.Ordering[T]
```

provides ways to compare elements of type `T`. So instead of parameterizing with the `lt` operation directly, we could parameterize with `Ordering` instead:

```
def msort[T](xs: List[T])(ord: Ordering) =

  def merge(xs: List[T], ys: List[T]) =
    ... if (ord.lt(x, y)) ...

  ... merge(msort(fst)(ord), msort(snd)(ord)) ...
```

Ordered Instances:

Calling the new `msort` can be done like this:

```
import math.Ordering

msort(nums)(Ordering.Int)
msort(fruits)(Ordering.String)
```

This makes use of the values `Int` and `String` defined in the `scala.math.Ordering` object, which produce the right orderings on integers and strings.

Aside: Implicit Parameters

Problem: Passing around `lt` or `ord` values is cumbersome.

We can avoid this by making `ord` an implicit parameter.

```
def msort[T](xs: List[T])(implicit ord: Ordering) =

  def merge(xs: List[T], ys: List[T]) =
    ... if (ord.lt(x, y)) ...

  ... merge(msort(fst), msort(snd)) ...
```

Then calls to `msort` can avoid the ordering parameters:

```
msort(nums)
msort(fruits)
```

The compiler will figure out the right implicit to pass based on the demanded type.

Rules for Implicit Parameters

Say, a function takes an implicit parameter of type `T`.

The compiler will search an implicit definition that

- is marked `implicit`
- has a type compatible with `T`
- is visible at the point of the function call, or is defined in a companion object associated with `T`.

If there is a single (most specific) definition, it will be taken as actual argument for the implicit parameter.

Otherwise it's an error.

Exercise: Implicit Parameters

Consider the following line of the definition of `msort`:

```
... merge(msort(fst), msort(snd)) ...
```

Which implicit argument is inserted?

- `Ordering.Int`
- `Ordering.String`
- the "ord" parameter of "msort"