App/Opt

Parsing arguments for Scala command-line applications

An `App` is the core of a command-line application. Its `Options` must be declared as a list of the `Opt`s that it is prepared to accept on the commandline. Its `Command` is the name it is known by. Its `Main` is invoked when the command-line options have been parsed. Many of its methods can be overridden.

Every `Opt` is associated with a pattern and an arity (usually between 1 and 2) and a meaning function. The subclasses of `Opt` defined here provide a useful selection of meaning functions (and their associated arities).

When an `App` starts, the list of command line arguments

```
arg1 arg2 ...
```

is parsed by finding a declared option whose pattern matches `arg1`, then taking `arity` arguments (including `arg1`) and passing them to the meaning function. Then `arity` arguments are removed from the list, and the parsing continues. The parsing stops when there are no arguments left (or when a parsing error occurs).

If the pattern starts with a single quote `"'"` then it is interpreted (without the `"'"`) as a literal. If the pattern ends with `"="` then it is also interpreted as a literal, except that it matches a command line argument that it prefixes, and the remainder of that argument (after the "=") is treated as if it had been given as an additional argument. This is (mostly) indended for parsing options of the form `--opt=`*argument*

See below for a simple and effective way of using the `App/Opt` packages to accumulate a queue of jobs to be performed by the program (and the environments in which each job is to be performed). The Job queue is available when the options and paths have all been parsed – but not before. An error in parsing causes the program to exit *before any "semantic" processing has been done.*

```
import ox.app.OPT._
import scala.collection.mutable

object OptTest extends App {
  import collection.mutable.Queue
  case class Env (
   var f: String  = "Undefined",
   var g: Boolean = false,
   var h: Boolean = false,
   var i: Boolean = true,
   var k: Int     = 45,
   var r: Double  = 3.1415
  )
  { override def toString = s"f=$f, g=$g, h=$h, i=$i, k=$k, r=$r" }

  var env  = Env()
  var jobs = new mutable.Queue[(Env, String)]
  val Options = List (
     OPT("-help",  { Usage() },                   "prints usage text")          ,
     OPT("-d",     { Console.println(Env()) },   "prints initial options")      ,
     OPT("-f",     env.f ,      "<path> sets f to <path>")                      ,
     OPT("-g",     env.g ,      "inverts g")                                    ,
     OPT("-h",     env.h ,      "inverts h")                                    ,
     OPT("-i",     env.i ,      "inverts i")                                    ,
     OPT("-k",     env.k ,      "<int> sets k")                                 ,
     OPT("--k=",   env.k ,      "<int> sets k")                                 ,
     OPT("-r",     env.r,       "<real> sets r")                                ,
     ELSE("<path>",   { f => jobs.enqueue((env.copy(), f)) },
                        "adds a path to the list to be processed")              ,
     REST("--", (args => for (f <- args) jobs.enqueue((env, f))),
                "interprets all subsequent arguments as paths")
     )

   val Command = "OptTest"

   // Do the "semantic" processing
   def Main() : Unit =
       for ((env, path) <- jobs) Console.println(s"$path in $env")

}
```