```
#!/usr/bin/env ruby

# $Id: ls−lR.rb 24 2004−07−30 06:24:36Z shugo $
# <URL:http://svn.shugo.net/src/llw2004/trunk/ls−lR.rb>

require "shellwords"

module LsLR
  class Node
    attr_reader :line, :parent, :name, :size

    def initialize(line, parent, name, size)
      @line = line
      @parent = parent
      @name = name
      @size = size
    end

    def to_s
      return @line
    end

    def path
      if parent.nil?
        return "/"
      end
      return File.expand_path(name, parent.path)
    end

    def relative_path(base)
      if base == self
        return "."
      end
      return File.join(parent.relative_path(base), name)
    end

    def directory?
      return false
    end

    def accept(visitor)
      raise ScriptError, "subclass must override Node#accept"
    end
  end

  class FileNode < Node
    def type
      return "f"
    end

    def accept(visitor)
      visitor.visit_file(self)
    end
  end

  class DirectoryNode < Node
    attr_reader :children

    def initialize(line, parent, name, size)
      super
      @children = []
    end

    def type
      return "d"
    end

    def get_child(name)
      child = children.detect { |child| child.name == name }
```



```
      if child.nil?
        raise format("no such file or directory − %s", name)
      end
      return child
    end

    def get_descendant(path)
      first, rest = path.split("/", 2)
      child = get_child(first)
      if rest.nil?
        return child
      else
        return child.get_descendant(rest)
      end
    end

    def directory?
      return true
    end

    def accept(visitor)
      visitor.visit_directory(self)
    end
  end

  class Parser
    NODE_CLASSES = {
      "f" => FileNode,
      "d" => DirectoryNode
    }

    def initialize
      @input = nil
      @directories = nil
      @root_directory = nil
    end

    def parse(input)
      @input = input
      @directories = {}
      @root_directory = nil
      loop do
        begin
          parse_files
        rescue EOFError
          break
        end
      end
      return @root_directory
    end

    private

    def parse_files
      line = @input.readline
      dirname = line.slice(/(.*):/n, 1)
      if @directories.key?(dirname)
        dir = @directories[dirname]
      else
        dir = @root_directory =
          DirectoryNode.new("", nil, "", 0)
      end
      total = @input.readline
      @input.each_line do |line|
        line.chomp!
        break if line.empty?
        file = parse_file(line, dir)
        dir.children.push(file)
        if file.directory?
```



```
          @directories[File.join(dirname, file.name)] = file
        end
      end
      @directories.delete(dirname)
    end

    def parse_file(line, parent)
      m = /^(.)\S+\s+\d+\s+\w+\s+\w+\s+(\d+)\s+.{12}\s+(.*)/n.match(line)
      type = m[1].tr("−", "f")
      size = m[2].to_i
      name = m[3]
      file = NODE_CLASSES[type].new(line, parent, name, size)
    end
  end

  class Command
    attr_reader :shell

    def initialize(shell)
      @shell = shell
    end

    def exec(args)
      raise ScriptError, "subclass must override Command#exec"
    end
  end

  class NullCommand < Command
    def exec(args)
      raise "no such command"
    end
  end

  class QuitCommand < Command
    def exec(args)
      exit
    end
  end

  class PwdCommand < Command
    def exec(args)
      shell.print(shell.current_directory.path, "\n")
    end
  end

  class CdCommand < Command
    def exec(args)
      if args.length < 1
        return
      end
      path = args[0]
      dir = shell.get_file(path)
      unless dir.directory?
        raise format("not a directory − %s\n", path)
      end
      shell.current_directory = dir
    end
  end

  class LsCommand < Command
    def exec(args)
      shell.current_directory.children.each do |file|
        shell.print(file, "\n")
      end
    end
  end

  class Expression
    def evaluate(file)
```



```
      raise ScriptError, "subclass must override Expression#evaluate"
    end

    def null?
      return false
    end
  end

  class NullExpression
    def evaluate(file)
      return true
    end

    def null?
      return true
    end
  end

  class SimpleExpression < Expression
    def initialize(value)
      @value = value
    end
  end

  class NameExpression < SimpleExpression
    def evaluate(file)
      return File.fnmatch(@value, file.name)
    end
  end

  class TypeExpression < SimpleExpression
    def evaluate(file)
      return file.type == @value
    end
  end

  class SizeEqExpression < SimpleExpression
    def evaluate(file)
      return file.size == @value
    end
  end

  class SizeLtExpression < SimpleExpression
    def evaluate(file)
      return file.size < @value
    end
  end

  class SizeGtExpression < SimpleExpression
    def evaluate(file)
      return file.size > @value
    end
  end

  class NotExpression < Expression
    def initialize(expression)
      @expression = expression
    end

    def evaluate(file)
      return !@expression.evaluate(file)
    end
  end

  class BinaryExpression < Expression
    def initialize(left, right)
      @left = left
      @right = right
    end
```

```
  end

  class AndExpression < BinaryExpression
    def evaluate(file)
      return @left.evaluate(file) && @right.evaluate(file)
    end
  end

  class OrExpression < BinaryExpression
    def evaluate(file)
      return @left.evaluate(file) || @right.evaluate(file)
    end
  end

  class FindExpressionParser
    def initialize
      @tokens = nil
    end

    def parse(tokens)
      @tokens = tokens.dup
      return exprs
    end

    private

    def exprs
      result = NullExpression.new
      loop do
        token = lookahead
        if token.nil? || token == ")"
          break
        end
        e = or_expr
        if result.null?
          result = e
        else
          result = AndExpression.new(result, e)
        end
      end
      return result
    end

    def or_expr
      result = and_expr
      while lookahead == "−o"
        shift_token
        right = and_expr
        result = OrExpression.new(result, right)
      end
      return result
    end

    def and_expr
      result = not_expr
      while lookahead == "−a"
        shift_token
        right = not_expr
        result = AndExpression.new(result, right)
      end
      return result
    end

    def not_expr
      if @tokens.first == "!"
        shift_token
        return NotExpression.new(not_expr)
      else
        return primary_expr
```



```
      end
    end

    def primary_expr
      case lookahead
      when "−name"
        return name_expr
      when "−type"
        return type_expr
      when "−size"
        return size_expr
      when "("
        return paren_expr
      else
        raise format("unknown expression − %s", lookahead)
      end
    end

    def name_expr
      shift_token
      val = shift_token
      return NameExpression.new(val)
    end

    def type_expr
      shift_token
      val = shift_token
      if !/\A[fd]\z/n.match(val)
        raise format("invalid argument for −type − %s", val)
      end
      return TypeExpression.new(val)
    end

    SIZE_EXPRESSION_CLASSES = {
      nil => SizeEqExpression,
      "+" => SizeGtExpression,
      "−" => SizeLtExpression
    }

    def size_expr
      shift_token
      val = shift_token
      m = /\A(\+|−)?(\d+)\z/n.match(val)
      if m.nil?
        raise format("invalid argument for −size − %s", val)
      end
      return SIZE_EXPRESSION_CLASSES[m[1]].new(m[2].to_i)
    end

    def paren_expr
      shift_token
      result = exprs
      match(")")
      return result
    end

    def lookahead
      return @tokens.first
    end

    def shift_token
      return @tokens.shift
    end

    def match(pat)
      token = shift_token
      unless pat === token
        raise format("invalid token − %s (expected %s)", token, pat)
      end
```



```
      return token
    end
  end

  class DfsScheduler
    def initialize
      @stack = []
    end

    def next_node
      return @stack.pop
    end

    def schedule(nodes)
      @stack.concat(nodes.reverse)
    end
  end

  class BfsScheduler
    def initialize
      @queue = []
    end

    def next_node
      return @queue.shift
    end

    def schedule(nodes)
      @queue.concat(nodes)
    end
  end

  class SearchCommand < Command
    def initialize(shell, scheduler)
      super(shell)
      @expr_parser = FindExpressionParser.new
      @scheduler = scheduler
    end

    def exec(args)
      @expr = @expr_parser.parse(args)
      @scheduler.schedule([@shell.current_directory])
      while node = @scheduler.next_node
        node.accept(self)
      end
    end

    def visit_file(file)
      if @expr.evaluate(file)
        shell.print(file.relative_path(@shell.current_directory), "\n")
      end
    end

    def visit_directory(dir)
      visit_file(dir)
      @scheduler.schedule(dir.children)
    end
  end

  class Shell
    attr_reader :root_directory
    attr_accessor :current_directory

    def initialize
      @current_directory = nil
      null_cmd = NullCommand.new(self)
      @commands = Hash.new(null_cmd)
      @commands["quit"] = @commands["exit"] = QuitCommand.new(self)
      @commands["pwd"] = PwdCommand.new(self)
```



```
      @commands["cd"] = CdCommand.new(self)
      @commands["ls"] = LsCommand.new(self)
      dfs_scheduler = DfsScheduler.new
      @commands["dfs"] = SearchCommand.new(self, dfs_scheduler)
      bfs_scheduler = BfsScheduler.new
      @commands["bfs"] = SearchCommand.new(self, bfs_scheduler)
    end

    def run(filename)
      parser = Parser.new
      @current_directory = @root_directory =
        open(filename) { |f| parser.parse(f) }
      each_cmd do |cmd, *args|
        begin
          @commands[cmd].exec(args)
        rescue
          print($!, "\n")
        end
      end
    end

    def get_file(path)
      abs_path = File.expand_path(path, current_directory.path)
      return root_directory if abs_path == "/"
      rel_path = abs_path.sub(/^\//n, "")
      return root_directory.get_descendant(rel_path)
    end

    def print(*args)
      $stdout.print(*args)
    end

    private

    def each_cmd
      while line = readline
        yield(Shellwords.shellwords(line))
      end
    end

    def readline
      if $stdin.tty?
        $stdout.print("ls−lR> ")
        $stdout.flush
      end
      return $stdin.gets
    end
  end
end

if $0 == __FILE__
  if ARGV.length < 1
    STDERR.printf("usage: %s <ls−lR file>\n", $0)
    exit(1)
  end
  filename = ARGV.shift
  shell = LsLR::Shell.new
  shell.run(filename)
end
```