10/10 EXTREMELY WELL DONE!

ps1.scm


```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; 6.891 PS1
;;;
;;; Dan R. K. Ports <drkp@mit.edu>
;;;
;;; $Date: 2007-02-14 12:25:52 -0500 (Wed, 14 Feb 2007) $ $Rev: 2254 $
;;;
;;; Collaborators: '(iyzhang amdragon)  THANKS
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(load "regexp.scm")

;;;
;;; Problem 1                ] 2/2 NICE.
;;;

(define (r:* expr)
  (r:repeat 0 #f expr))

(define (r:+ expr)
  (r:repeat 1 #f expr))

#|
;; A couple simple test cases.

(pp (r:grep
     (r:seq (r:quote " ") (r:* (r:quote "cat")) (r:quote "dog"))
     "tests.txt"))
("[09]. catdogcat" "[10]. catcatdogdog"
                   "[11]. dogdogcatdogdog"
                   "[12]. catcatcatdogdogdog")


(pp (r:grep
     (r:seq (r:quote " ") (r:+ (r:quote "cat")) (r:quote "dog"))
     "tests.txt"))
("[09]. catdogcat" "[10]. catcatdogdog" "[12]. catcatcatdogdogdog")

|#


;;;
;;; Problem 2                ] 2/2 WELL DONE!
;;;
;;; a. We're trying to implement r:repeat here; we can't implement its
;;; base case in terms of itself!   RIGHT.
;;;
;;; b. Alyssa's proposal generates a much longer regular expression
;;; than Bonnie's (a regexp of up to O(n^2) characters, where n is the
;;; maximum number of repetitions, vs O(n) for Bonnie's). Besides
;;; making the output larger and clumsier, it will be slower to
;;; generate.  AND THE CODE IS MORE COMPLICATED TOO.
;;;
;;; c. Ben's proposal uses interval notation, which is actually
;;; provided by the BRE spec, unlike the '?' and '|' operators, which
;;; aren't. It also results in a much shorter regular expression
;;; (O(log n)), which will be much clearer since the interval notation  EXCELLENT!
;;; directly expresses the intent of the repeat clause.
;;;
;;; d. The new r:repeat follows:

(define (r:repeat min max expr)
  (if (not (exact-nonnegative-integer? min))
      (error "Min must be non-negative integer:" min))
  (if max
      (begin
        (if (not (exact-nonnegative-integer? max))
            (error "Max must be non-negative integer:" max))
        (if (not (<= min max))
            (error "Min not less than max:" min max))))
  (r:seq expr
         "\\{"
         (number->string min)
         ","
         (if max (number->string max) "")
         "\\}"))
```

OK, BUT expr{n,n} ≡ expr{n}

TEST CASES?

```
;;;
;;; Problem 3                ] 2/2 VERY WELL DONE!
;;;
;;; We'll eliminate all the unnecessary parentheses and achieve some
;;; design elegance by throwing most of the existing implementation
;;; out and starting anew.   AWESOME!
;;;
;;; Specifically, we'll redefine the pattern-building constructs to
;;; represent their pattern abstractly rather than as a regexp string,
;;; and provide a function for converting the abstract representation
;;; to a BRE string. Besides letting us adapt easily to generating
;;; EREs (but that's Problem 5!), we can examine the structure of the
;;; abstract representation and, by comparing its nesting to the
;;; precedence rules for BREs, apply parens only when necessary.
;;;
;;; Each element in the regular expression has a precedence, where 1
;;; is the most tightly binding. If a sub-expression (in, e.g. a seq or
;;; alt expression) is less tightly binding than its parent, we need
;;; to add parentheses around it. This generates the minimal
;;; parenthesation.   OR "parenthesization"
;;;

;;; Functions to generate the abstract representation

(define (r:dot)
  (list 'dot))

(define (r:bol)
  (list 'bol))

(define (r:eol)
  (list 'eol))

(define (r:char-from char-set)
  (list 'char-from char-set))

(define (r:char-not-from char-set)
  (list 'char-not-from char-set))

(define (r:seq . exprs)
  (cons 'seq exprs))

(define (r:alt . exprs)
  (cons 'alt exprs))

(define (r:quote string)
  (apply r:seq (map r:char-from
                    (map char-set (string->list string)))))

(define (r:repeat min max expr)
  (if (not (exact-nonnegative-integer? min))
```

```
      (error "Min must be non-negative integer:" min))
  (if max
      (begin
        (if (not (exact-nonnegative-integer? max))
            (error "Max must be non-negative integer:" max))
        (if (not (<= min max))
            (error "Min not less than max:" min max))))
  (list 'repeat min max expr))

(define (r:* expr)
  (r:repeat 0 #f expr))

(define (r:+ expr)
  (r:repeat 1 #f expr))


;;; Precedence rules

(define (r:precedence expr)
  (case (car expr)
    ((dot bol eol char-from char-not-from) 1)
    ((repeat) 2)
    ((seq)
     (if (null? (cddr expr))
         ;; A one-element sequence does not truly have the sequence
         ;; nature; treat it as whatever it contains
         (r:precedence (cadr expr))
         3))
    ((alt) 4)))

;;; Conversion function
;;;  The rules for parenthesizing and characters to escape are
;;;  specified at the beginning of this procedure; we'll pull them out
;;;  in the interests of generality later.

(define (r:bre-str expr)
  (define (parenthesize parent-prec child-prec expr)
    (if (> child-prec parent-prec)
        (string-append "\\(" expr "\\)")
        expr))
  (define escape-characters (string->char-set ".(\\^$"))
  (case (car expr)
    ((dot) ".")
    ((bol) "^")
    ((eol) "$")
    ((char-from)
     (let ((members (char-set-members (cadr expr))))
       (cond ((not (pair? members)) (r:seq))
             ((not (pair? (cdr members)))
              (escape escape-characters (string (car members))))
             (else (%char-from #f members)))))
    ((char-not-from)
     (let ((members (char-set-members (cadr expr))))
       (%char-from #t members)))
    ((seq)
     (apply string-append
      (map (lambda (x) (parenthesize (r:precedence expr)
                                     (r:precedence x)
                                     (r:bre-str x))) (cdr expr))))
    ((repeat)
     (let ((min (second expr)) (max (third expr)) (subexpr (fourth expr)))
       (string-append
        (parenthesize (r:precedence expr)
                      (r:precedence subexpr)
                      (r:bre-str subexpr))
        "\\{"
        (number->string min)
        ","
        (if max (number->string max) "")
        "\\}")))
    ((alt)
     (apply string-append
            (intersperse "\\|"
                         (map (lambda (subexpr)
                                (parenthesize (r:precedence expr)
                                              (r:precedence subexpr)
                                              (r:bre-str subexpr)))
                              (cdr expr)))))
    (else (error "Unknown regexp element"))))


;;; Random utility functions

(define (intersperse elt lst)
  (cond ((null? lst) '())
        ((null? (cdr lst)) lst)
        (else
         (append (list (car lst) elt)
                 (intersperse elt (cdr lst))))))

(define (escape charset str)
  (list->string
   (append-map
    (lambda (char) (if (char-set-member? charset char)
                       (list #\\ char)
                       (list char)))
    (string->list str))))

(define (assert x msg)
  (if (not x) (error msg)))

(define (r:grep-command cmd expr filename)
  (let ((port (open-output-string)))
    (and (= (run-synchronous-subprocess cmd
                                        (list "-e" expr (->namestring filename))
                                        'output port)
            0)
         (r:split-lines (get-output-string port)))))

(define (r:grep expr filename)
  (r:grep-command "grep" expr filename))

(define (r:egrep expr filename)
  (r:grep-command "egrep" expr filename))

;;; Test cases
;;;
;;; For simplicity, we do our testing by checking the generated
;;; BRE expression against what it should be, rather than by running
;;; grep; this also lets us check that the parentheses are
;;; minimized. For completeness, we do some testing using grep and
;;; egrep; these are at the end of Problem 5.

(define (r:test fn expr str)
  (assert (equal? (fn expr) str)
          (string-append
           "Test failed: expected: "
           str
           " got: "
           (fn expr))))
```

```
  #t)

(r:test r:bre-str (r:dot) ".")
(r:test r:bre-str (r:bol) "^")
(r:test r:bre-str (r:eol) "$")
(r:test r:bre-str (r:char-from (char-set #\a)) "a")
(r:test r:bre-str (r:quote "a") "a")
(r:test r:bre-str (r:char-from (char-set #\a #\b)) "[ab]")
(r:test r:bre-str (r:char-not-from (char-set #\a)) "[^a]")
(r:test r:bre-str (r:char-not-from (char-set #\a #\b)) "[^ab]")
(r:test r:bre-str (r:seq (r:bol) (r:dot) (r:eol)) "^.$")
(r:test r:bre-str (r:seq (r:bol) (r:quote "abc") (r:eol)) "^abc$")
(r:test r:bre-str (r:seq (r:bol) (r:quote "abc") (r:eol)) "^abc$")
(r:test r:bre-str (r:repeat 2 5 (r:quote "a")) "a\\{2,5\\}")
(r:test r:bre-str (r:repeat 0 5 (r:quote "a")) "a\\{0,5\\}")
(r:test r:bre-str (r:repeat 5 5 (r:quote "a")) "a\\{5,5\\}")
(r:test r:bre-str (r:repeat 5 #f (r:quote "a")) "a\\{5,\\}")
(r:test r:bre-str (r:repeat 2 5 (r:quote "abc")) "\\(abc\\)\\{2,5\\}")
(r:test r:bre-str (r:repeat 2 5 (r:seq (r:quote "abc") (r:dot)))
        "\\(abc.\\)\\{2,5\\}")
(r:test r:bre-str (r:alt (r:quote "abc") (r:dot)) "abc\\|.")
(r:test r:bre-str
        (r:seq (r:alt (r:quote "abc") (r:dot)) (r:quote "def"))
        "\\(abc\\|.\\)def")


;;;
;;; Problem 4
;;;
;;; We implement backreferences using a (r:tag-backref) form to assign
;;; a name to a subexpression, and a (r:ref-backref) to backreference
;;; it.
;;;
;;; As we convert the abstract representation to a BRE, we keep state:
;;; the number of left-parentheses generated so far, in order to
;;; determine backreference numbers, and an alist mapping tags to
;;; their number.
;;;

(define (r:tag-backref name expr)
  (list 'tag-backref name expr))

(define (r:ref-backref name)
  (list 'ref-backref name))

(define (r:precedence-backref expr)
  (case (car expr)
    ((ref-backref) 1)
    ((tag-backref) 1)
    (else
     (r:precedence expr))))

(define (r:bre-backref-str expr)
  (define escape-characters (string->char-set ".(\\^$"))
  (let ((num-parens 0)
        (backrefs '()))
    (define (bre-backref-str-internal expr)
      (define (parenthesize-and-evaluate parent-prec child-prec expr)
        (if (> child-prec parent-prec)
            (begin
              (set! num-parens (+ 1 num-parens))
              (string-append "\\("
                             (bre-backref-str-internal expr)
                             "\\)"))
            (bre-backref-str-internal expr)))
      (case (car expr)
        ((dot) ".")
        ((bol) "^")
        ((eol) "$")
        ((char-from)
         (let ((members (char-set-members (cadr expr))))
           (cond ((not (pair? members)) (r:seq))
                 ((not (pair? (cdr members)))
                  (escape escape-characters (string (car members))))
                 (else (%char-from #f members)))))
        ((char-not-from)
         (let ((members (char-set-members (cadr expr))))
           (%char-from #t members)))
        ((seq)
         (apply string-append
                (map (lambda (x) (parenthesize-and-evaluate
                                  (r:precedence-backref expr)
                                  (r:precedence-backref x)
                                  x))
                     (cdr expr))))
        ((repeat)
         (let ((min (second expr)) (max (third expr)) (subexpr (fourth expr)))
           (string-append
            (parenthesize-and-evaluate
             (r:precedence-backref expr)
             (r:precedence-backref subexpr)
             subexpr)
            "\\{"
            (number->string min)
            ","
            (if max (number->string max) "")
            "\\}")))
        ((alt)
         (apply string-append
                (intersperse "\\|"
                             (map (lambda (subexpr)
                                    (parenthesize-and-evaluate
                                     (r:precedence-backref expr)
                                     (r:precedence-backref subexpr)
                                     subexpr))
                                  (cdr expr)))))
        ((tag-backref)
         (set! num-parens (+ 1 num-parens))
         (set! backrefs (append backrefs (list (cons (second expr) num-parens))))
         (string-append
          "\\("
          (bre-backref-str-internal (third expr))
          "\\)"))
        ((ref-backref)
         (if (assq (second expr) backrefs)
             (string-append
              "\\"
              (number->string (cdr (assq (second expr) backrefs))))
             (error (string-append "Unknown backref: " (symbol->string
                                                        (second expr))))))
        (else (error "Unknown regexp element"))))
    (bre-backref-str-internal expr)))


(r:test r:bre-backref-str (r:dot) ".")
(r:test r:bre-backref-str (r:bol) "^")
(r:test r:bre-backref-str (r:eol) "$")
(r:test r:bre-backref-str (r:char-from (char-set #\a)) "a")
```

EXCELLENT!

2/2 WELL DONE!

COMMON IDIOMS

EEK!

EEK!

name

```
(r:test r:bre-backref-str (r:quote "a") "a")
(r:test r:bre-backref-str (r:char-from (char-set #\a #\b)) "[ab]")
(r:test r:bre-backref-str (r:char-not-from (char-set #\a)) "[^a]")
(r:test r:bre-backref-str (r:char-not-from (char-set #\a #\b)) "[^ab]")
(r:test r:bre-backref-str (r:seq (r:bol) (r:dot) (r:eol)) "^.$")
(r:test r:bre-backref-str (r:seq (r:bol) (r:quote "abc") (r:eol)) "^abc$")
(r:test r:bre-backref-str (r:seq (r:bol) (r:quote "abc") (r:eol)) "^abc$")
(r:test r:bre-backref-str (r:repeat 2 5 (r:quote "a")) "a\\{2,5\\}")
(r:test r:bre-backref-str (r:repeat 0 5 (r:quote "a")) "a\\{0,5\\}")
(r:test r:bre-backref-str (r:repeat 5 5 (r:quote "a")) "a\\{5,5\\}")
(r:test r:bre-backref-str (r:repeat 5 #f (r:quote "a")) "a\\{5,\\}")
(r:test r:bre-backref-str
        (r:repeat 2 5 (r:quote "abc")) "\\(abc\\)\\{2,5\\}")
(r:test r:bre-backref-str
        (r:repeat 2 5 (r:seq (r:quote "abc") (r:dot))) "\\(abc.\\)\\{2,5\\}")
(r:test r:bre-backref-str
        (r:alt (r:quote "abc") (r:dot)) "abc\\|.")
(r:test r:bre-backref-str
        (r:seq (r:alt (r:quote "abc") (r:dot)) (r:quote "def"))
        "\\(abc\\|.\\)def")
(r:test r:bre-backref-str
        (r:seq (r:tag-backref 'foo (r:dot)) (r:ref-backref 'foo))
        "\\(.\\)\\1")
(r:test r:bre-backref-str
        (r:* (r:seq (r:tag-backref 'foo (r:dot)) (r:ref-backref 'foo)))
        "\\(\\(.\\)\\2\\)\\{0,\\}")


;;;
;;; Problem 5
;;;
;;; a) BREs require the various metacharacters ('(', ')', '\', ...) to
;;; be preceded by backslashes; EREs require them not to
;;; be. Similarly, if we need to insert a literal of one of those
;;; characters, in an ERE they must be escaped with a backslash, and
;;; in a BRE they must not be. [BREs also don't support the '?' and '*'
;;; metacharacters, but we don't use those anyway.] Technically, BREs
;;; don't support alternation (though GNU grep provides it anyway), so
;;; we shouldn't use that either, but we blithely ignore that.
;;;
;;; b) Our redesign from Problem 3 gives us the flexibility to
;;; implement both BRE and ERE backends, since we have an abstract
;;; representation. The only difference is the use of backslashes.
;;;
;;; c) Implementation follows;
;;;

(define (r:bre-or-ere-str expr escape-characters symbols)
  (let ((num-parens 0)
        (backrefs '()))
    (define (bre-backref-str-internal expr)
      (define (parenthesize-and-evaluate parent-prec child-prec expr)
        (if (> child-prec parent-prec)
            (begin
              (set! num-parens (+ 1 num-parens))
              (string-append (cdr (assq 'left-paren symbols))
                             (bre-backref-str-internal expr)
                             (cdr (assq 'right-paren symbols))))
            (bre-backref-str-internal expr)))
      (case (car expr)
        ((dot) ".")
        ((bol) "^")
        ((eol) "$")
        ((char-from)
         (let ((members (char-set-members (cadr expr))))
           (cond ((not (pair? members)) (r:seq))
                 ((not (pair? (cdr members)))
                  (escape escape-characters (string (car members))))
                 (else (%char-from #f members)))))
        ((char-not-from)
         (let ((members (char-set-members (cadr expr))))
           (%char-from #t members)))
        ((seq)
         (apply string-append
                (map (lambda (x) (parenthesize-and-evaluate
                                  (r:precedence-backref expr)
                                  (r:precedence-backref x)
                                  x))
                     (cdr expr))))
        ((repeat)
         (let ((min (second expr)) (max (third expr)) (subexpr (fourth expr)))
           (string-append
            (parenthesize-and-evaluate
             (r:precedence-backref expr)
             (r:precedence-backref subexpr)
             subexpr)
            (cdr (assq 'left-brace symbols))
            (number->string min)
            ","
            (if max (number->string max) "")
            (cdr (assq 'right-brace symbols)))))
        ((alt)
         (apply string-append
                (intersperse (cdr (assq 'pipe symbols))
                             (map (lambda (subexpr)
                                    (parenthesize-and-evaluate
                                     (r:precedence-backref expr)
                                     (r:precedence-backref subexpr)
                                     subexpr))
                                  (cdr expr)))))
        ((tag-backref)
         (set! num-parens (+ 1 num-parens))
         (set! backrefs (append backrefs (list (cons (second expr) num-parens))))
         (string-append
          (cdr (assq 'left-paren symbols))
          (bre-backref-str-internal (third expr))
          (cdr (assq 'right-paren symbols))))
        ((ref-backref)
         (if (assq (second expr) backrefs)
             (string-append
              (cdr (assq 'backref symbols))
              (number->string (cdr (assq (second expr) backrefs))))
             (error (string-append "Unknown backref: " (symbol->string
                                                        (second expr))))))

        (else (error "Unknown regexp element"))))
    (bre-backref-str-internal expr)))

(define (r:bre-str expr)
  (r:bre-or-ere-str expr
                    (string->char-set ".[\\^$")
                    (list (cons 'left-paren "\\(")
                          (cons 'right-paren "\\)")
                          (cons 'left-brace "\\{")
                          (cons 'right-brace "\\}")
                          (cons 'pipe "\\|")
                          (cons 'backref "\\"))))

(define (r:ere-str expr)
  (r:bre-or-ere-str expr
```

```
                (string->char-set ".[\\^$(){}|") — Good
                (list (cons 'left-paren "(")
                      (cons 'right-paren ")")
                      (cons 'left-brace "{")
                      (cons 'right-brace "}")
                      (cons 'pipe "|")
                      (cons 'backref "\\")))


;;; Test cases


(r:test r:bre-str (r:dot) ".")
(r:test r:bre-str (r:bol) "^")
(r:test r:bre-str (r:eol) "$")
(r:test r:bre-str (r:char-from (char-set #\a)) "a")
(r:test r:bre-str (r:quote "a") "a")
(r:test r:bre-str (r:char-from (char-set #\a #\b)) "[ab]")
(r:test r:bre-str (r:char-not-from (char-set #\a)) "[^a]")
(r:test r:bre-str (r:char-not-from (char-set #\a #\b)) "[^ab]")
(r:test r:bre-str (r:seq (r:bol) (r:dot) (r:eol)) "^.$")
(r:test r:bre-str (r:seq (r:bol) (r:quote "abc") (r:eol)) "^abc$")
(r:test r:bre-str (r:seq (r:bol) (r:quote "abc") (r:eol)) "^abc$")
(r:test r:bre-str (r:repeat 2 5 (r:quote "a")) "a\\{2,5\\}")
(r:test r:bre-str (r:repeat 0 5 (r:quote "a")) "a\\{0,5\\}")
(r:test r:bre-str (r:repeat 5 5 (r:quote "a")) "a\\{5,5\\}")
(r:test r:bre-str (r:repeat 5 #f (r:quote "a")) "a\\{5,\\}")
(r:test r:bre-str (r:repeat 2 5 (r:quote "abc")) "\\(abc\\)\\{2,5\\}")
(r:test r:bre-str
        (r:repeat 2 5 (r:seq (r:quote "abc") (r:dot))) "\\(abc.\\)\\{2,5\\}")
(r:test r:bre-str
        (r:alt (r:quote "abc") (r:dot)) "abc\\|.")
(r:test r:bre-str
        (r:seq (r:alt (r:quote "abc") (r:dot)) (r:quote "def")) "\\(abc\\|.\\)def")
(r:test r:bre-str
        (r:seq (r:tag-backref 'foo (r:dot)) (r:ref-backref 'foo)) "\\(.\\)\\1")
(r:test r:bre-str
        (r:* (r:seq (r:tag-backref 'foo (r:dot)) (r:ref-backref 'foo)))
        "\\(\\(.\\)\\2\\)\\{0,\\}")

(r:test r:ere-str (r:dot) ".")
(r:test r:ere-str (r:bol) "^")
(r:test r:ere-str (r:eol) "$")
(r:test r:ere-str (r:char-from (char-set #\a)) "a")
(r:test r:ere-str (r:quote "a") "a")
(r:test r:ere-str (r:char-from (char-set #\a #\b)) "[ab]")
(r:test r:ere-str (r:char-not-from (char-set #\a)) "[^a]")
(r:test r:ere-str (r:char-not-from (char-set #\a #\b)) "[^ab]")
(r:test r:ere-str (r:seq (r:bol) (r:dot) (r:eol)) "^.$")
(r:test r:ere-str (r:seq (r:bol) (r:quote "abc") (r:eol)) "^abc$")
(r:test r:ere-str (r:seq (r:bol) (r:quote "abc") (r:eol)) "^abc$")
(r:test r:ere-str (r:repeat 2 5 (r:quote "a")) "a{2,5}")
(r:test r:ere-str (r:repeat 0 5 (r:quote "a")) "a{0,5}")
(r:test r:ere-str (r:repeat 5 5 (r:quote "a")) "a{5,5}")
(r:test r:ere-str (r:repeat 5 #f (r:quote "a")) "a{5,}")
(r:test r:ere-str (r:repeat 2 5 (r:quote "abc")) "(abc){2,5}")
(r:test r:ere-str
        (r:repeat 2 5 (r:seq (r:quote "abc") (r:dot))) "(abc.){2,5}")
(r:test r:ere-str
        (r:alt (r:quote "abc") (r:dot)) "abc|.")
(r:test r:ere-str
        (r:seq (r:alt (r:quote "abc") (r:dot)) (r:quote "def")) "(abc|.)def")
(r:test r:ere-str
        (r:seq (r:tag-backref 'foo (r:dot)) (r:ref-backref 'foo)) "(.)\\1")
(r:test r:ere-str
        (r:* (r:seq (r:tag-backref 'foo (r:dot)) (r:ref-backref 'foo)))
        "((.)\\2){0,}")


#|

(pp (r:grep (r:bre-str (r:seq (r:quote "a") (r:dot) (r:quote "c"))) "tests.txt"))

("[00]. abc" "[01]. aac"
             "[02]. acc"
             "[03]. zzzaxcqqq"
             "[10]. catcatdogdog"
             "[12]. catcatcatdogdogdog")
;Unspecified return value


(pp (r:grep
     (r:bre-str (r:alt (r:quote "foo") (r:quote "bar") (r:quote
                                                        "baz")))
     "tests.txt"))
("[05]. foo" "[06]. bar" "[07]. foo bar baz quux")
;Unspecified return value


(pp (r:grep
     (r:bre-str (r:repeat 3 5 (r:alt (r:quote "cat") (r:quote "dog"))))
     "tests.txt"))
("[09]. catdogcat" "[10]. catcatdogdog"
                   "[11]. dogdogcatdogdog"
                   "[12]. catcatcatdogdogdog"
                   "[13]. acatdogdogcats"
                   "[14]. ifacatdogdogs"
                   "[15]. acatdogdogsme")
;Unspecified return value

(pp
 (r:grep (r:bre-str
          (r:seq (r:quote " ")
                 (r:repeat 3 5 (r:alt (r:quote "cat") (r:quote "dog")))
                 (r:eol)))
         "tests.txt"))
("[09]. catdogcat" "[10]. catcatdogdog" "[11]. dogdogcatdogdog")
;Unspecified return value

Excellent!




(pp (r:egrep (r:ere-str (r:seq (r:quote "a") (r:dot) (r:quote "c"))) "tests.txt"))

("[00]. abc" "[01]. aac"
             "[02]. acc"
             "[03]. zzzaxcqqq"
             "[10]. catcatdogdog"
             "[12]. catcatcatdogdogdog")
;Unspecified return value


(pp (r:egrep
     (r:ere-str (r:alt (r:quote "foo") (r:quote "bar") (r:quote
                                                        "baz")))
     "tests.txt"))
("[05]. foo" "[06]. bar" "[07]. foo bar baz quux")
;Unspecified return value
```

```
(pp (r:egrep
     (r:ere-str (r:repeat 3 5 (r:alt (r:quote "cat") (r:quote "dog"))))
     "tests.txt"))
("[09]. catdogcat" "[10]. catcatdogdog"
                   "[11]. dogdogcatdogdog"
                   "[12]. catcatcatdogdogdog"
                   "[13]. acatdogdogcats"
                   "[14]. ifacatdogdogs"
                   "[15]. acatdogdogsme")
;Unspecified return value

(pp
 (r:egrep (r:ere-str
           (r:seq (r:quote " ")
                  (r:repeat 3 5 (r:alt (r:quote "cat") (r:quote "dog")))
                  (r:eol)))
          "tests.txt"))
("[09]. catdogcat" "[10]. catcatdogdog" "[11]. dogdogcatdogdog")
;Unspecified return value
|#
```

VERY WELL DONE!