Readings:
  - Review SICP Chapter 1 (especially section 1.3)
  - Software Design for Flexibility (SDF)
  - Chapter 2 (Elementary Flexibility Strategies)
  - MIT/GNU Scheme documentation, sections 5 and 6 (characters and strings)
  - Debian GNU/Linux info on regular expressions from the grep man page (attached). This is sane.

Code:
  - function-combinators.scm, regexp.scm, tests.txt

Documentation:
  - The MIT/GNU Scheme installation and documentation can be found online at http://www.gnu.org/software/mit-scheme/
  - The (insane) POSIX manual page for regular expressions: http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html

Compatibility:
  - See note about Mac support in mac-support.txt

To do:
  - The reading material for this week is foundational. Read Chapter 2 in SDF. Sections 2.1 and 2.2 are needed to do this problem set. Then download the code for the problem set from the class website.
Just to repeat what was said for problem set 0:

In general, you need to make a directory (folder, in modern usage) on your computer for the problem set (I would personally call it "~/6.945/ps01/") and a subdirectory called "code" (so you would have the directory "~/6.945/ps01/code/"). Download the problem set text into "~/6.945/ps01/" and the code into "~/6.945/ps01/code/". There will be a file in the code directory named "load.scm". You can load the code you need into the Scheme system by pointing the Scheme at the code directory, with executing (cd "~/6.945/ps01/code/") and then executing (load "load"). The code files you load will contain some extra material, not in the text, that support the problem set.

In this case, the procedure r:grep is not described in the text, because it uses features special to MIT/GNU Scheme. So I suggest that you scan the code before starting the problems. Also, the file "test.txt" may be used in the problem set.

Note: Loading of the support files for the problem set by loading "load.scm" reinitializes the top-level environment of Scheme, so you will lose any definitions you have made in that environment. But this will not lose buffers in your EMACS or EDWIN, so your code is not lost. Of course, you should write out the files you are building regularly, probably in your directory "~/6.945/ps01/".

To Do

Exercise 2.1: Arity Repair

Exercise 2.2: Useful Combinators

Exercise 2.3: Adding * and + to regular expressions

Exercise 2.4: A subtle bug, one bad joke, two tweaks, and a revelation
The following is an excerpt from the Debian GNU/Linux man page on grep.

REGULAR EXPRESSIONS

A regular expression is a pattern that describes a set of strings. Regular expressions are constructed analogously to arithmetic expressions, by using various operators to combine smaller expressions.

Grep understands three different versions of regular expression syntax: "basic," "extended," and "perl." In GNU grep, there is no difference in available functionality using either of the first two syntaxes. In other implementations, basic regular expressions are less powerful. The following description applies to extended regular expressions; differences for basic regular expressions are summarized afterwards. Perl regular expressions add additional functionality, but the implementation used here is undocumented and is not compatible with other grep implementations.

The fundamental building blocks are the regular expressions that match a single character. Most characters, including all letters and digits, are regular expressions that match themselves. Any metacharacter with special meaning may be quoted by preceding it with a backslash.

A bracket expression is a list of characters enclosed by [ and ]. It matches any single character in that list; if the first character of the list is the caret ^ then it matches any character not in the list. For example, the regular expression [0123456789] matches any single digit.

Within a bracket expression, a range expression consists of two characters separated by a hyphen. It matches any single character that sorts between the two characters, inclusive, using the locale’s collating sequence and character set. For example, in the default C locale, [a-d] is equivalent to [abcd]. Many locales sort characters in dictionary order, and in these locales [a-d] is typically not equivalent to [abcd]; it might be equivalent to [aBbCcDd], for example. To obtain the traditional interpretation of bracket expressions, you can use the C locale by setting the LC_ALL environment variable to the value C.

Finally, certain named classes of characters are predefined within bracket expressions, as follows. Their names are self explanatory, and they are [:alnum:], [:alpha:], [:cntrl:], [:digit:], [:graph:], [:lower:], [:print:], [:punct:], [:space:], [:upper:], and [:xdigit:]. For example, [:alnum:] means [0-9A-Za-z], except the latter form depends upon the C locale and the ASCII character encoding, whereas the former is independent of locale and character set. (Note that the brackets in these class names are part of the symbolic names, and must be included in addition to the brackets delimiting the bracket list.) Most metacharacters lose their special meaning inside lists. To include a literal ] place it first in the list. Similarly, to include a literal ^ place it anywhere but first. Finally, to include a literal - place it last.
The period . matches any single character. The symbol \w is a synonym for [[:alnum:]] and \W is a synonym for [^[[:alnum]]].

The caret ^ and the dollar sign $ are metacharacters that respectively match the empty string at the beginning and end of a line. The symbols \< and \> respectively match the empty string at the beginning and end of a word. The symbol \b matches the empty string at the edge of a word, and \B matches the empty string provided it’s not at the edge of a word.

A regular expression may be followed by one of several repetition operators:

? The preceding item is optional and matched at most once.
* The preceding item will be matched zero or more times.
+ The preceding item will be matched one or more times.
{n} The preceding item is matched exactly n times.
{n,} The preceding item is matched n or more times.
{n,m} The preceding item is matched at least n times, but not more than m times.

Two regular expressions may be concatenated; the resulting regular expression matches any string formed by concatenating two substrings that respectively match the concatenated subexpressions.

Two regular expressions may be joined by the infix operator |; the resulting regular expression matches any string matching either subexpression.

Repetition takes precedence over concatenation, which in turn takes precedence over alternation. A whole subexpression may be enclosed in parentheses to override these precedence rules.

The backreference \n, where n is a single digit, matches the substring previously matched by the nth parenthesized subexpression of the regular expression.

In basic regular expressions the metacharacters ?, +, {, |, (, and ) lose their special meaning; instead use the backslashed versions \\?, \\+, \\{, \|, \(, and \).

Traditional egrep did not support the { metacharacter, and some egrep implementations support \{ instead, so portable scripts should avoid { in egrep patterns and should use {{} to match a literal {}.

GNU egrep attempts to support traditional usage by assuming that { is not special if it would be the start of an invalid interval specification. For example, the shell command egrep '{1' searches for the two-character string {1 instead of reporting a syntax error in the regular expression. POSIX.2 allows this behavior as an extension, but portable scripts should avoid it.
;;; Scheme Regular Expression Language Implementation -- regexp.scm

(define (r:dot) "\.")
(define (r:bol) "\^")
(define (r:eol) "\$")

(define (r:quote string)
  (r:seq
   (list->string
    (append-map (lambda (char)
                  (if (memv char chars-needing-quoting)
                      (list \ char)
                      (list char)))
                (string->list string))))

(define chars-needing-quoting
  '(#\. #\[ #\ #\^ #\$ #\*))

(define (r:char-from string)
  (case (string-length string)
    ((0) (r:seq))
    ((1) (r:quote string))
    (else
     (bracket string
               (lambda (members)
                 (if (lset= eqv? '(#\- #\^) members)
                     '(#\- #\^)
                     (quote-bracketed-contents members)))))))

(define (r:char-not-from string)
  (bracket string
            (lambda (members)
              (cons #\^ (quote-bracketed-contents members)))))

(define (bracket string procedure)
  (list->string
   (append '(#\[
             (procedure (string->list string))
            '(#\])))

(define (quote-bracketed-contents members)
  (let ((optional
          (lambda (char) (if (memv char members) (list char) '(())))))
   (append (optional #\])
          (remove (lambda (c)
                    (memv c chars-needing-quoting-in-brackets))
                members)
          (optional #\^)
          (optional #\-)))))

(define chars-needing-quoting-in-brackets
  '(#\[ #\^ #\]))
;; Means of combination for patterns

(define (r:seq . exprs)
  (string-append "\(" (apply string-append exprs) "\)"))

;; An extension to POSIX basic regular expressions.
;; Supported by GNU grep and possibly others.
(define (r:alt . exprs)
  (if (pair? exprs)
      (apply r:seq
        (cons (car exprs)
          (append-map (lambda (expr)
                        (list "\|" expr))
                    (cdr exprs))))
      (r:seq)))

(define (r:repeat min max expr)
  (apply r:seq
    (append (make-list min expr)
      (if (eqv? max min)
          ()
          (if max
              (make-list (- max min)
                (r:alt expr ""))
              (list expr "*")))))
;;; Using system’s grep.
(define (write-bourne-shell-grep-command expr filename)
  (display (bourne-shell-grep-command-string expr filename)))

(define (bourne-shell-grep-command-string expr filename)
  (string-append "grep -e "
                 (bourne-shell-quote-string expr)
                 " "
                 filename))

;;; Works for any string without newlines.
(define (bourne-shell-quote-string string)
  (list->string
   (append (list #\')
     (append-map (lambda (char)
                   (if (char=? char #\')
                       (list #\' #\" char #\")
                       (list char)))
      (string->list string))
   (list #\'))))

;;; This is MIT/Scheme specific and compatible with grep for the
;;; purposes of this code.

(load-option 'synchronous-subprocess)

(define (r:grep expr filename)
  (let ((port (open-output-string)))
    (and (= (run-shell-command
              (bourne-shell-grep-command-string expr filename)
              'output port) 0)
         (r:split-lines (get-output-string port))))))

(define (r:split-lines string)
  (reverse
   (let ((i 0) (lines '()))
     (if (< i end)
       (let ((j (substring-find-next-char string i end #\newline)))
         (if j
           (loop (+ j 1)
                 (cons (substring string i j) lines))
           (cons (substring string i end) lines)))))

)
# | ;; For example...

(pp (r:grep (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

;; And...

(pp (r:grep (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: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:seq " "  
(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
(pp
  (r:grep
   (let ((digit
       (r:char-from "0123456789")))
    (r:seq (r:bol)
       (r:quote "[")
      digit
      digit
      (r:quote "]")
      (r:quote ".")
      (r:quote " ")
      (r:char-from "ab")
      (r:repeat 3 5 (r:alt "cat" "dog"))
      (r:char-not-from "def")
      (r:eol)))
"tests.txt")
("[13]. acatdogdogcats")
;Unspecified return value
\#
This is the file tests.txt

[00]. abc
[01]. aac
[02]. acc
[03]. zzzaxcqqq
[04]. abdabec

[05]. foo
[06]. bar
[07]. foo bar baz quux
[08]. anything containing them

[09]. catdogcat
[10]. catcatdogdog
[11]. dogdogcatdogdog
[12]. catcatcatdogdogdog

[13]. acatdogdogcats
[14]. ifacatdogdogs
[15]. acatdogdogsme