6.945 Adventures in Advanced Symbolic Programming
Spring 2009

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.945 Spring 2009
Problem Set 1

Issued: Wed. 4 Feb. 2009   Due: Wed. 11 Feb. 2009

General instructions for problem sets:

A problem set for this class generally asks you to do some programming. We usually give you a program to read and extend. You should turn in your extensions, in the form of clearly annotated code and executions that demonstrate its effectiveness. We may also ask for short essays explaining an idea. Your answers to these questions must be clear and concise: good logic expressed in good English is required. Thank you.

Readings:

Review SICP chapter 1 (especially section 1.3)
MIT/GNU Scheme documentation, section 5.6 (character sets)
Debian GNU/Linux info on regular expressions
  from the grep man page (attached). This is sane.

Code:

regexp.scm, tests.txt (both attached)
Windows grep: http://gnuwin32.sourceforge.net/packages/grep.htm

Documentation:

The MIT/GNU Scheme installation and documentation can
be found online at http://www.gnu.org/software/mit-scheme/
The reference manual is in:
http://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/

The (Insane) POSIX manual page for regular expressions:
http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html

Regular Expressions

Regular expressions are ubiquitous. But the syntax of the regular-expression language is awful. There are various incompatable forms of the language and the quotation conventions are baroquen [sic]. Nevertheless, there is a great deal of useful software, for example grep, that uses regular expressions to specify the desired behavior.

Part of the value of this problem set is to experience how bad things can be. Although regular expression systems are derived from a perfectly good mathematical formalism, the particular choices made by implementers to expand the formalism into useful software systems are often disastrous: the quotation conventions adopted are highly irregular; the particularly egregious misuse of parenthesis, both for grouping and for backward reference, is a miracle to behold. In addition, attempts to increase the expressive power and address shortcomings of earlier designs have led to a proliferation of incompatible derivative languages.

In this problem set we will invent both a Lisp-friendly language for specifying regular expressions and a means of translating this language to conventional regular-expression syntax. The application is to be able to use the capabilities of systems like grep from inside the Scheme environment.

As with any language there are primitives, means of combination, and means of abstraction. Our language allows the construction of patterns that utilities like grep can match against character-string data. Because this language is embedded in Scheme we inherit all of the power of Scheme: we can use Scheme constructs to combine patterns and Scheme procedures to abstract them.

Patterns are built out of primitive patterns.
The primitive patterns are:

```
 (r:dot)
     matches any character except newline

 (r:bol)
     matches only the beginning of a line

 (r:eol)
     matches only the end of a line

 (r:quote <string>)
     matches the given string

 (r:char-from <char-set>)
     matches one character from the specified
     MIT/GNU Scheme character set

 (r:char-not-from <char-set>)
     matches one character not from the specified
     MIT/GNU Scheme character set

     MIT/GNU Scheme provides a variety of character sets,
     but you can make your own with the Scheme procedures
     char-set and string->char-set.
```

Patterns can be combined to make compound patterns:

(r:seq <pattern> ...)
This pattern matches each of the argument patterns in sequence, from left to right

(r:alt <pattern> ...)
This pattern tries each argument pattern from left to right, until one of these alternatives matches. If none matches then this pattern does not match.

(r:repeat <min> <max> <pattern>)
This pattern tries to match the given argument pattern a minimum of min times but no more than a maximum of max times. If max is given as #f then there is no maximum specified. Note that if max=min the given pattern must be matched exactly that many times.

Because these are all Scheme procedures (in the file regexp.scm) you can freely mix these with any Scheme code.

Here are some examples:

```
 Pattern:  (r:seq (r:quote "a") (r:dot) (r:quote "c"))
 Matches:  "abc" and "aac" and "acc"

 Pattern:  (r:alt (r:quote "foo") (r:quote "bar") (r:quote "baz"))
 Matches:  "foo" and "bar" and "baz"

 Pattern:  (r:repeat 3 5 (r:alt (r:quote "cat") (r:quote "dog")))
 Matches:  "catdogcat" and "catcatdogdog" and "dogdogcatdogdog"
           but not "catcatcatdogdogdog"

 Pattern:  (let ((digit
                  (r:char-from (string->char-set "0123456789"))))
             (r:seq (r:bol)
                    (r:quote "[")
                    digit
                    digit
                    (r:quote "]")
                    (r:quote ".")
                    (r:quote " ")
                    (r:char-from (char-set #\a #\b))
                    (r:repeat 3 5 (r:alt (r:quote "cat") (r:quote "dog")))
                    (r:char-not-from (char-set #\d #\e #\f))
                    (r:eol)))
 Matches:  "[09]. acatdogdogcats" but not
           "[10]. ifacatdogdogs" nor
           "[11]. acatdogdogsme"
```

In the file regexp.scm we define an interface to the grep utility, which allows a Scheme program to call grep with a regular expression (constructed from the given pattern) on a given file name. The grep utility extracts lines that contain a substring that can match the given pattern.

So, for example: given the test file test.txt (supplied) we can find the lines in the file that contain a match to the given pattern.

```
(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
```

Note that the pretty-printer (pp) returns an unspecified value, after printing the list of lines that grep found matching the pattern specified.

Problem 1.1: Warmup

In the traditional regular expression language the asterisk (*) operator following a subpattern means zero-or-more copies of the subpattern and the plus-sign (+) operator following a subpattern means one-or-more copies of the subpattern. Define Scheme procedures r:* and r:+ to take a pattern and iterate it as necessary. This can be done in terms of r:repeat.

Demonstrate your procedures on real data in complex patterns.

A Subtle Bug, One Bad Joke, Two Tweaks, and a Revelation

Ben Bitdiddle has noticed a problem with our implementation of (r:repeat <min> <max> <pattern>).

The use of (r:seq expr "\\|") at the end of the r:repeat procedure is a bit dodgy. This code fragment compiles to an ERE Alternation regular expression of the form (expr|). (See 9.4.7 of the POSIX regular expression document.)

This relies on the fact that alternation with something and nothing is the equivalent of saying "one or none". That is: (expr|) denotes one or no instances of expr. Unfortunately, this depends on an undocumented GNU extension to the formal POSIX standard for REs.

Specifically, section 9.4.3 states that a vertical line appearing immediately before a close parenthesis (or immediately after an open parenthesis) produces undefined behavior. In essence, an RE must not be a null sequence.

GNU grep just happens to Do The Right Thing (tm) when presented with (x|). Not all grep implementations are as tolerant.

Therefore, Ben asks his team of three code hackers (Louis, Alyssa and his niece Bonnie) to propose alternative workarounds. Ultimately, he proposes his own patch, which you will implement.

* Louis Reasoner suggests that a simple, elegant fix would be to replace the code fragment (r:seq expr "\\|") with a straightforward call to (r:repeat 0 1 expr).

* Alyssa P. Hacker proposes that an alternative fix would be to rewrite the else clause of r:repeat to compile (r:repeat 3 5 <x>) into the equivalent of (xxx|xxxx|xxxxx) instead of the naughty xxx(x|)(x|) non-POSIX-compliant undefined regular expression. She refers to section 9.4.7 of the POSIX regular expression doc.

* Bonnie Bitdiddle points to the question mark (?) operator in section 9.4.6.4 and proposes that a better fix would be to implement an r:? operator then replace (r:seq expr "\\|") with (r:? expr).

* Meanwhile, Ben looks closely at the RE spec and has a revelation. He proposes that r:repeat be re-implemented to emit Interval expressions. See section 9.3.6.5 of the POSIX documentation. Please try not to get sick.

Problem 1.2: The Proposals

Let’s very briefly consider each proposal:

a. Everyone in the room immediately giggles at Louis’ silly joke. What’s so funny about it? That is, what’s wrong with this idea?

   A one-sentence punchline will do.

b. What advantages does Bonnie’s proposal have over Alyssa’s in terms of both code and data?

   A brief, concise yet convincing few sentences suffices.

c. What advantage does Ben’s proposal have over all the others? Specifically, ponder which section of the POSIX document he cites versus which sections the others cite, then take a quick peek at Problem 1.5 below and consider the implications. Also, consider the size of the output strings in this new code as well as the overall clarity of the code.

   Again, a brief sentence or two is sufficient.

d. Following Ben’s proposal, re-implement r:repeat to emit Interval expressions. Hint: Scheme’s number->string procedure should be handy. Caveat: beware the backslashes.

   Show the output it generates on a few well-chosen sample inputs. Demonstrate your procedure on real data in complex patterns.

Too Much Nesting

Our program produces excessively nested regular expressions: it makes groups even when they are not necessary. For example, the following simple pattern leads to an overly complex regular expression:

```
(display (r:seq (r:quote "a") (r:dot) (r:quote "c")))
\(\(a\).\(c\)\)
```

Another problem is that BREs may involve back-references. (See 9.3.6.3 of the POSIX regular expression documentation.) A back-reference refers to a previously parenthesized subexpression. So it is important that the parenthesized subexpressions be ones explicitly placed by the author of the pattern. (Aargh! This is one of the worst ideas I (GJS) have ever heard of -- grouping, which is necessary for iteration, was confused with naming for later reference. What a crock!)

Problem 1.3: Optimization

Edit our program to eliminate as much of the unnecessary nesting as you can. Caution: there are subtle cases here that you have to watch out for. What is such a case? Demonstrate your better version of our program and show how it handles the subtleties.

Hint: Our program uses strings as its intermediate representation as well as its result. You might consider using a different intermediate representation.

Problem 1.4: Back-references

Add in a procedure for constructing back-references.
Have fun getting confused about BREs.

Standards?

> The best thing about standards is that
> there are so many to choose from.
> Tom Knight

There are also Extended Regular Expressions (EREs) defined in the POSIX regular expression documentation. Some software, such as egrep, uses this version of regular expressions. Unfortunately EREs are not a conservative extension of BREs: ERE syntax is actually inconsistent with BRE syntax! It is an interesting project to extend our Scheme pattern language so that the target can be either BREs or EREs.

Problem 1.5: Ugh!

a. What are the significant differences between BREs and EREs that make this a pain? List the differences that must be addressed.

b. How can the back end be factored so that our language can compile into either kind of regular expression, depending on what is needed? How can we maintain the abstract layer that is independent of the target regular expression language? Explain your strategy.

c. Extend our implementation to have both back ends.

Demonstrate your work by making sure that you can run egrep as well as grep, with equivalent results in cases that test the differences you found in part a.

End of Problem Set. Reference Material Follows.

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
   (call-with-output-string             ; see RefMan section 14.3
    (lambda (port)
      (let ((end (string-length string)))
        (do ((i 0 (+ i 1))) ((not (< i end))) ; see RefMan 2.9
          (let ((c (string-ref string i)))
            (if (or (char=? c #\.)
                    (char=? c #\[)
                    (char=? c #\\)
                    (char=? c #\^)
                    (char=? c #\$)
                    (char=? c #\*))
                (write-char #\\ port))
            (write-char c port))))))))

(define (r:char-from char-set)          ; see RefMan section 5.6
  (let ((members (char-set-members char-set)))
    (cond ((not (pair? members))
           (r:seq))
          ((not (pair? (cdr members)))
           (r:quote (string (car members))))
          (else
           (%char-from #f members)))))

(define (r:char-not-from char-set)
  (%char-from #t (char-set-members char-set)))

(define (%char-from negate? members)
  (let ((right? (memv #\] members))
        (caret? (memv #\^ members))
        (hyphen? (memv #\- members))
        (others
         (delete-matching-items members
                                (lambda (c)
                                  (or (char=? c #\])
                                      (char=? c #\^)
                                      (char=? c #\-))))))
    (if (and caret?
             hyphen?
             (not right?)
             (not negate?)
             (null? others))
        "[-^]"
        (string-append "["
                       (if negate? "^" "")
                       (if right? "]" "")
                       (list->string others)
                       (if caret? "^" "")
                       (if hyphen? "-" "")
                       "]"))))
```

```
;;; Means of combination for patterns

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

(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)
  (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))))
  (cond ((not max)
         (apply r:seq
                (append (make-list min expr)
                        (list expr "*"))))
        ((= max min)
         (apply r:seq (make-list min expr)))
        (else
         (apply r:seq
                (append (make-list min expr)
                        (make-list (- max min)
                                   (r:seq expr "\\|")))))))
```

```
;;; The following magic allows a program in MIT/GNU Scheme to call the
;;; grep system utility, returning the list of grep output lines to
;;; the caller.  You can make similar mechanisms to call other system
;;; utilities.

(load-option 'synchronous-subprocess)

(define (r:grep expr filename)
  (r:grep-like "grep" '() expr filename))

(define (r:egrep expr filename)
  (if (eq? microcode-id/operating-system 'nt)
      (r:grep-like "grep" '("-E") expr filename)
      (r:grep-like "egrep" '() expr filename)))

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

(define (r:split-lines string)
  (reverse
   (let ((end (string-length string)))
     (let loop ((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)))
           lines)))))

#|
;;; An alternate implementation using MIT/GNU Scheme's internal
;;; regular-expression interpreter.

(define (r:grep expr filename)
  (call-with-input-file filename
    (lambda (port)
      (let loop ((lines '()))
        (let ((line (read-line port)))
          (if (eof-object? line)
              (reverse lines)
              (loop (if (re-string-search-forward expr line #f)
                        (cons line lines)
                        lines))))))))
|#
```

```
#|
;;; For example...

;;; Note, the result of the next two requests were not in this file
;;; when the requests were made!

(pp (r:grep (r:quote "r:sex") "regexp.scm"))
("(pp (r:grep (r:quote \"r:sex\") \"regexp.scm\"))")
;Unspecified return value

(pp (r:grep (r:quote "r:seq") "regexp.scm"))
("  (r:seq"
 "\t   (r:seq))"
 "(define (r:seq . exprs)"
 "      (apply r:seq"
 "      (r:seq)))"
 "\t (apply r:seq"
 "\t (apply r:seq (make-list min expr)))"
 "\t (apply r:seq"
 "\t\t\t\t   (r:seq expr \"\\\\|\")))))))"
 "(pp (r:grep (r:quote \"r:seq\") \"regexp.scm\"))"
 "(pp (r:grep (r:seq (r:quote \"a\") (r:dot) (r:quote \"c\")) \"tests.txt\"))"
 " (r:grep (r:seq \" \""
 "    (r:seq (r:bol)")
;Unspecified return value

(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 (string->char-set "0123456789"))))
    (r:seq (r:bol)
           (r:quote "[")
           digit
           digit
           (r:quote "]")
           (r:quote ".")
           (r:quote " ")
           (r:char-from (char-set #\a #\b))
           (r:repeat 3 5 (r:alt "cat" "dog"))
           (r:char-not-from (char-set #\d #\e #\f))
           (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
```