Title

Scheme Regular Expressions

Author

Alex Shinn

Status

This SRFI is currently in *final* status. Here is an explanation of each status that a SRFI can hold. To provide input on this SRFI, please send email to `srfi-115@srfi.schemers.org`. To subscribe to the list, follow these instructions. You can access previous messages via the mailing list archive.

- Received: 2013-10-08
- Revised: 2013-11-17
- Revised: 2013-12-08
- Revised: 2014-04-24
- Revised: 2014-05-19
- Revised: 2014-06-03
- Final: 2014-07-14
- Draft: 2013-10-12--2013-12-12
- Revised to fix errata:
  - 2015-08-24 (Fix examples, grammar, etc.)
  - 2015-10-11 (Add clarifications and more examples. Update location of sample implementation.)
  - 2018-09-18 (Fix typo in sample grapheme definition.)
  - 2020-05-11 (Fix error in `regexp-split` example.)
  - 2020-08-21 (Fix code for handling regional indicator characters to account for a fix to Unicode TR29 itself. See Unicode.org message and Alex's SRFI message.)
  - **Post-finalization note**: In `regexp-replace` and `regexp-replace-all`, the `subst` is defined to be a string, integer or symbol. However, the sample implementation as well as equivalent procedures in IrRegex and SCSH also allow a list of any of these, with the results concatenated. In addition, the sample implementation accepts a procedure of one argument, which is applied to the corresponding `regexp-match` object to obtain the substituted string (as in Emacs `regexp-replace-in-string`). This can be used to avoid the ambiguity whereby `'pre` and `'post` have predefined meanings and can't refer to submatch names. Implementors are encouraged to support these two extensions.

Table of Contents

Abstract

This SRFI provides a library for matching strings with regular expressions described using the SRE "Scheme Regular Expression" notation first introduced by SCSH, and extended heavily by IrRegex.

Rationale

Regular expressions, coming from a long history of formal language theory, are today the lingua franca of simple string matching. A regular expression is an expression describing a regular language, the simplest level in the Chomsky hierarchy. They have the nice property that they can match in linear time, whereas parsers for the next level in the hierarchy require cubic time. This combined with their conciseness led them to be a popular choice for searching in editors, tools and search interfaces. Other tools may be better suited to specific purposes, but it is assumed any modern language will provide regular expression support.

SREs were first introduced in SCSH as an s-expression based alternative to the more common string based description. This format offers many advantages, including being easier to read and write (notably with structured editors), easier to compose (with no escaping issues), and faster and simpler to compile. An efficient reference implementation of this SRFI can be written in under 1000 lines of code, whereas in IrRegex the full PCRE parser alone requires over 500 lines.

Procedure Index

```
regexp   rx      regexp->sre        char-set->sre
valid-sre?       regexp?            regexp-match?
regexp-matches   regexp-matches?    regexp-search
regexp-fold      regexp-extract     regexp-split   regexp-partition
regexp-replace            regexp-replace-all
regexp-match-submatch     regexp-match-submatch-start   regexp-match-submatch-end
regexp-match-count        regexp-match->list
```

Sre Syntax Index

```
Basic Patterns
<string> seq : or |

Options
w/nocase w/case w/ascii w/unicode
w/nocapture

Repetition
optional ?      zero-or-more *
one-or-more +   at-least >=
exactly =       repeated **
```

```
Submatches
submatch $ submatch-named ->
backref

Character Sets
<char> (<string>) char-set
char-range / or | and &
complement ~ difference -

Named Character Sets
any nonl ascii   lower-case lower
upper-case upper title-case title
alphabetic       alpha numeric num
alphanumeric     alphanum alnum
punctuation      punct symbol
graphic graph    printing print
whitespace       white space
control cntrl    hex-digit xdigit

Boundary Checks
bos  eos   bol eol
bow  eow   nwb
word word+ word
bog  eog   grapheme

Non-greedy Patterns
non-greedy-optional     ??
non-greedy-zero-or-more *?
non-greedy-repeated     **?

Look Around Patterns
look-ahead look-behind neg-look-ahead neg-look-behind
```

Types and Naming Conventions

We introduce two new types, `regexp` and `regexp-match`, which are disjoint from all other types. We also introduce the concept of an "SRE," which is not a disjoint type but is a Scheme object following the specification described below.

SRFI 14 defines the `char-set` type, which can be used as part of an SRE.

In the prototypes below the following naming conventions imply type restrictions:

- *char-set*: a SRFI 14 character set
- *cset-sre*: an sre which corresponds to matching a single character out of a set of characters
- *end*: an exact, non-negative integer, defaulting to the `(string-length str)`
- *finish*: a procedure `(lambda (i regexp-match str obj) ...)`
- *obj*: any object
- *knil*: any object
- *kons*: a procedure `(lambda (i regexp-match str obj) ...)`
- *re*: an SRE or pre-compiled regexp object
- *regexp-match*: a regexp-match object from a successful match
- *sre*: an SRE as described below
- *start*: an exact, non-negative integer, defaulting to 0
- *str*: a string
- *subst*: an sexp describing a substitution template
- *X-or-false*: either an object of type X or the false value

Compatibility Levels and Features

We specify a thorough, though not exhaustive, syntax with many extensions popular in modern regular expression libraries such as PCRE. This is because it is assumed in many cases said libraries will be used as the underlying implementation, the features will be desirable, and if left unspecified people will provide their own, often incompatible, extensions.

On the other hand it is acknowledged that not all implementations will be able to support all extensions. Some are difficult to implement for DFA implementations, and some, like `backref`, are prohibitively expensive for any implementation. Furthermore, even if an implementation has Unicode support, its regexp library may not.

To resolve these differences we divide the syntax into a minimal core which all implementations are required to support, and additional extensions. In R7RS or other implementations which support SRFI 0 `cond-expand` with library level features, the availability can be tested with the following `cond-expand` features:

- `regexp-non-greedy` - the non-greedy repetition patterns `??`, `*?`, and `**?` are supported
- `regexp-look-around` - the `[neg-]look-ahead` and `[neg-]look-behind` zero-width assertions are supported
- `regexp-backrefs` - the `backref` pattern is supported
- `regexp-unicode` - regexp character sets support Unicode

The first three simply refer to support for certain SRE patterns.

`regexp-unicode` indicates support for Unicode contexts. Toggling between Unicode and ASCII can be done with the `w/unicode` and `w/ascii` patterns. In a Unicode context, the named character sets have their full Unicode definition as described below and grapheme boundaries are "extended grapheme clusters" as defined in UAX #29 (Unicode Text Segmentation). Implementations which provide this feature may still support non-Unicode characters.

Library Procedures and Syntax

(`regexp` *re*) => regexp

Compiles a regexp if given an object whose structure matches the SRE syntax. This may be written as a literal or partial literal with `quote` or `quasiquote`, or may be generated entirely programmatically. Returns *re* unmodified if it is already a regexp. Raises an error if *re* is neither a regexp nor a valid representation of an SRE.

Mutating *re* may invalidate the resulting regexp, causing unspecified results if subsequently used for matching.

(`rx` *sre ...*) => regexp

Macro shorthand for `(regexp `(: sre ...))`. May be able to perform some or all computation at compile time if *sre* is not unquoted. Note because of this equivalence with the procedural constructor `regexp`, the semantics of `unquote` differs from the original SCSH implementation in that unquoted expressions can expand into any object matching the SRE syntax, but not a compiled regexp object. Further, `unquote` and `unquote-splicing` both expand all matches.

> Rationale: Providing a procedural interface provides for greater flexibility, and without loss of potential compile-time optimizations by preserving the syntactic shorthand. The alternative is to rely on `eval` to dynamically generate regular expressions. However regexps in many cases come from untrusted sources, such as search parameters to a server, or from serialized sources such as config files or command-line arguments. Moreover many applications may want to keep many thousands of regexps in memory at once. Given the relatively heavy cost and insecurity of `eval`, and the frequency with which SREs are read and written as text, we prefer the procedural interface.

(`regexp->sre` *re*) => sre

Returns an SRE corresponding to the given regexp *re*. The SRE will be equivalent to (will match the same strings) but not necessarily `equal?` to the SRE originally used to compile *re*. Mutating the result may invalidate *re*, causing unspecified results if subsequently used for matching.

(`char-set->sre` *char-set*) => sre

Returns an SRE corresponding to the given SRFI 14 character set. The resulting SRE expands the character set into notation which does not make use of embedded SRFI 14 character sets, and so is suitable for writing portably.

(`valid-sre?` *obj*) => boolean

Returns true iff *obj* can be safely passed to *regexp*.

(`regexp?` *obj*) => boolean

Returns true iff *obj* is a regexp.

(`regexp-matches` *re str [start [end]]*) => regexp-match-or-false

Returns an regexp-match object if *re* successfully matches the entire string *str* from *start* (inclusive) to *end* (exclusive), or #f is the match fails. The regexp-match object will contain information needed to extract any submatches.

(`regexp-matches?` *re str [start [end]]*) => boolean?

Returns `#t` if *re* matches *str* as in regexp-matches, or `#f` otherwise. May be faster than `regexp-matches` since it doesn't need to return submatch data.

(`regexp-search` *re str [start [end]]*) => regexp-match-or-false

Returns a regexp-match object if *re* successfully matches a substring of *str* between *start* (inclusive) and *end* (exclusive), or `#f` if the match fails. The regexp-match object will contain information needed to extract any submatches.

(`regexp-fold` *re kons knil str [finish [start [end]]]*) => obj

The fundamental regexp matching iterator. Repeatedly searches *str* for the regexp *re* so long as a match can be found. On each successful match, applies `(kons i regexp-match str acc)` where *i* is the index since the last match (beginning with *start*), *regexp-match* is the resulting match, and *acc* is the result of the previous *kons* application, beginning with *knil*. When no more matches can be found, calls *finish* with the same arguments, except that *regexp-match* is `#f`.

By default *finish* just returns *acc*.

```
(regexp-fold 'word
             (lambda (i m str acc)
               (let ((s (regexp-match-submatch m 0)))
                (cond ((assoc s acc)
                       => (lambda (x) (set-cdr! x (+ 1 (cdr x))) acc))
                      (else `((,s . 1) ,@acc)))))
             '()
             "to be or not to be")
=> '(("not" . 1) ("or" . 1) ("be" . 2) ("to" . 2))
```

(`regexp-extract` *re str [start [end]]*) => list

Extracts all non-empty substrings of *str* which match *re* between *start* and *end* as a list of strings.

```
(regexp-extract '(+ numeric) "192.168.0.1")
=> ("192" "168" "0" "1")
```

(`regexp-split` *re str [start [end]]*) => list

Splits *str* into a list of (possibly empty) strings separated by non-empty matches of *re*.

```
(regexp-split '(+ space) " fee fi  fo\tfum\n")
=> ("" "fee" "fi" "fo" "fum" "")
(regexp-split '(",;") "a,,b,")
=> ("a" "" "b" "")
(regexp-split '(* digit) "abc123def456ghi789")
=> ("abc" "def" "ghi" "")
```

(`regexp-partition` *re str [start [end]]*) => list

Partitions *str* into a list of non-empty strings matching *re*, interspersed with the unmatched portions of the string. The first and every odd element is an unmatched substring, which will be the empty string if *re* matches at the beginning of the string or end of the previous match. The second and every even element will be a substring matching *re*. If the final match ends at the end of the string, no trailing empty string will be included. Thus, in the degenerate case where *str* is the empty string, the result is `("")`.

Note that `regexp-partition` is equivalent to interleaving the results of `regexp-split` and `regexp-extract`, starting with the former.

```
(regexp-partition '(+ (or space punct)) "")
=> ("")
(regexp-partition '(+ (or space punct)) "Hello, world!\n")
=> ("Hello" ", " "world" "!\n")
(regexp-partition '(+ (or space punct)) "¿Dónde Estás?")
=> ("" "¿" "Dónde" " " "Estás" "?")
(regexp-partition '(* digit) "abc123def456ghi789")
=> ("abc" "123" "def" "456" "ghi" "789")
```

(`regexp-replace` *re str subst* [*start* [*end* [*count*]]]) => string

Returns a new string replacing the *count*th match of *re* in *str* the *subst*, where the zero-indexed *count* defaults to zero (i.e. the first match). If there are not *count* matches, returns the selected substring unmodified.

subst can be a string, an integer or symbol indicating the contents of a numbered or named submatch of *re*, `'pre` for the substring to the left of the match, or `'post` for the substring to the right of the match.

The optional parameters *start* and *end* restrict both the matching and the substitution, to the given indices, such that the result is equivalent to omitting these parameters and replacing on `(substring str start end)`. As a convenience, a value of `#f` for *end* is equivalent to `(string-length str)`.

```
(regexp-replace '(+ space) "one two three" "_")
=> "one_two three"
(regexp-replace '(+ space) "one two three" "_" 0 #f 0)
=> "one_two three"
(regexp-replace '(+ space) "one two three" "_" 0 #f 1)
=> "one two_three"
(regexp-replace '(+ space) "one two three" "_" 0 #f 2)
=> "one two three"
```

(`regexp-replace-all` *re str subst [start [end]]*) => string

Equivalent to *regexp-replace*, but replaces all occurrences of *re* in *str*.

```
(regexp-replace-all '(+ space) "one two three" "_")
=> "one_two_three"
```

(`regexp-match?` *obj*) => boolean

Returns true iff *obj* is a successful match from `regexp-matches` or `regexp-search`.

```
(regexp-match? (regexp-matches "x" "x"))  => #t
(regexp-match? (regexp-matches "x" "y"))  => #f
```

(`regexp-match-count` *regexp-match*) => integer

Returns the number of submatches of regexp-match, regardless of whether they matched or not. Does not include the implicit zero full match in the count.

```
(regexp-match-count (regexp-matches "x" "x"))  => 0
(regexp-match-count (regexp-matches '($ "x") "x"))  => 1
```

(`regexp-match-submatch` *regexp-match field*) => string-or-false

Returns the substring matched in *regexp-match* corresponding to *field*, either an integer or a symbol for a named submatch. Index 0 refers to the entire match, index 1 to the first lexicographic submatch, and so on. If there are multiple submatches with the same name, the first which matched is returned. If passed an integer outside the range of matches, or a symbol which does not correspond to a named submatch of the pattern, it is an error. If the corresponding submatch did not match, returns false.

The result of extracting a submatch after the original matched string has been mutated is unspecified.

```
(regexp-match-submatch (regexp-search 'word "**foo**") 0)  => "foo"
(regexp-match-submatch
 (regexp-search '(: "*" ($ word) "*") "**foo**") 0)  => "*foo*"
(regexp-match-submatch
 (regexp-search '(: "*" ($ word) "*") "**foo**") 1)  => "foo"
```

(`regexp-match-submatch-start` *regexp-match field*) => integer-or-false

Returns the start index *regexp-match* corresponding to *field*, as in *regexp-match-submatch*.

```
(regexp-match-submatch-start
 (regexp-search 'word "**foo**") 0)  => 2
(regexp-match-submatch-start
 (regexp-search '(: "*" ($ word) "*") "**foo**") 0)  => 1
(regexp-match-submatch-start
 (regexp-search '(: "*" ($ word) "*") "**foo**") 1)  => 2
```

(`regexp-match-submatch-end` *regexp-match field*) => integer-or-false

Returns the end index in *regexp-match* corresponding to *field*, as in *regexp-match-submatch*.

```
(regexp-match-submatch-end
 (regexp-search 'word "**foo**") 0)  => 5
(regexp-match-submatch-end
 (regexp-search '(: "*" ($ word) "*") "**foo**") 0)  => 6
(regexp-match-submatch-end
 (regexp-search '(: "*" ($ word) "*") "**foo**") 1)  => 5
```

(`regexp-match->list` *regexp-match*) => list

Returns a list of all submatches in *regexp-match* as string or false, beginning with the entire match 0.

```
(regexp-match->list
 (regexp-search '(: ($ word) (+ (or space punct)) ($ word)) "cats & dogs"))
 => '("cats & dogs" "cats" "dogs")
```

SRE Syntax

The grammar for SREs is summarized below. Note that an SRE is a first-class object consisting of nested lists of strings, chars, char-sets, symbols and numbers. Where the syntax is described as `(foo bar)`, this can be constructed equivalently as `'(foo bar)` or `(list 'foo 'bar)`, etc. The following sections explain the semantics in greater detail.

```
<sre> ::=
 | <string>                    ; A literal string match.
 | <cset-sre>                  ; A character set match.
 | (* <sre> ...)               ; 0 or more matches.
 | (zero-or-more <sre> ...)
 | (+ <sre> ...)               ; 1 or more matches.
 | (one-or-more <sre> ...)
 | (? <sre> ...)               ; 0 or 1 matches.
 | (optional <sre> ...)
 | (= <n> <sre> ...)           ; <n> matches.
 | (exactly <n> <sre> ...)
 | (>= <n> <sre> ...)          ; <n> or more matches.
 | (at-least <n> <sre> ...)
 | (** <n> <m> <sre> ...)      ; <n> to <m> matches.
 | (repeated <n> <m> <sre> ...)

 | (|  <sre> ...)              ; Alternation.
 | (or <sre> ...)

 | (:   <sre> ...)             ; Sequence.
 | (seq <sre> ...)
 | ($ <sre> ...)               ; Numbered submatch.
 | (submatch <sre> ...)
 | (-> <name> <sre> ...)               ;  Named submatch.  <name> is
 | (submatch-named <name> <sre> ...)   ;  a symbol.
```

```
     | (w/case   <sre> ...)        ; Introduce a case-sensitive context.
     | (w/nocase <sre> ...)        ; Introduce a case-insensitive context.

     | (w/unicode   <sre> ...)     ; Introduce a unicode context.
     | (w/ascii <sre> ...)         ; Introduce an ascii context.

     | (w/nocapture <sre> ...)     ; Ignore all enclosed submatches.

     | bos                         ; Beginning of string.
     | eos                         ; End of string.

     | bol                         ; Beginning of line.
     | eol                         ; End of line.

     | bog                         ; Beginning of grapheme cluster.
     | eog                         ; End of grapheme cluster.
     | grapheme                    ; A single grapheme cluster.

     | bow                         ; Beginning of word.
     | eow                         ; End of word.
     | nwb                         ; A non-word boundary.
     | (word <sre> ...)            ; An SRE wrapped in word boundaries.
     | (word+ <cset-sre> ...)      ; A single word restricted to a cset.
     | word                        ; A single word.

     | (?? <sre> ...)              ; A non-greedy pattern, 0 or 1 match.
     | (non-greedy-optional <sre> ...)
     | (*? <sre> ...)              ; Non-greedy 0 or more matches.
     | (non-greedy-zero-or-more <sre> ...)
     | (**? <m> <n> <sre> ...)     ; Non-greedy <m> to <n> matches.
     | (non-greedy-repeated <sre> ...)
     | (look-ahead <sre> ...)      ; Zero-width look-ahead assertion.
     | (look-behind <sre> ...)     ; Zero-width look-behind assertion.
     | (neg-look-ahead <sre> ...)  ; Zero-width negative look-ahead assertion.
     | (neg-look-behind <sre> ...) ; Zero-width negative look-behind assertion.

     | (backref <n-or-name>)       ; Match a previous submatch.
```

The grammar for `cset-sre` is as follows.

```
    <cset-sre> ::=
     | <char>                      ; literal char
     | "<char>"                    ; string of one char
     | <char-set>                  ; embedded SRFI 14 char set
     | (<string>)                  ; literal char set
     | (char-set <string>)
     | (/ <range-spec> ...)        ; ranges
     | (char-range <range-spec> ...)
     | (or <cset-sre> ...)         ; union
     | (|\|| <cset-sre> ...)
     | (and <cset-sre> ...)        ; intersection
     | (& <cset-sre> ...)
     | (- <cset-sre> ...)          ; difference
     | (- <difference> ...)
     | (~ <cset-sre> ...)          ; complement of union
     | (complement <cset-sre> ...)
     | (w/case <cset-sre> ...)     ; case and unicode toggling
     | (w/nocase <cset-sre> ...)
     | (w/ascii <cset-sre> ...)
     | (w/unicode <cset-sre> ...)
     | any | nonl | ascii | lower-case | lower
     | upper-case | upper | title-case | title
     | alphabetic | alpha | alphanumeric | alphanum | alnum
     | numeric | num | punctuation | punct | symbol
```

```
      | graphic | graph | whitespace | white | space
      | printing | print | control | cntrl | hex-digit | xdigit

    <range-spec> ::= <string> | <char>
```

Basic Patterns

`<string>`

A literal string.

```
(regexp-search "needle" "hayneedlehay") => #<regexp-match>
(regexp-search "needle" "haynEEdlehay") => #f
```

(`seq` *sre ...*)
(`:` *sre ...*)

Sequencing. Matches if each of *sre* matches adjacently in order.

```
(regexp-search '(: "one" space "two" space "three") "one two three") => #<regexp-match>
```

(`or` *sre ...*)
(`|\||` *sre ...*)

Alternation. Matches if any of *sre* match.

```
(regexp-search '(or "eeney" "meeney" "miney") "meeney") => #<regexp-match>
(regexp-search '(or "eeney" "meeney" "miney") "moe") => #f
```

(`w/nocase` *sre ...*)

Enclosed *sres* are case-insensitive. In a Unicode context character and string literals match with the default simple Unicode case-insensitive matching. Implementations may, but are not required to, handle variable length case conversions, such as #\x00DF "ß" matching the two characters "SS".

Character sets match if any character in the set matches case-insensitively to the input. Conceptually each `cset-sre` is expanded to contain all case variants for all of its characters. In a compound `cset-sre` the expansion is applied at the terminals consisting of characters, strings, embedded SRFI 14 char-sets, and named character sets. For simple unions this would be equivalent to computing the full union first and then expanding case variants, but the semantics can differ when differences and intersections are applied. For example, `(w/nocase (~ ("Aab")))` is equivalent to `(~ ("AaBb"))`, for which "B" is clearly not a member. However if you were to compute `(~ ("Aab"))` first then you would have a char-set containing "B", and after expanding case variants both "B" and "b" would be members.

In an ASCII context only the 52 ASCII letters `(/ "a-zA-Z")` match case-insensitively to each other.

In a Unicode context the only named `cset-sre` which are affected by `w/nocase` are `upper` and `lower`. Note that the case insensitive versions of these are not equivalent to `alpha` as there are characters with the letter property but no case.

```
(regexp-search "needle" "haynEEdlehay") => #f
(regexp-search '(w/nocase "needle") "haynEEdlehay") => #<regexp-match>

(regexp-search '(~ ("Aab")) "B") => #<regexp-match>
(regexp-search '(~ ("Aab")) "b") => #f
```

```
(regexp-search '(w/nocase (~ ("Aab"))) "B") => #f
(regexp-search '(w/nocase (~ ("Aab"))) "b") => #f
(regexp-search '(~ (w/nocase ("Aab"))) "B") => #f
(regexp-search '(~ (w/nocase ("Aab"))) "b") => #f
```

(`w/case` *sre* ...)

Enclosed *sres* are case-sensitive. This is the default, and overrides any enclosing `w/nocase` setting.

```
(regexp-search '(w/nocase "SMALL" (w/case "BIG")) "smallBIGsmall") => #<regexp-match>
(regexp-search '(w/nocase (~ (w/case ("Aab")))) "b") => #f
```

(`w/ascii` *sre* ...)

Enclosed *sres* are interpreted in an ASCII context. In practice many regular expressions are used for simple parsing and only ASCII characters are relevant. Switching to ASCII mode can improve performance in some implementations.

```
(regexp-search '(w/ascii bos (* alpha) eos) "English") => #<regexp-match>
(regexp-search '(w/ascii bos (* alpha) eos) "Ελληνική") => #f
```

(`w/unicode` *sre* ...)

Enclosed *sres* are interpreted in a Unicode context - character sets with both an ASCII and Unicode definition take the latter. Has no effect if the `regexp-unicode` feature is not provided. This is the default.

```
(regexp-search '(w/unicode bos (* alpha) eos) "English") => #<regexp-match>
(regexp-search '(w/unicode bos (* alpha) eos) "Ελληνική") => #<regexp-match>
```

(`w/nocapture` *sre* ...)

Disables capturing for all submatches (`$`, `submatch`, `->` and `submatch-named`) in the enclosed *sres*. The resulting SRE matches exactly the same strings, but without any associated submatch info. Useful for utility SREs which you want to incorporate without affecting your submatch positions.

```
(let ((number '($ (+ digit))))
  (cdr
   (regexp-match->list
    (regexp-search `(: ,number "-" ,number "-" ,number)
                   "555-867-5309")))  ; => '("555" "867" "5309")
  (cdr
   (regexp-match->list
    (regexp-search `(: ,number "-" (w/nocapture ,number) "-" ,number)
                   "555-867-5309"))))   => '("555" "5309")
```

Repeating patterns

(`optional` *sre* ...)
(`?` *sre* ...)

An optional pattern - matches 1 or 0 times.

```
(regexp-search '(: "match" (? "es") "!") "matches!") => #<regexp-match>
(regexp-search '(: "match" (? "es") "!") "match!") => #<regexp-match>
(regexp-search '(: "match" (? "es") "!") "matche!") => #f
```

(`zero-or-more` *sre* ...)
(`*` *sre* ...)

Kleene star, matches 0 or more times.

```
(regexp-search '(: "<" (* (~ #\>)) ">") "<html>") => #<regexp-match>
(regexp-search '(: "<" (* (~ #\>)) ">") "<>") => #<regexp-match>
(regexp-search '(: "<" (* (~ #\>)) ">") "<html") => #f
```

(`one-or-more` *sre ...*)

(`+` *sre ...*)

1 or more matches. Like `*` but requires at least a single match.

```
(regexp-search '(: "<" (+ (~ #\>)) ">") "<html>") => #<regexp-match>
(regexp-search '(: "<" (+ (~ #\>)) ">") "<a>") => #<regexp-match>
(regexp-search '(: "<" (+ (~ #\>)) ">") "<>") => #f
```

(`at-least` *n sre ...*)

(`>=` *n sre ...*)

More generally, *n* or more matches.

```
(regexp-search '(: "<" (>= 3 (~ #\>)) ">") "<table>") => #<regexp-match>
(regexp-search '(: "<" (>= 3 (~ #\>)) ">") "<pre>") => #<regexp-match>
(regexp-search '(: "<" (>= 3 (~ #\>)) ">") "<tr>") => #f
```

(`exactly` *n sre ...*)

(`=` *n sre ...*)

Exactly *n* matches.

```
(regexp-search '(: "<" (= 4 (~ #\>)) ">") "<html>") => #<regexp-match>
(regexp-search '(: "<" (= 4 (~ #\>)) ">") "<table>") => #f
```

(`repeated` *from to sre ...*)

(`**` *from to sre ...*)

The most general form, from *n* to *m* matches, inclusive.

```
(regexp-search '(: (= 3 (** 1 3 numeric) ".") (** 1 3 numeric)) "192.168.1.10") => #<regexp-r
(regexp-search '(: (= 3 (** 1 3 numeric) ".") (** 1 3 numeric)) "192.0168.1.10") => #f
```

Submatch Patterns

(`submatch` *sre ...*)

(`$` *sre ...*)

A numbered submatch. The contents matching the pattern will be available in the resulting regexp-match.

(`submatch-named` *name sre ...*)

(`->` *name sre ...*)

A named submatch. Behaves just like *submatch*, but the field may also be referred to by *name*.

(`backref` *n-or-name*)

Optional: Match a previously matched submatch. The feature `regexp-backrefs` will be provided if this pattern is supported. Backreferences are expensive, and can trivially be shown to be NP-hard, so one should avoid their use even in implementations which support them.

Character Sets

A character set pattern matches a single character.

`<char>`

A singleton char set.

```
(regexp-matches '(* #\-) "---") => #<regexp-match>
(regexp-matches '(* #\-) "-_-") => #f
```

`"<char>"`

A singleton char set written as a string of length one rather than a character. Equivalent to its interpretation as a literal string match, but included to clarify it can be composed in `cset-sre`s.

`<char-set>`

A SRFI 14 character set, which matches any character in the set. Note that currently there is no portable written representation of SRFI 14 character sets, which means that this pattern is typically generated programmatically, such as with a quasiquoted expression.

```
(regexp-partition `(+ ,char-set:vowels) "vowels")
=> ("v" "o" "w" "e" "ls")
```

> Rationale: Many useful character sets are likely to be available as SRFI 14 `char-set`s, so it is desirable to reuse them in regular expressions. Since many Unicode character sets are extremely large, converting back and forth between an internal and external representation can be expensive, so the option of direct embedding is necessary. When a readable external representation is needed, `char-set->sre` can be used.

`(char-set <string>)`
`(<string>)`

The set of chars as formed by SRFI 14 `(string->char-set` *`<string>`*`)`.

Note that char-sets contain code points, not grapheme clusters, so any combining characters in *<string>* will be inserted separately from any preceding base characters by `string->char-set`.

```
(regexp-matches '(* ("aeiou")) "oui") => #<regexp-match>
(regexp-matches '(* ("aeiou")) "ouais") => #f
(regexp-matches '(* ("e\x0301")) "e\x0301") => #<regexp-match>
(regexp-matches '("e\x0301") "e\x0301") => #f
(regexp-matches '("e\x0301") "e") => #<regexp-match>
(regexp-matches '("e\x0301") "\x0301") => #<regexp-match>
(regexp-matches '("e\x0301") "\x00E9") => #f
```

(`char-range` *<range-spec> ...*)
(`/` *<range-spec> ...*)

Ranged char set. The *<range-spec>* is a list of strings and characters. These are flattened and grouped into pairs of characters, and all ranges formed by the pairs are included in the char set.

```
(regexp-matches '(* (/ "AZ09")) "R2D2") => #<regexp-match>
(regexp-matches '(* (/ "AZ09")) "C-3PO") => #f
```

(`or` *<cset-sre> ...*)
(`|\||` *<cset-sre> ...*)

Char set union. The single vertical bar form is provided for consistency and compatibility with SCSH, although it needs to be escaped in R7RS.

(`complement` *<cset-sre> ...*)
(`~` *<cset-sre> ...*)

Char set complement (i.e. [^...] in PCRE notation). `(~ x)` is equivalent to `(- any x)`, thus in an ASCII context the complement is always ASCII.

(`difference` *<cset-sre> ...*)
(`-` *<cset-sre> ...*)

Char set difference.

```
(regexp-matches '(* (- (/ "az") ("aeiou"))) "xyzzy") => #<regexp-match>
(regexp-matches '(* (- (/ "az") ("aeiou"))) "vowels") => #f
```

(`and` *<cset-sre> ...*)
(`&` *<cset-sre> ...*)

Char set intersection.

```
(regexp-matches '(* (& (/ "az") (~ ("aeiou")))) "xyzzy") => #<regexp-match>
(regexp-matches '(* (& (/ "az") (~ ("aeiou")))) "vowels") => #f
```

Named Character Sets

`any`

Match any character. Equivalent to `ascii` in an ASCII context.

`nonl`

Match any character other than `#\return` or `#\newline`.

`ascii`

Match any ASCII character [0..127].

`lower-case`
`lower`

Matches any character for which `char-lower-case?` returns true. In a Unicode context this corresponds to the Lowercase (Ll + Other_Lowercase) property. In an ASCII context corresponds to `(/ "az")`.

`upper-case`
`upper`

Matches any character for which `char-upper-case?` returns true. In a Unicode context this corresponds to the Uppercase (Lu + Other_Uppercase) property. In an ASCII context corresponds to `(/ "AZ")`.

`title-case`
`title`

Matches any character with the Unicode Titlecase (Lt) property. This property only exists for the sake of ligature characters, of which only 31 exist at time of writing. In an ASCII context this is empty.

`alphabetic`
`alpha`

Matches any character for which `char-alphabetic?` returns true. In a Unicode context this corresponds to the Alphabetic (L + Nl + Other_Alphabetic) property. In an ASCII context corresponds to `(w/nocase (/ "az"))`.

`numeric`
`num`

Matches any character for which `char-numeric?` returns true. For In a Unicode context this corresponds to the Numeric_Digit (Nd) property. In an ASCII context corresponds to `(/ "09")`.

`alphanumeric`
`alphanum`
`alnum`

Matches any character which is either a letter or number. Equivalent to:

```
(or alphabetic numeric)
```

`punctuation`
`punct`

Matches any punctuation character. In a Unicode context this corresponds to the Punctuation property (P). In an ASCII context this corresponds to `"!\"#%&'()*,-./:;?@[\]_{}"`.

`symbol`

Matches any symbol character. In a Unicode context this corresponds to the Symbol property (Sm, Sc, Sk, or So). In an ASCII this corresponds to `"$+<=>^`|~"`.

`graphic`
`graph`

Matches any graphic character. Equivalent to:

```
(or alphanumeric punctuation symbol)
```

`whitespace`
`white`
`space`

Matches any whitespace character. In a Unicode context this corresponds to the Separator property (Zs, Zl or Zp). In an ASCII context this corresponds to space, tab, line feed, form feed, and carriage return.

`printing`
`print`

Matches any printing character. Equivalent to:

```
(or graphic whitespace)
```

`control`
`cntrl`

Matches any control or other character. In a Unicode context this corresponds to the Other property (Cc, Cf, Co, Cs or Cn). In an ASCII context this corresponds to:

```
`(/ ,(integer->char 0) ,(integer-char 31))
```

`hex-digit`
`xdigit`

Matches any valid digit in hexadecimal notation. Always ASCII-only. Equivalent to:

```
(w/ascii (w/nocase (or numeric "abcdef")))
```

Boundary Assertions

`bos`

`eos`

Matches at the beginning/end of string without consuming any characters (a zero-width assertion). If the search was initiated with start/end parameters, these are considered the end points, rather than the full string.

`bol`

`eol`

Matches at the beginning/end of a line without consuming any characters (a zero-width assertion). A line is a possibly empty sequence of characters followed by an end of line sequence as understood by the R7RS `read-line` procedure, specifically any of a linefeed character, carriage return character, or a carriage return followed by a linefeed character. The string is assumed to contain end of line sequences before the start and after the end of the string, even if the search was made on a substring and the actual surrounding characters differ.

`bow`

`eow`

Matches at the beginning/end of a word without consuming any characters (a zero-width assertion). A word is a contiguous sequence of characters that are either alphanumeric or the underscore character, i.e. `(or alphanumeric "_")`, with the definition of `alphanumeric` depending on the Unicode or ASCII context. The string is assumed to contain non-word characters immediately before the start and after the end, even if the search was made on a substring and word constituent characters appear immediately before the beginning or after the end.

```
(regexp-search '(: bow "foo") "foo") => #<regexp-match>
(regexp-search '(: bow "foo") "<foo>>") => #<regexp-match>
(regexp-search '(: bow "foo") "snafoo") => #f
(regexp-search '(: "foo" eow) "foo") => #<regexp-match>
(regexp-search '(: "foo" eow) "foo!") => #<regexp-match>
(regexp-search '(: "foo" eow) "foobar") => #f
```

`nwb`

Matches a non-word-boundary (i.e. \B in PCRE). Equivalent to `(neg-look-ahead (or bow eow))`.

(`word` *sre ...*)

Anchors a sequence to word boundaries. Equivalent to:

```
(: bow sre ... eow)
```

(`word+` *cset-sre ...*)

Matches a single word composed of characters in the intersection of the given *cset-sre* and the word constituent characters. Equivalent to:

```
(word (+ (and (or alphanumeric "_") (or cset-sre ...))))
```

`word`

A shorthand for `(word+ any)`.

`bog`
`eog`

Matches at the beginning/end of a single extended grapheme cluster without consuming any characters (a zero-width assertion). Grapheme cluster boundaries are defined in Unicode TR29. The string is assumed to contain non-combining codepoints immediately before the start and after the end. These always succeed in an ASCII context.

`grapheme`

Matches a single grapheme cluster (i.e. \X in PCRE). This is what the end-user typically thinks of as a single character, comprised of a base non-combining codepoint followed by zero or more combining marks. In an ASCII context this is equivalent to `any`.

Assuming `char-set:mark` contains all characters with the Extend or SpacingMark properties defined in TR29, and `char-set:control`, `char-set:regional-indicator` and `char-set:hangul-*` are defined similarly, then the following SRE can be used with `regexp-extract` to define grapheme:

```
`(or (: (* ,char-set:hangul-l) (+ ,char-set:hangul-v)
        (* ,char-set:hangul-t))
     (: (* ,char-set:hangul-l) ,char-set:hangul-v
        (* ,char-set:hangul-v) (* ,char-set:hangul-t))
     (: (* ,char-set:hangul-l) ,char-set:hangul-lvt
        (* ,char-set:hangul-t))
     (+ ,char-set:hangul-l)
     (+ ,char-set:hangul-t)
     (= 2 ,char-set:regional-indicator)
     (: "\r\n")
     (: (~ control ("\r\n"))
        (* ,char-set:mark))
     control)
```

Non-Greedy Patterns

The following patterns are only supported if the feature `regexp-non-greedy` is provided.

(`non-greedy-optional` *sre ...*)
(`??` *sre ...*)

Non-greedy pattern, matches 0 or 1 times, preferring the shorter match.

(`non-greedy-zero-or-more`< *sre ...*)
(`*?` *sre ...*)

Non-greedy Kleene star, matches 0 or more times, preferring the shorter match.

(`non-greedy-repeated` *m n sre ...*)
(`**?` *m n sre ...*)

Non-greedy Kleene star, matches *m* to *n* times, preferring the shorter match.

Look Around Patterns

The following patterns are only supported if the feature `regexp-look-around` is provided.

(`look-ahead` *sre ...*)

Zero-width look-ahead assertion. Asserts the sequence matches from the current position, without advancing the position.

```
(regexp-matches '(: "regular" (look-ahead " expression") " expression") "regular expression")
(regexp-matches '(: "regular" (look-ahead " ") "expression") "regular expression")=> #f
```

(`look-behind` *sre ...*)

Zero-width look-behind assertion. Asserts the sequence matches behind the current position, without advancing the position. It is an error if the sequence does not have a fixed length.

(`neg-look-ahead` *sre ...*)

Zero-width negative look-ahead assertion.

(`neg-look-behind` *sre ...*)

Zero-width negative look-behind assertion.

Implementation

A sample implementation in portable R7RS is available at

https://github.com/ashinn/chibi-scheme/blob/master/lib/srfi/115.sld
https://github.com/ashinn/chibi-scheme/blob/master/lib/chibi/regexp.sld
https://github.com/ashinn/chibi-scheme/blob/master/lib/chibi/regexp.scm
https://github.com/ashinn/chibi-scheme/blob/master/lib/chibi/char-set/boundary.sld
https://github.com/ashinn/chibi-scheme/blob/master/lib/chibi/char-set/boundary.scm

depending only on SRFI 14, SRFI 33 and SRFI 69. This is implemented as a Thompson-style non-backtracking NFA, a discussion of which can be found at Russ Cox's Implementing Regexps. At time of writing the implementation supports full unicode, but does not yet support look-around patterns. See IrRegex for an alternate implementation supporting all features, providing both a backtracking algorithm and DFA compilation.

References

R7RS

Alex Shinn, John Cowan, Arthur Gleckler, Revised^7 Report on the Algorithmic Language S
http://trac.sacrideo.us/wg/raw-attachment/wiki/WikiStart/r7rs.pdf

SCSH

Olin Shivers, A Scheme Shell
Massachusetts Institute of Technology Cambridge, MA, USA, 1994
http://www.scsh.net/docu/scsh-paper/scsh-paper.html

IrRegex

Alex Shinn, IrRegex - IrRegular Expressions
http://synthcode.com/scheme/irregex/

TR18

`Mark Davis, Andy Heninger, UTR #18: Unicode Regular Expressions`
`http://www.unicode.org/reports/tr18/`

UAX29

`Mark Davis, UAX #29: Unicode Text Segmentation`
`http://www.unicode.org/reports/tr29/`

SRFI 0

`Marc Feeley, Feature-based conditional expansion construct`
`http://srfi.schemers.org/srfi-0/srfi-0.html`

SRFI 14

`Olin Shivers, Character-set Library`
`http://srfi.schemers.org/srfi-14/srfi-14.html`

ImplementingRegexps

`Russ Cox, Implementing Regular Expressions`
`http://swtch.com/~rsc/regexp/`

Tcl

`Russ Cox, Henry Spencer's Tcl Regex Library`
`http://compilers.iecc.com/comparch/article/07-10-026`

Gauche

`Shiro Kawai, Gauche Scheme - Regular Expressions`
`http://practical-scheme.net/gauche/man/?p=Regular+expressions`

Perl6

`Damian Conway, Perl6 Exegesis 5 - Regular Expressions`
`http://www.perl.com/pub/a/2002/08/22/exegesis5.html`

PCRE

`Philip Hazel, PCRE - Perl Compatible Regular Expressions`
`http://www.pcre.org/`

Copyright

Editor: Mike Sperber