Reference Manual for the

Elk Regular Expression Extension

Oliver Laumann

1. Introduction

The regular expression extension defines Scheme language bindings for the POSIX regular expression functions that are provided by most modern UNIX versions (*regcomp()* and *regexec()*). You may want to refer to your UNIX system's *regcomp(3)* manual for details. The Scheme interface to the regular expression functions makes the entire functionality of the usual C language interface available to the Scheme programmer. To load the regular expression extension, evaluate the expression

```
(require 'regexp)
```

This causes the files *regexp.scm* and *regexp.o* to be loaded (*regexp.o* must be statically linked with the interpreter on platforms that do not support dynamic loading of object files).

Loading the extension provides the features *regexp* and *regexp.o*. On systems that do not support the regular expression library functions, loading the extension succeeds, but no further primitives or features are defined. Otherwise, the additional feature *:regular-expressions* is provided, so that the expression

```
(feature? ':regular-expressions)
```

can be used in Scheme programs to check whether regular expressions are available on the local platform.

2. Creating Regular Expressions

(make-regexp *pattern***)** **procedure**

(make-regexp *pattern flags***)** **procedure**

make-regexp returns an object of the new Scheme type *regexp* representing the regular expression specified by the string argument *pattern*. An error is signaled if the underlying call to the C library function *regcomp(3)* fails. The optional *flags* argument is a list of zero or more of the symbols *extended, ignore-case, no-subexpr,* and *newline*; these correspond to the C constants *REG_EXTENDED, REG_ICASE, REG_NOSUB,* and *REG_NEWLINE.*

Two objects of the type *regexp* are equal in the sense of *equal?* if their flags are identical and if their patterns are equal in the sense of *string=?*. Two regular expressions are *eq?* if their flags are identical and if they share the same pattern string.

(regexp? *obj***)** **procedure**

This type predicate returns #t if *obj* is a regular expression, #f otherwise.

(regexp-pattern *regexp***)** **procedure**

(regexp-flags *regexp***)** **procedure**

These primitives return the pattern (or flags, respectively) specified in the call to *make-regexp* that has created the regular expression object.

3. Matching Regular Expressions

(regexp-exec *regexp string offset***)** **procedure**

(regexp-exec *regexp string offset flags***)** **procedure**

This primitive applies the specified regular expression to the given string starting at the given offset. *offset* is an integer larger than or equal to zero and less than or equal to the length of *string*. If the match succeeds, *regexp-exec* returns an object of the new Scheme type *regexp-match*, otherwise #f. The optional *flags* argument is a list of zero or more of the symbols *not-bol* and *not-eol* which correspond to the constants *REG_NOTBOL* and *NOT_EOL* in the C language interface.

(regexp-match? *obj***)** **procedure**

This type predicate returns #t if *obj* is a regular expression match (that is, the return value of a successful call to *regexp-match*), #f otherwise.

(regexp-match-number *match***)** **procedure**

This primitive returns the number of substrings that matched parenthetic subexpressions in the original pattern when the given match was created, plus one (the first substring corresponds to the entire regular expression rather than a subexpression; see *regexec(3)* for details). A value of zero is returned if the match has been created by applying a regular expression with the *no-subexpr* flag set.

(regexp-match-start *match number***)** **procedure**

(regexp-match-end *match number***)** **procedure**

These primitives return the start offset (or end offset, respectively) of the substring denoted by the integer *number*. A *number* argument of zero refers to the substring corresponding to the entire pattern. The offsets returned by these primitives can be directly used as arguments to the *substring* primitive of Elk.

4. Example

The following program demonstrates a simple Scheme procedure *matches* that returns a list of substrings of a given string that match a given pattern. An error message is displayed if regular expressions are not supported by the local platform.

```
(require 'regexp)

(define (matches str pat)
  (let loop ((r (make-regexp pat '(extended))) (result '()) (from 0))
       (let ((m (regexp-exec r str from)))
         (if (regexp-match? m)
             (loop r (cons (substring str (+ from (regexp-match-start m 0))
                                          (+ from (regexp-match-end m 0)))
                           result)
                   (+ from (regexp-match-end m 0)))
             (reverse result)))))
```

Index

Table of Contents