The String Library Version 7

W. O. Riha 15-06-13

(revised Mark Tarver, 15-12-13)

Section 1 Introduction

1.1 Basic Definitions

In Shen, a **unit string** is represented by double quotes flanking a single character; e. g. "A", "$", "2" etc. Shen supports the full keyboard set for unit strings and, under current platforms, the ASCII set whose codes are found in the range 0-127. The notation "c#N;", where N is a decimal integer, is used to access the non-keyboard characters e.g. "c#27;" is read as "←". It is thus possible to access the full Unicode set if the platform permits it, but this is not guaranteed by the specification.

Unit strings are the building blocks of strings – they may therefore be viewed as a generalisation of characters, used in other languages. The set of characters in such languages constitutes a type, often called char, equipped with specific operations and/or functions, e.g. char-upcase, char-lowercase? etc.

A **string** is the result of n ($n \geq 0$) concatenations of unit strings to the empty string "" using the two-place primitive cn, e.g. (cn "h" (cn "e" (cn "l" (cn "l" "o")))) = "hello". The polyadic @s can also be used (@s "h" "e" "l" "l" "o") = "hello". See section 3.1.

Let S be a string consisting of L ≥ 0 unit strings, S[0], S[1], …, S[L-1]. To emphasise this fact, we use the notation S[0..L-1]. If L = 0 then S is the **null string**, denoted "".

The **length** of S is L.

If M ≤ N, then the **substring S[M..N]** of S is the string composed of the unit strings S[M], …, S[N] otherwise the substring is the null-string "".

S1 is a **prefix of S** iff S1 is "" or equal to S[0..M], for some M.

S1 is a **suffix of S** iff S1 is "" or equal to S[M..L-1], for some M.

1.2 Remarks and Conventions

a. The choice of functions (and their names) was inspired by the Scheme SRFI-13 string libraries. Identifiers usually include the prefix string. except when this is redundant and/or clumsy. For example, string.map but not string.substring or ustring.whitespace?

b. The notation employed in the code makes a distinction between unit strings and proper strings: unit string variables are denoted by S, S1, S2, …, whereas string variables are named Str, Str1, Str2,…, for example (@s S Str). This notation is not used in the present document, where the meaning of each parameter is explained in detail.

c. An attempt was made to be consistent when passing arguments to a function: the string being operated upon always comes last. For example, string.take : number → string → string, which returns a prefix of specified length e.g. (string.take 4 "ABCDEF"), or string.replace-all : string → string → string → string, where the third argument is the target string (see description).

d. All functions are tail-recursive.

e. Error messages are kept to a minimum. The functions are robust and will not cause a system crash, when supplied with illegal or silly arguments.

1.3 The Primitive String Functions

The following seven primitive string functions are defined (see Shendoc, *The Primitive Functions of Kλ*):

string? : A → boolean
Input: Any object *x*.
Output: true if *x* is a string, otherwise false.

```
(string? Mark)
false : boolean

(string? "Mark")
true : boolean
```

n->string : number → string
Input: A number (integer) *n*.
Output: If *n* is recognised as a code point, the corresponding unit string, otherwise an error message.

```
(n->string 65)
"A" : string

(n->string 120)
"x" : string
```

Only the code-points c#*n*; with $0 \leq n < 128$ and $160 \leq N < 256$ are recognised on the Lisp platform.

```
(n->string 165)
"¥" : string                    \* the yen sign ! *\
```

Shen is only required to cater for codes < 128 (the ASCII codes).

A (kind of) 'inverse' of n->string, is the function

string->n : string → number
Input: A string S.
Output: The code of the leading unit string of S.

```
(string->n "A")
65 : number

(string->n "shen")
115 : number
```

cn : string → string → string
Input: Two strings S1 and S2.
Output: The concatenation of S1 and S2.

```
(cn "Mark " "Tarver")
"Mark Tarver" : string
```

Note: (@s S1 S2) is equivalent to (cn S1 S2). @s, however, is more general, as it allows multiple concatenation. (See **Strings and Pattern Matching** in the Shen Document)

```
(@s "Dr. " " Mark" " " "Tarver")
"Dr. Mark Tarver" : string
```

tlstr : string → string

Input: A string S.

Output: S without its first unit string.

```
(tlstr "Mark")
ark : string
```

pos : string → number → string

Input: A string S and a (natural) number N.

Output: The *n*th unit string in S, if *n* is an integer satisfying $0 \leq n <$ (strlen S), otherwise an error message.

```
(pos "12345" 2)
"3" : string                    \* indexing starts at 0 ! *\
```

str : A → string

Input: Any atom *x*.

Output: The *x* embedded in a string.

```
(str "Mark")  \* a string is converted to a string *\
""Mark"" : string
```

1.4 Other String Functions

explode : A → (list string)

Input: Anything.

Output: The exploded input as a list of unit strings.

```
(explode "howdy")
["h" "o" "w" "d" "y"] : (list string)
```

hdstr : string → string

Input: A string S.

Output: The first unit string of S – same as (string.ref 0 S), or an error message if S is "".

A very useful string constructor is

make-string (**no type**)

For **Input/Output**, see **String, Bytes and Unicode** in the Shen Document.

```
(define hrs-mins
  {number → number → string}
  H M -> (make-string "~A hrs ~A mins" H M))

 (hrs-mins 12 45)
"12 hrs 45 mins" : string
```

Section 2 The Library Functions

2.1 String Predicates

digit? : string → boolean
Input: A string S.
Output: true if S is a digit, otherwise false.

```
(digit? "1")
true : boolean
```

uppercase? : string → boolean
Input: A string S.
Output: true if S is an upper-case letters "A","B", ...,"Z", otherwise false.

```
(uppercase? "4A")
false : boolean
```

lowercase? : string → boolean
Input: A string S.
Output: true if S if S is a lowercase letter "a","b",... "z", otherwise false.

```
(lowercase? "qq")
true : boolean
```

letter? : string → boolean
Input: A unit string S.
Output: true if S is an upper or lowercase letters, otherwise false.

```
(letter? "qQ")
true : boolean

(letter? "q2")
false : boolean
```

string.every? : (string → boolean) → string → boolean
Input: A predicate P of unit strings, and a string S.
Output: true if P is true for all unit strings of S, otherwise false. (checks if a string consists entirely of letters and spaces)

```
(string.every? (/. S (or (letter? S) (= S " "))) "String Library")
true : boolean
```

string.some? : (string → boolean) → string → boolean
Input: A predicate P of unit strings, and a string S.
Output: true if P is true for at least one unit string of S, otherwise false. (checks if a string contains a digit?)

```
(string.some? (function digits?) "String Library")
false : boolean
```

Using predicate string.every? it is easy to define functions to test if a given string is alphanumeric etc.

string.prefix? : string → string → boolean
Input: Two strings S1 and S2.
Output: true if S1 is a prefix of S2, otherwise false.

```
(string.prefix? "cat" "catapult")
true : boolean
```

string.suffix? : string → string → boolean
Input: Two strings S1 and S2.
Output: true if S1 is a suffix of S2, otherwise false.

```
(string.suffix? "ton" "Newton")
true : boolean
```

substring? : string → string → boolean
Input: Two strings S1 and S2.
Output: true if S1 is a substring of S2, otherwise false.

```
(substring? "tap" "catapult")
true : boolean
```

ustring? : string → boolean

Input: A string S.
Output: true if S is a unit string otherwise false.

```
(ustring? "12")
false : boolean

(ustring? "A")
true : boolean
```

whitespace? : string → boolean
Input: A string S.
Output: true if every unit string in S is in ["c#9;" "c#10;" "c#11;" "c#12;" "c#13;" " "], otherwise false.

```
(whitespace? " ")
true : boolean
```

2.2 String to String Functions

string.map : (string → string) → string → string

Input: A function F : string → string, and a string S.

Output: The string obtained from S by applying F to its constituent unit strings (for string.n-copy, see below).

```
(string.map (/. S (string.n-copy 3 S)) "1234")
"111222333444" : string
```

Using string.map one may define the following functions

string.upcase : string → string

Input: A string S.

Output: The string obtained from S by converting its lower-case letters to upper case.

```
(string.upcase "Shen 3.1")
"SHEN 3.1" : string
```

string.downcase : string → string

Input: A string S.

Output: The string obtained from S by converting all its upper-case letters to lower case.

```
(string.downcase "Shen 3.1")
"shen 3.1" : string
```

2.3 String Comparison

This is the lexicographic extension of the unit string comparisons to proper strings (only the ASCII codes are considered). The comparison functions for strings all have the signature

string → string → boolean

The following functions are available:

a. = equal (equality is defined for all native types, including strings).
b. != not equal (!= is defined for all types as shorthand for (not (= X Y))
c. <str less than
d. >str greater than
e. <=str less than or equal
f. >=str greater than or equal

```
(<=str "zebra" "zebu")
true : boolean
```

but

```
(<=str "zebra" "Zebu")
false : boolean     \* upper-case letters precede the lower-case ones *\
```

Note: Some libraries provide 'case independent' comparisons, in our notation <str-ci, =str-ci etc. These have not been included, because, as far as ASCII is concerned, they can be expressed by converting both arguments to the same case. For example, (if (<str (string.upcase S1) (string.upcase S2)) …)

2.4 Length Functions

string.length : string → number
Input: A string S.
Output: The length of S, i.e. the number of unit strings in S.
Note: Considering that many programmers are used to strlen, this identifier can be used as an alternative.

```
(string.length "ABCDE")
5 : number
```

string.prefix-length : string → string → number
Input: Two strings S1 and S2.
Output: The length of the longest common prefix of S1 and S2.

```
(string.prefix-length "Mark Tarver" "Mark Anthony")
5 : number

(string.prefix-length "Mark Tarver" "Willi")
0 : number
```

string.suffix-length : string → string → number
Input: Two strings S1 and S2.
Output: The length of the longest common suffix of S1 and S2.

```
(string.suffix-length "preclude" "interlude")
4 : number
```

2.5 Selection

string.take : number → string → string
Input: An integer N and a string S.
Output: The prefix of length N of S, i.e. the substring S[0..N-1].
Note: if N is greater than (string.length S), S is returned
if N is less than 1, the null string is returned.

```
(string.take 3 "ABCDEFG")
"ABC" : string

(string.take -1 "ABCDEFG")
"" : string
```

string.drop : number → string → string
Input: An integer N and a string S.
Output: S without its prefix of length N of S, i.e. the substring S[N..(strlen S)].
Note: if N is greater than (string.length S), "" is returned
if N is less than 1, then S is returned.

```
(string.drop 3 "ABCDEFG")
"DEFG" : string
```

```
(string.drop -1 "ABCD")
"ABCD" : string
```

In certain situations, both the ‘take’ and the corresponding ‘drop’ of a string are required. It would be wasteful to compute them separately as one can get the ‘drop’ for free, when working out the ‘take’. The following function takes this fact into account.

string.split : number → string → (string * string)

Input: An integer N and a string S.

Output: The pair consisting of the prefix of length N of S and the remaining suffix.

```
(string.split 5 "ABCDEFGH")
(@p "ABCDE" "FGH") : (string * string)
```

string.take-right : number → string → string

Input: An integer N and a string S.

Output: The suffix of length N of S.

Note: if N is greater than (string.length S), S is returned
if N is less than 1, the null string is returned.

```
(string.take-right 3 "ABCDEFG")
"EFG" : string

(string.take-right 13 "ABCDEFG")
"ABCDEFG" : string
```

string.drop-right : number → string → string

Input: An integer N and a string S.

Output: S without its suffix of length N.

Note: if N is greater than (string.length S), "" is returned
if N is less than 1, S is returned.

```
(string.drop-right 3 "ABCDEFG")
"ABCD" : string

(string.drop-right 10 "ABCDEFG")
"" : string
```

substring : number → number → string → string

Input: Two integers M and N and a string S.

Output: if M > N: the null string ""
if M ≤ N: the substring S[m..n], with m = (max 0 M) and n = (min N, (- (strlen S) 1).

Note: neither m nor n is actually computed!

```
(substring 1 3 "ABCDEFG")
"BCD" : string

(substring 3 1 "ABCDEFG")
"" : string

(substring 3 10 "ABCDEFG")
"DEFG" : string
```

2.5.1 Note on Non-Integer Arguments

In the functions above, if non-integer values for N (and/or M) are supplied as input, the output returned is as if ⌈N⌉ (and ⌈M⌉) had been supplied, i.e. non-integer values are rounded up (see **Maths Library**, section 3.3.3). No actual rounding takes place–the output is a product of the way functions are coded.

```
(string.take 4.76 "ABCDEFG") \* takes 5 *\
"ABCDE" : string

(string.drop 3.004 "ABCDEFG") \* drops 4 *\
"EFG" : string

(substring 1.2 3.02 "ABCDEFG") \* substring 2 4 ... *\
"CDE" : string
```

string.trim-left : (string → boolean) → (string → string)

Input: A predicate P of unit strings and a string S.

Output: The string obtained by dropping the longest prefix of S whose unit strings all satisfy P.

```
(string.trim-left (/. S (element? S [" " "0"])) "0  0  12003400 0")
"12003400 0" : string
```

string.trim-right : (string → boolean) → (string → string)

Input: A predicate P of unit strings and a string S.

Output: The string obtained by dropping the longest suffix of S whose unit strings all satisfy P..

```
(string.trim-right (/. S (element? S [" " "0"])) "0  0  12003400 0")
"0  0  120034" : string
```

string.trim : (string → boolean) → (string → string)

Input: A predicate P of unit strings and a string S.

Output: The string obtained from S by trimming it at both ends.

```
(string.trim (/. S (element? S [" " "0"])) "0  0  12003400 0")
120034 : string
```

Note: Choosing for P the predicate whitespace? will strip off all leading and/or trailing white space.

```
(string.trim whitespace? "   123  c#13;")
"123" : string
```

string.pad : string → number → string → string

Input: A unit string S1, a non-negative integer N and a string S2.

Output: The string of length N obtained from S2 by padding it to length N with copies of S1. If N < (string.length S2), the suffix of length N of S2 is returned, if N ≤ 0 the null string "".

```
(string.pad " " 10 "123456")   \* pad with spaces *\
"    123456" : string

(string.pad " " 10.33 "123456")   \* non-integer values are rounded down *\
"    123456" : string

(string.pad " " 4 "123456")
"3456" : string
```

2.6 Searching

The following function may be regarded as an ‘inverse’ of string.ref

string.index : string → string → number
Input: Two strings S1 and S2.
Output: If S1 is a substring of S2, the starting position of the first occurrence of S1 in S2, otherwise -1.

```
(string.index "is" "Mississippi")
1 : number

(string.index "eros" "heroine")
-1 : number
```

string.index-last : string → string → number
Input: Two strings S1 and S2.
Output: If S1 is a substring of S2, the starting position of the last occurrence of S1 in S2, otherwise -1.

```
(string.index-last "is" "Mississippi")
4 : number
```

string.count : string → string → number
Input: Two strings S1 and S2.
Output: The number of times S1 occurs as a substring in S (ignoring “overlapping” occurrences).

```
(string.count "11" "231145111")
2 : number                    \* "11" occurs only once in "111" *\
```

string.replace-all : string → string → string → string
Input: Three strings S1, S2, S3.
Output: The string obtained from S3 by replacing all occurrences of S2 with S1.

```
(string.replace-all "-" "/" "16/07/12")
"16-07-12" : string

(string.replace-all "XX" "000" "12000000340000789000")
"12XXXX34XX0789XX" : string
```

string.replace : string → number → string → string → string
Input: An integer N, and three strings S1, S2, S3.
Output: The string obtained from S3 by replacing the N-th occurrence of S2 with S1.

```
(string.replace "=" "+" 2 "100 + 3 + 103")
"100 + 3 = 103" : string
```

string.delete-all : string → string → string

Input: Two strings S1, S2.

Output: The string obtained from S2 by deleting all occurrences of S1.

```
(string.delete-all "00" "12000340005600")
"12034056" : string
```

delete-substring : number → number → string → string

Input: Two integers M and N and a string S.

Output: S if M > N,
if M ≤ N the string obtained from S by deleting substring S[m..n],
where m = (max 0 M), n = (min N (- (strlen S) 1)).
Note: neither m nor n are actually evaluated!

```
(delete-substring 2 4 "01234567")
"01567" : string

(delete-substring 2.2 4.8 "01234567")          \* non-integer values are rounded up *\
"01267" : string

(delete-substring -2 4 "01234567")
"567" : string

(delete-substring 2 40 "01234567")
"01" : string

(delete-substring 4 1 "01234567")
"01234567" : string
```

string.insert : number → string → string → string

Input: An integer N and two strings S1 and S2.

Output: The string obtained by inserting S1 into S2 after the N-prefix of S2.

Note: If N ≤ 0 then S1 is prepended, if N is greater than the length of S2, S1 is appended.

```
(string.insert 4 "ty" "nine days")
"ninety days" : string

(string.insert -4 "ty" "nine days")
"tynine days" : string
```

string.tokenise : (string → boolean) → string → (list string)

Input: A string S and a function F : string → boolean (defining the separators).

Output: The list of tokens.

(tokenise a date-and-time string "04-05-2012 20h 15m 32.5s" with separators "-" and " ").

```
(string.tokenise (/. S (element? S ["-" " "])) "04-05-2012 20h 15m 32.5s")
["04" "05" "2012" "20h" "15m" "32.5s"] : (list string)
```

A (kind of) 'inverse' of tokenise is the following function which produces a string from a list of strings by inserting a string between every two strings in the list.

string.join : string → (list string) → string

Input: A string S (to be inserted) and a list of strings StrL (the tokens).

Output: The string obtained by inserting S between every two strings in StrL.

```
(string.join " - " ["one" "two" "three"])
"one - two - three" : string

(string.join "  " (string.tokenise (/. S (element? S ["-" " "])) "04-05-2012 20h 15m 32.5s"))
"04  05  2012  20h  15m" : string
```

A related function is 'interpose' which inserts a string between every two unit strings of a string.

string.interpose : string → string → string

Input: Two strings S1 and S2.

Output: The string obtained by inserting S1 between every two unit strings of S2.

```
(string.interpose " + " "123456")
"1 + 2 + 3 + 4 + 5 + 6" : string
```

2.7 List and String Conversion

string->list : string → (list string)

Input: A string S.

Output: The list of unit strings of S.

Note: string->list is defined in terms of the system function explode : A → (list string) (which explodes any object into a list of unit strings).

```
(string->list "ABCD")
["A" "B" "C" "D"] : (list string)
```

list->string : (list string) → string

Input: A list StrL of strings.

Note: The elements of StrL can be general strings!

Output: The string formed from the strings in StrL.

```
(list->string (string->list "ABCD")) \* list->string is the left-inverse of string->list *\
"ABCD" : string

(list->string ["AA" "BBB" "C" "DD"])
"AABBBCDD" : string
```

2.8 Miscellaneous

string.reverse : string → string

Input: A string S.

Output: S reversed.

```
(string.reverse "abcd")
"dcba" : string
```

string.n-copy : number → string → string

Input: An integer N and a string S.

Output: A string of N copies of S (or an error message if N is negative).

```
(tlstr (string.n-copy 3 " hello"))  \* to get rid of the leading space *\
"hello hello hello" : string

(tlstr (string.n-copy 2.75 " hello"))  \* non-integer is rounded down *\
"hello hello hello" : string
```

string.filter : (string → boolean) → string → string

Input: A predicate P of unit strings and a string S.

Output: The string of all unit strings of S for which P is true.

```
(string.filter lowercase? "Abc1Dd4")
"bcd" : string

(string.filter (/. S (= S "0")) "10011100101")
"00000" : string
```

string.reduce : (string → A → A) → A → string → A

Input: A function F : string → A → A, an element I of type A and a string S.

Output: The (right-left) reduction of S with respect to F. (I is the value of the reduction of "")

Note: reduce is alternatively known as foldr ("fold-right").

If F : string → string → string is the function (/. S Str (if (= S "0") (@s "zero " Str) (@s "one " Str))) and I is "", then a binary-string as input is 'reduced' to a string as shown below

```
(string.reduce (/. S Str (if (= S "0") (@s "zero " Str) (@s "one " Str)))"" "011001")
"zero one one zero zero one " : string
```

string.foldl : (string → A → A) → A → string → A

Input: A function F : string → A → A, an element I of type A and a string S.

Output: The (left-right) reduction of S with respect to F. (I is the value of the reduction of "")

Note: For associative operations string.reduce and string.foldl yield the same result, but not for non-associative operations.

```
(string.foldl (/. S Str (if (= S "0") (@s "zero " Str) (@s "one " Str)))"" "011001")
"one zero zero one one zero " : string \* the reverse of the previous example *\
```

If F : string → number → number is the function (/. S N (+ N 1)) and I is 0, the reduction (either right or left) of a string is the length of the string. Thus, one could define

```
(define strlen'
  { string --> number }
  Str -> (string.reduce (/. S N (+ N 1)) 0 Str))
```

More generally, if P : string → boolean is any predicate of unit strings then F : string → number → number, with F equal to (/. S N (if (P S) (+ N 1) N)) used as an argument in string.reduce (string.foldl), will count all the unit strings of a string S satisfying predicate P.

string.count-strings : (string → boolean) → string → number
Input: A predicate P of unit strings and a string S.
Output: The number of unit strings of S for which P is true.

```
(string.count-strings digit? "a103b48k A*7")
6 : number
```

2.9 String to Number Conversion

string->number : string → number
Input: A string S representing a Shen number
Output: The number corresponding to S, or an error message, if S does not represent a valid Shen number.

```
(string->number "45e3")
45000 : number
```

number->radix : number → number → string
Input: Natural numbers *x* and *y*.
Output: A string representing *x* to the base *y*.

```
(number->radix 255 16)
"ff" : string
```

radix->number : string → number → string
Input: A string representing *x* to the base *y* and the base *y*.
Output: The corresponding decimal number.

```
(radix->number "ff" 16)
255 : number
```