The String Library

(v.3 04-07-12)

© W O Riha

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.

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.

1.2 Important Note

Earlier versions introduced a subtype `ustring` of type `string`; see section 2. This is to provide increased type security. However, there will be programmers who are suspicious of user-defined types and prefer the simplicity of the native Shen type system. For this reason, the string library is now loaded without `ustring`.

Earlier versions also made use of a type `integer`, a subtype of type `number`, also for reasons of type security. It is found, however, that the string library works perfectly ok, without such a type, as most of the functions return meaningful results, when invoked with non-integer values, where whole numbers are normally expected. For examples, see section 5.10.1 (Note on Non-Integer Arguments).

2. Unit Strings

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.

Shen's type system does not distinguish between strings and unit strings: a unit string is just another string.

A string S is a unit string precisely when `(ustring? S)` is `true`.

```
ustring? : string --> boolean
```

Input: A string S.

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

Examples:

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

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

The (annotated) definition of `datatype ustring` may be found in file `ustring-type.shen`.

3. Data Type `string`

`string` is one of the basic data types in Shen. It comes equipped with a number of functions and predicates.

3.1 The Primitive String Functions

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

```
string? : A --> boolean
```

Input: Any object X.

Output: `true` if X is a string, otherwise `false`.

Examples:

```
(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.

Note: a better name for this function would be `n->ustring`, since it returns a value of type `ustring`. This fact is important when type checking. For this reason, the function has been assigned the alternative signature `number --> ustring.`

Examples:

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

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

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 ! *\
```

The current library only caters 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.

Example:

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

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

Note: On the Lisp platform, the following identity holds for any `N` with `0 ≤ N ≤ 1114111 (= 0x10ffff)`

`(string->n (n->string N)) = N` [`string->n` is a left-inverse of `n->string`]

even when `(n->string N)` raises an error. (Try for N >255!)

```
(n->string (string->n "Mark"))  \* n->string is not the inverse of string->n *\
"M" : string
```

`cn : string --> string --> string`

Input: Two strings S1 and S2.

Output: The concatenation of S1 and S2.

Example:

```
(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)

Example:

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

`tlstr : string --> string`

Input: A string S.

Output: S without its first unit string.

Example:

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

`pos : string --> number --> string`

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

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

Example:

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

Note: The function returns what is located at position N in the string, and not a position. A better name would have been `string-ref`, as for example in the Scheme string library. The return value is a unit string and not a general string. The function has therefore been given the alternative signature `string --> number --> ustring`, so as not to cause a type conflict.

To avoid any confusion, the string library provides the alternative function

`string-ref : number --> string --> string`

Input: An integer N and a string S.

Output: The unit string at position N.

Note: The order of the arguments has been changed to conform to our convention of parameter passing (see section 5.1 c).

Example:

```
(string-ref 3 "ABCDE")
D : ustring
```

`str : A --> string`

Input: Any object A.

Output: The normal form of A as a string.

Examples:

```
(str 3.141592653)   \* conversion of a number *\
"3.141592653" : string

(str (+ 2 2))
"4" : string \* expressions are evaluated before conversion *\

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

```
(str "Mark")  \* a string is converted to a string *\
"\"Mark\"" : string       \* note: the double quotes are also mapped! *\
```

3.2 Other String Functions

`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 `""`.

Example:

```
(hdstr "Mark") : string
"M" : ustring
```

A very useful string constructor is

`make-string` (`no type`)

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

Example:

```
(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
```

4. Definitions

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.

5. The Library Functions

5.1 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-` or `ustring-` except when this is redundant and/or clumsy. For example, `string-map` and `ustring-upcase` (see below), 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. For example,

```
(substring 2.9 5.1 "01234567")  \* bounds should be integers *\
"3456"
```

5.2 Unit String Predicates

Note: These predicates are defined for unit strings. A type error will therefore result when a predicate is invoked with a `string` argument.

`digit? : string --> boolean`

Input: A unit string S.

Output: `true` if S is in ["0", "1", …,"9"], otherwise `false.`

Example:

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

`uppercase? : string --> boolean`

Input: A unit string S.

Output: `true` if S is an upper-case letter "A","B", …,"Z", otherwise `false.`

Example:

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

`lowercase? : string --> boolean`

Input: A unit string S.

Output: `true` if S is a lower-case letter "a","b",… "z", otherwise `false.`

Example:

```
(lowercase? "qq")
type error
```

`letter? : string --> boolean`

Input: A unit string S.

Output: `true` if S is an upper- or lower-case letter, otherwise `false.`

Example:

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

(letter? "qq")
type error

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

`whitespace? : string --> boolean`

Input: A unit string S.

Output: `true` if S is in ["c#9;" "c#10;" "c#11;" "c#12;" "c#13;" " "], otherwise `false.`

Example:

```
(whitespace? "q")
false : boolean
```

5.3 Functions on Unit Strings

`string-upcase : ustring --> ustring`

Input: A unit string S.

Output: If S is lower-case the upper-case equivalent, otherwise S.

Example:

```
(ustring-upcase "a")
"A" : string
```

`string-downcase : ustring --> ustring`

Input: A unit string S.

Output: If S is upper-case the lower-case equivalent, otherwise S.

Example:

```
(string-downcase "1")
"1" : string

(string-downcase "A")
"a" : string
```

5.4 Comparison of Unit Strings

The comparison functions for unit strings all have the signature

`string --> string --> boolean`.

Comparison is based on the (ASCII) code of the unit strings. This is consistent with common usage. For general Unicodes, however, this is no longer true, since for example in German, the letter 'ö' represented as `"c#246;"` precedes `"p"` with code 111.

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. `<ustr` less than
d. `>ustr` greater than
e. `<=ustr` less than or equal
f. `>=ustr` greater than or equal

Examples:

```
(<ustr "A" "X")
true : boolean

(>=ustr "A" "y")
false : boolean
```

5.5 String Predicates

`string-prefix? : string --> string --> boolean`

Input: Two strings S1 and S2.

Output: `true` if S1 is a prefix of S2, otherwise `false.`

Example:

```
(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.`

Example:

```
(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.`

Example:

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

5.6 Extending Unit String Predicates to Strings

`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.`

Example: (checks if a string consists entirely of letters and spaces)

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

`string-any? :  (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.`

Example: (checks if a string contains a digit?)

```
(string-any? digit? "String Library")
false : boolean
```

Using predicate `string-every?` it is easy to define functions to test if a given string consists entirely of upper-case letters, is alpha-numeric (whatever your definition), numeric, a digit-sequence, and many more. The library only includes

`digit-string? : string --> boolean`

Input: A string S.

Output: `true` if S is a string consisting of digits (i.e. represents an unsigned integer), otherwise `false.`

Example:

```
(digit-string? "143")
true : boolean
```

5.7 Extending Unit String Functions to Strings

`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

Example: (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.

Example:

```
(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.

Example:

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

5.8 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

Examples:

```
(<=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)) …)`

5.9 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.

Example:

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

(strlen "I prefer 'strlen'!")
18 : 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.

Examples:

```
(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.

Example:

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

5.10 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.

Examples:

```
(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.

Examples:

```
(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.

Example:

```
(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.

Examples:

```
(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.

Examples:

```
(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!

Examples:

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

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

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

15.10.1 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.

Examples:

```
(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.

Example:

```
(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..

Example:

```
(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.

Example:

```
(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.

Example:

```
(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 "".

Examples:

```
(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

(string-pad " " -1 "123456")
"" : string
```

5.11 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.

Examples:

```
(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.

Examples:

```
(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).

Example:

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

5.12 Replacing and Tokenising

`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.

Examples:

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

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

`string-delete-all : string --> string --> string`

Input: Two strings S1, S2.

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

Example:

```
(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!

Example:

```
(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-tokenise : (string --> boolean) --> string --> (list string)`

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

Output: The list of tokens.

Example: (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")
["12" "03" "2012" "13h" "34m" "12.5s"] : (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.

Examples:

```
(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.

Example:

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

5.13 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).

Example:

```
(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.

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

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

5.14 Miscellaneous

`string-reverse : string --> string`

Input: A string S.

Output: S reversed.

Example:
```
(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).

Example:
```
(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`.

Examples:
```
(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").

Examples:

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.

Examples:

```
(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 the function

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-ustrings : (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.

Example:

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

5.15 String to Number Conversion

There is only one function in this section

`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 number.

Examples:

```
(string->number "--0023.78")
23.78 : number

(string->number "--+.367")
0.367 : number

(string->number "--+23.01e-1")
2.301 : number

(string->number "--+23.0p1e-1")
illegal character 'p' in number

(string->number "555.01e3")
555010.0 : number

(string->number "555.01e+3")
illegal character '+' in exponent

(string->number "666.")
fractional part missing!
```

5.16 Radix Conversion

The functions in this section are used to convert between different radix number systems. Only unsigned integers are considered. The attribute "decimal" used in function names indicates an unsigned decimal integer; numbers expressed in other number systems are always represented as strings.

The following function converts from decimal to radix-B. B must be greater than 1, and should be no greater than 36. The radix-B digits are taken from the sequence 0, 1, …, 9, a, b, c, …, z. (Capital letters are permitted).

Note: All the functions in this section will also work for B = 1. This is the ***unary number system***, which has only one 'digit', the tally mark '|'. A decimal number N is represented as a string of N tally marks.

`decimal->radixB : number --> number --> string`

Input: Two integers N and B.

Output: The decimal integer N converted to radix-B (represented as a string),
or an error message, if N or B are not integers.

Examples:

```
(decimal->radixB 65535 16)  \* conversion to hex *\
"ffff" : string

(decimal->radixB 65535 16.7)
radix must be an integer!

(decimal->radixB 65535 8)  \* conversion to octal *\
"177777" : string

(decimal->radixB 65535 2)  \* conversion to binary *\
"1111111111111111" : string

(decimal->radixB 8 1)      \* conversion to unary *\
"||||||||" : string

(decimal->radixB 65535 24) \* conversion to radix 24 *\
"4hif" : string
```

The inverse of `decimal->radixB` is

`radixB->decimal : string --> number --> number`

Input: A string S and an integer B, where S represents an integer in the radix-B number system.

Note: An error is raised if B is not an integer or is less than 1.

Output: If all the unit strings of S are radix-B digits, S is converted to a decimal integer, otherwise an error is raised.

Examples:

```
(radixB->decimal "111111111" 2)  \* conversion from binary *\
511 : number

(radixB->decimal "abc" 16)       \* conversion from hex *\
2748 : number

(radixB->decimal "|||||||||" 1)  \* conversion from unary *\
9 : number

(radixB->decimal "abc" 12)       \* the highest digit in radix-12 is 'b', decimal 11 *\
illegal digit 'c' in radix '12' number

(radixB->decimal "Mark" 28)      \* upper-case is allowed *\
491560 : number
```

```
(radixB->decimal "Willi" 33)   \* radix-33 number system is smallest with a digit 'w' *\
38619918 : number
```

It is easy to combine the preceding two functions to convert between any two number systems.

radixB->radixC : string --> number --> number --> string

Input: A string S and two integers B and C (both ≥ 1).

Output: If all the unit strings of S are radix-B digits, S is converted to a string representing the number in the radix-C number system, otherwise an error is raised.

Examples:

```
(radixB->radixC "345" 8 16)   \* octal -> hex *\
"e5" : string

(radixB->radixC "345" 16 8)   \* hex -> octal *\
"1505" : string

(radixB->radixC "121212" 3 2) \* ternary -> binary *\
"111000111" : string

(radixB->radixC "1011" 2 1)    \* binary -> unary *\
"||||||||||" : string

(radixB->radixC "||||||||||||||||" 1 16) \* unary -> hex *\
"f" : string

(radixB->radixC "abcdef" 16 32) \* hex -> radix-32 *\
"anjff" : string
```

In many programming languages, including C/C++ and Javascript, there is a convention for denoting octal and hexadecimal numbers:

- any 'digit' sequence starting with '0' denotes an octal integer
- any sequence preceded by '0x' is a hex-integer
- any other digit sequence is taken as decimal.

Examples:

1234 is a decimal integer

01234 is an octal number (668 in decimal)

0x1234 is a hex number (4660 in decimal)

A function `string->integer` with either one or two argument(s), which therefore has no type, is available. This function (vaguely inspired by the Javascript function `parseInt`) behaves as follows:

(1) when invoked with one argument S (of type string) assumes that S is the string representation of either a hex, octal, unary or decimal integer, and attempts to convert it to a decimal integer.

(2) when invoked with two arguments, a string S and an integer B ≥ 1, converts S to decimal, assuming that S represents a radix-B (or unary) integer.

Examples:

```
(string->integer "001234")   \* one argument - octal assumed *\
668 : number

(string->integer "001234" 10) \* radix 10 specified - octal overridden *\
1234 : number
```

```
(string->decimal "0x1234")  \* prefix ‘0x’ - hex assumed *\
4660 : number

(string->integer "0x1234" 10)   \* radix 10 specified *\
illegal digit 'x' in radix '10' number

(string->integer "||||")      \* from unary *\
4 : number

(string->integer "||||||" 1)   \* from unary *\
6 : number

(string->integer "0x1234" 16)   \* radix-16 (hex) *\
4660 : number
```