

Strings

Summary: In this reading, we consider yet another basic data type: Strings. A string is a sequence of characters. Unlike symbols, which are *atomic*, strings can be separated into constituent parts.

Related readings:

- Characters

Procedures covered in this reading

- String predicates: `string?`
- String constructors: `make-string`, `string`, `string-append`
- String destructors: `string-ref`, `substring`
- String conversion: `list->string`, `number->string`, `string->list`, `string->number`
- String analysis: `string-length`,
- String comparison: `string<?`, `string<=?`, `string=?`, `string>=?`, `string>?`, `string-ci<?`, `string-ci<=?`, `string-ci=?`, `string-ci>=?`, `string-ci>?`

Contents:

- String Basics
- String Procedures

String Basics

As we've just learned in the reading on characters, characters provide the basic building blocks of the things we might call "texts". What do we do with characters? We combine them into strings.

A string is a sequence of zero or more characters. Most strings can be named by enclosing the characters they contain between plain double quotation marks, to produce a *string literal*. For instance, "hyperbola" is the nine-character string consisting of the characters `#\h`, `#\y`, `#\p`, `#\e`, `#\r`, `#\b`, `#\o`, `#\l`, and `#\a`, in that order. Similarly, "" is the zero-character string (the *null string*).

String literals may contain spaces and newline characters; when such characters are between double quotation marks, they are treated like any other characters in the string. There is a slight problem when one wants to put a double quotation mark into a string literal: To indicate that the double quotation mark is part of the string (rather than marking the end of the string), one must place a backslash character immediately in front of it. For instance, "Say \"hi\"" is the eight-character string consisting of the characters `#\S`, `#\a`, `#\y`, `#\space`, `#\"`, `#\h`, `#\i`, and `#\"`, in that order. The backslash before a double quotation mark in a string literal is an *escape* character, present only to indicate that the character immediately following it is part of the string.

This use of the backslash character causes yet another slight problem: What if one wants to put a backslash into a string? The solution is similar: Place another backslash character immediately in front of it. For instance, "a\\b" is the three-character string consisting of the characters #\a, #\\, and #\b, in that order. The first backslash in the string literal is an escape, and the second is the character that it protects, the one that is part of the string.

String Procedures

Scheme provides several basic procedures for working with strings:

The (string? *val*) predicate determines whether its argument is or is not a string.

The (make-string *count char*) procedure constructs and returns a string that consists of *count* repetitions of a single character. Its first argument indicates how long the string should be, and the second argument specifies which character it should be made of. For instance, the following code constructs and returns the string "aaaaa".

```
> (make-string 5 #\a)
"aaaaa"
```

The (string *ch1 ... chn*) procedure takes any number of characters as arguments and constructs and returns a string consisting of exactly those characters. For instance, (string #\H #\i #\!) constructs and returns the string "Hi!". This procedure can be useful for building strings with quotes. For example, (string #\" #\") produces "\"\"". (Isn't that ugly?)

The string->list procedure converts a string into a list of characters. The list->string procedure converts a list of characters into a string. It is invalid to call list->string on a non-list or on a list that contains values other than characters.

```
> (string->list "Hello")
(#\H #\e #\l #\l #\o)
> (list->string (list #\a #\b #\c))
"abc"
> (list->string (list 'a 'b))
list->string: expects argument of type <list of character>; given (a b)
```

The string-length procedure takes any string as argument and returns the number of characters in that string. For instance, the value of (string-length "parabola") is 8 and the value of (string-length "a\\b") is 3.

The string-ref procedure is used to select the character at a specified position within a string. Like list-ref, string-ref presupposes *zero-based indexing*; the position is specified by the number of characters that precede it in the string. (So the initial character in the string is at position 0, the next at position 1, and so on.) For instance, the value of (string-ref "ellipse" 4) is #\p -- the character that follows four other characters and so is at position 4 in zero-based indexing.

Strings can be compared for “lexicographic order”, the extension of alphabetical order that is derived from the collating sequence of the local character set. Once more, Scheme provides both case-sensitive and case-insensitive versions of these predicates: string<?, string<=?, string=?, string=>?, and

`string>?` are the case-sensitive versions, and `string-ci<?`, `string-ci<=?`, `string-ci=?`, `string-ci>=?`, and `string-ci>?` the case-insensitive ones.

The `(substring str start end)` procedure takes three arguments. The first is a string and the second and third are non-negative integers not exceeding the length of that string. `Substring` returns the part of its first argument that starts after the number of characters specified by the second argument and ends after the number of characters specified by the third argument. For instance: `(substring "hypocycloid" 3 8)` returns the substring "ocycl" -- the substring that starts after the initial "hyp" and ends after the eighth character, the l. (If you think of the characters in a string as being numbered starting at 0, `substring` takes the characters from *start* to *end*-1.)

The `string-append` procedure takes any number of strings as arguments and returns a string formed by concatenating those arguments. For instance, the value of `(string-append "al" "fal" "fa")` is "alfalfa".

The `number->string` procedure takes any Scheme number as its argument and returns a string that denotes the number.

```
> (number->string 23)
"23"
> (number->string 6/5)
6/5"
> (number->string #i6/5)
"1.2"
> (number->string (sqrt -1))
"0+1i"
```

The `string->number` procedure provides the inverse operation. Given a string that represents a number, it returns the corresponding number. Fascinatingly, unlike other procedures that give up if you give them inappropriate input, when `string->number` is called with a string that does not represent a number, it returns the value `#f` (which represents “no” or “false”).

```
> (string->number "23")
23
> (string->number "6/5")
1 1/5
> (string->number "3+4i")
3+4i
> (string->number "")
#f
> (string->number "two")
#f
> (string->number "3 + 4i")
#f
```

Copyright © 2006 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.