

Programming Language—Common Lisp

16. Strings

Version 15.17R, X3J13/94-101R.
Fri 12-Aug-1994 6:35pm EDT

16.1 String Concepts

16.1.1 Implications of Strings Being Arrays

Since all *strings* are *arrays*, all rules which apply generally to *arrays* also apply to *strings*. See Section 15.1 (Array Concepts).

For example, *strings* can have *fill pointers*, and *strings* are also subject to the rules of *element type upgrading* that apply to *arrays*.

16.1.2 Subtypes of STRING

All functions that operate on *strings* will operate on *subtypes* of *string* as well.

However, the consequences are undefined if a *character* is inserted into a *string* for which the *element type* of the *string* does not include that *character*.

string

System Class

Class Precedence List:

string, vector, array, sequence, t

Description:

A *string* is a *specialized vector* whose *elements* are of *type* **character** or a *subtype* of *type* **character**. When used as a *type specifier* for object creation, **string** means (**vector character**).

Compound Type Specifier Kind:

Abbreviating.

Compound Type Specifier Syntax:

(string [*size*])

Compound Type Specifier Arguments:

size—a non-negative *fixnum*, or the *symbol* *.

Compound Type Specifier Description:

This denotes the union of all *types* (array *c* (*size*)) for all *subtypes* *c* of **character**; that is, the set of *strings* of size *size*.

See Also:

Section 16.1 (String Concepts), Section 2.4.5 (Double-Quote), Section 22.1.3.4 (Printing Strings)

base-string

Type

Supertypes:

base-string, string, vector, array, sequence, t

Description:

The *type* **base-string** is equivalent to (**vector base-char**). The *base string* representation is the most efficient *string* representation that can hold an arbitrary sequence of *standard characters*.

Compound Type Specifier Kind:

Abbreviating.

Compound Type Specifier Syntax:

(base-string [*size*])

Compound Type Specifier Arguments:

size—a non-negative *fixnum*, or the *symbol* `*`.

Compound Type Specifier Description:

This is equivalent to the type `(vector base-char size)`; that is, the set of *base strings* of size *size*.

simple-string

Type

Supertypes:

`simple-string`, `string`, `vector`, `simple-array`, `array`, `sequence`, `t`

Description:

A *simple string* is a specialized one-dimensional *simple array* whose *elements* are of *type* **character** or a *subtype* of *type* **character**. When used as a *type specifier* for object creation, **simple-string** means `(simple-array character (size))`.

Compound Type Specifier Kind:

Abbreviating.

Compound Type Specifier Syntax:

`(simple-string [size])`

Compound Type Specifier Arguments:

size—a non-negative *fixnum*, or the *symbol* `*`.

Compound Type Specifier Description:

This denotes the union of all *types* `(simple-array c (size))` for all *subtypes* *c* of **character**; that is, the set of *simple strings* of size *size*.

simple-base-string

Type

Supertypes:

simple-base-string, base-string, simple-string, string, vector, simple-array, array, sequence, t

Description:

The *type* **simple-base-string** is equivalent to `(simple-array base-char (*))`.

Compound Type Specifier Kind:

Abbreviating.

Compound Type Specifier Syntax:

`(simple-base-string [size])`

Compound Type Specifier Arguments:

size—a non-negative *fixnum*, or the *symbol* `*`.

Compound Type Specifier Description:

This is equivalent to the type `(simple-array base-char (size))`; that is, the set of *simple base strings* of size *size*.

simple-string-p

Function

Syntax:

`simple-string-p object` \rightarrow *generalized-boolean*

Arguments and Values:

object—an *object*.

generalized-boolean—a *generalized boolean*.

Description:

Returns *true* if *object* is of *type* **simple-string**; otherwise, returns *false*.

Examples:

```
(simple-string-p "aaaaaa")  $\rightarrow$  true
(simple-string-p (make-array 6
                             :element-type 'character
                             :fill-pointer t))  $\rightarrow$  false
```

Notes:

(simple-string-p *object*) \equiv (typep *object* 'simple-string)

char, schar

Accessor

Syntax:

`char` *string index* \rightarrow *character*
`schar` *string index* \rightarrow *character*

(setf (`char` *string index*) *new-character*)
(setf (`schar` *string index*) *new-character*)

Arguments and Values:

string—for **char**, a *string*; for **schar**, a *simple string*.

index—a *valid array index* for the *string*.

character, *new-character*—a *character*.

Description:

char and **schar** *access* the *element* of *string* specified by *index*.

char ignores *fill pointers* when *accessing elements*.

Examples:

```
(setq my-simple-string (make-string 6 :initial-element #\A))  $\rightarrow$  "AAAAAA"
(schar my-simple-string 4)  $\rightarrow$  #\A
(setf (schar my-simple-string 4) #\B)  $\rightarrow$  #\B
my-simple-string  $\rightarrow$  "AAAABA"
(setq my-filled-string
  (make-array 6 :element-type 'character
              :fill-pointer 5
              :initial-contents my-simple-string))
 $\rightarrow$  "AAAAB"
(char my-filled-string 4)  $\rightarrow$  #\B
(char my-filled-string 5)  $\rightarrow$  #\A
(setf (char my-filled-string 3) #\C)  $\rightarrow$  #\C
(setf (char my-filled-string 5) #\D)  $\rightarrow$  #\D
(setf (fill-pointer my-filled-string) 6)  $\rightarrow$  6
my-filled-string  $\rightarrow$  "AAACBD"
```

See Also:

`aref`, `elt`, Section 3.2.1 (Compiler Terminology)

Notes:

`(char s j) ≡ (aref (the string s) j)`

string

Function

Syntax:

`string x` \rightarrow *string*

Arguments and Values:

x—a *string*, a *symbol*, or a *character*.

string—a *string*.

Description:

Returns a *string* described by *x*; specifically:

- If *x* is a *string*, it is returned.
- If *x* is a *symbol*, its *name* is returned.
- If *x* is a *character*, then a *string* containing that one *character* is returned.
- `string` might perform additional, *implementation-defined* conversions.

Examples:

```
(string "already a string")  $\rightarrow$  "already a string"  
(string 'elm)  $\rightarrow$  "ELM"  
(string #\c)  $\rightarrow$  "c"
```

Exceptional Situations:

In the case where a conversion is defined neither by this specification nor by the *implementation*, an error of *type* **type-error** is signaled.

See Also:

`coerce`, `string` (*type*).

Notes:

`coerce` can be used to convert a *sequence* of *characters* to a *string*.

prin1-to-string, **princ-to-string**, **write-to-string**, or **format** (with a first argument of **nil**) can be used to get a *string* representation of a *number* or any other *object*.

string-upcase, string-downcase, string-capitalize, nstring-upcase, nstring-downcase, nstring- capitalize

Function

Syntax:

string-upcase *string* &key *start end* → *cased-string*
string-downcase *string* &key *start end* → *cased-string*
string-capitalize *string* &key *start end* → *cased-string*

nstring-upcase *string* &key *start end* → *string*
nstring-downcase *string* &key *start end* → *string*
nstring-capitalize *string* &key *start end* → *string*

Arguments and Values:

string—a *string designator*. For **nstring-upcase**, **nstring-downcase**, and **nstring-capitalize**, the *string designator* must be a *string*.

start, *end*—*bounding index designators* of *string*. The defaults for *start* and *end* are 0 and **nil**, respectively.

cased-string—a *string*.

Description:

string-upcase, **string-downcase**, **string-capitalize**, **nstring-upcase**, **nstring-downcase**, **nstring-capitalize** change the case of the subsequence of *string* bounded by *start* and *end* as follows:

string-upcase

string-upcase returns a *string* just like *string* with all lowercase characters replaced by the corresponding uppercase characters. More precisely, each character of the result *string* is produced by applying the *function* **char-upcase** to the corresponding character of *string*.

string-downcase

string-downcase is like **string-upcase** except that all uppercase characters are replaced by the corresponding lowercase characters (using **char-downcase**).

string-upcase, string-downcase, string-capitalize, ...

string-capitalize

string-capitalize produces a copy of *string* such that, for every word in the copy, the first *character* of the “word,” if it has *case*, is *uppercase* and any other *characters* with *case* in the word are *lowercase*. For the purposes of **string-capitalize**, a “word” is defined to be a consecutive subsequence consisting of *alphanumeric characters*, delimited at each end either by a non-*alphanumeric character* or by an end of the *string*.

nstring-upcase, nstring-downcase, nstring-capitalize

nstring-upcase, **nstring-downcase**, and **nstring-capitalize** are identical to **string-upcase**, **string-downcase**, and **string-capitalize** respectively except that they modify *string*.

For **string-upcase**, **string-downcase**, and **string-capitalize**, *string* is not modified. However, if no characters in *string* require conversion, the result may be either *string* or a copy of it, at the implementation’s discretion.

Examples:

```
(string-upcase "abcde") → "ABCDE"
(string-upcase "Dr. Livingston, I presume?")
→ "DR. LIVINGSTON, I PRESUME?"
(string-upcase "Dr. Livingston, I presume?" :start 6 :end 10)
→ "Dr. LiViNGston, I presume?"
(string-downcase "Dr. Livingston, I presume?")
→ "dr. livingston, i presume?"

(string-capitalize "elm 13c arthur;fig don't") → "Elm 13c Arthur;Fig Don'T"
(string-capitalize " hello ") → " Hello "
(string-capitalize "occlUDeD cASEmenTs F0reSTAll iNADVertent DEFenestraTION")
→ "Occluded Casements Forestall Inadvertent Defenestration"
(string-capitalize 'kludgy-hash-search) → "Kludgy-Hash-Search"
(string-capitalize "DON'T!") → "Don'T!" ;not "Don't!"
(string-capitalize "pipe 13a, foo16c") → "Pipe 13a, Foo16c"

(setq str (copy-seq "0123ABCD890a")) → "0123ABCD890a"
(nstring-downcase str :start 5 :end 7) → "0123AbcD890a"
str → "0123AbcD890a"
```

Side Effects:

nstring-upcase, **nstring-downcase**, and **nstring-capitalize** modify *string* as appropriate rather than constructing a new *string*.

See Also:

char-upcase, **char-downcase**

Notes:

The result is always of the same length as *string*.

string-trim, string-left-trim, string-right-trim *Function*

Syntax:

string-trim *character-bag string* → *trimmed-string*
string-left-trim *character-bag string* → *trimmed-string*
string-right-trim *character-bag string* → *trimmed-string*

Arguments and Values:

character-bag—a *sequence* containing *characters*.

string—a *string designator*.

trimmed-string—a *string*.

Description:

string-trim returns a substring of *string*, with all characters in *character-bag* stripped off the beginning and end. **string-left-trim** is similar but strips characters off only the beginning; **string-right-trim** strips off only the end.

If no *characters* need to be trimmed from the *string*, then either *string* itself or a copy of it may be returned, at the discretion of the implementation.

All of these *functions* observe the *fill pointer*.

Examples:

```
(string-trim "abc" "abcaakaaakabcaaa") → "kaaak"
(string-trim '(#\Space #\Tab #\Newline) " garbanzo beans
") → "garbanzo beans"
(string-trim " (*)" " ( *three (silly) words* ) ")
→ "three (silly) words"

(string-left-trim "abc" "labcabcab") → "labcabcab"
(string-left-trim " (*)" " ( *three (silly) words* ) ")
→ "three (silly) words* ) "

(string-right-trim " (*)" " ( *three (silly) words* ) ")
→ " ( *three (silly) words"
```

Affected By:

The *implementation*.

**string=, string/=, string<, string>, string<=,
string>=, string-equal, string-not-equal, string-
lessp, string-greaterp, string-not-greaterp, string-
not-lessp** *Function*

Syntax:

`string= string1 string2 &key start1 end1 start2 end2` → *generalized-boolean*

`string/= string1 string2 &key start1 end1 start2 end2` → *mismatch-index*
`string< string1 string2 &key start1 end1 start2 end2` → *mismatch-index*
`string> string1 string2 &key start1 end1 start2 end2` → *mismatch-index*
`string<= string1 string2 &key start1 end1 start2 end2` → *mismatch-index*
`string>= string1 string2 &key start1 end1 start2 end2` → *mismatch-index*

`string-equal string1 string2 &key start1 end1 start2 end2` → *generalized-boolean*

`string-not-equal string1 string2 &key start1 end1 start2 end2` → *mismatch-index*
`string-lessp string1 string2 &key start1 end1 start2 end2` → *mismatch-index*
`string-greaterp string1 string2 &key start1 end1 start2 end2` → *mismatch-index*
`string-not-greaterp string1 string2 &key start1 end1 start2 end2` → *mismatch-index*
`string-not-lessp string1 string2 &key start1 end1 start2 end2` → *mismatch-index*

Arguments and Values:

string1—a *string designator*.

string2—a *string designator*.

start1, *end1*—*bounding index designators* of *string1*. The defaults for *start* and *end* are 0 and **nil**, respectively.

start2, *end2*—*bounding index designators* of *string2*. The defaults for *start* and *end* are 0 and **nil**, respectively.

generalized-boolean—a *generalized boolean*.

mismatch-index—a *bounding index* of *string1*, or **nil**.

Description:

These functions perform lexicographic comparisons on *string1* and *string2*. **string=** and **string-equal** are called equality functions; the others are called inequality functions. The

string=, string/=, string<, string>, string<=, ...

comparison operations these *functions* perform are restricted to the subsequence of *string1* bounded by *start1* and *end1* and to the subsequence of *string2* bounded by *start2* and *end2*.

A string *a* is equal to a string *b* if it contains the same number of characters, and the corresponding characters are the *same* under **char=** or **char-equal**, as appropriate.

A string *a* is less than a string *b* if in the first position in which they differ the character of *a* is less than the corresponding character of *b* according to **char<** or **char-lessp** as appropriate, or if string *a* is a proper prefix of string *b* (of shorter length and matching in all the characters of *a*).

The equality functions return a *generalized boolean* that is *true* if the strings are equal, or *false* otherwise.

The inequality functions return a *mismatch-index* that is *true* if the strings are not equal, or *false* otherwise. When the *mismatch-index* is *true*, it is an *integer* representing the first character position at which the two substrings differ, as an offset from the beginning of *string1*.

The comparison has one of the following results:

string=

string= is *true* if the supplied substrings are of the same length and contain the *same* characters in corresponding positions; otherwise it is *false*.

string/=

string/= is *true* if the supplied substrings are different; otherwise it is *false*.

string-equal

string-equal is just like **string=** except that differences in case are ignored; two characters are considered to be the same if **char-equal** is *true* of them.

string<

string< is *true* if substring1 is less than substring2; otherwise it is *false*.

string>

string> is *true* if substring1 is greater than substring2; otherwise it is *false*.

string-lessp, string-greaterp

string-lessp and **string-greaterp** are exactly like **string<** and **string>**, respectively, except that distinctions between uppercase and lowercase letters are ignored. It is as if **char-lessp** were used instead of **char<** for comparing characters.

string<=

string<= is *true* if substring1 is less than or equal to substring2; otherwise it is *false*.

string>=

string>= is *true* if **substring1** is greater than or equal to **substring2**; otherwise it is *false*.

string-not-greaterp, **string-not-lessp**

string-not-greaterp and **string-not-lessp** are exactly like **string<=** and **string>=**, respectively, except that distinctions between uppercase and lowercase letters are ignored. It is as if **char-lessp** were used instead of **char<** for comparing characters.

Examples:

```
(string= "foo" "foo") → true
(string= "foo" "Foo") → false
(string= "foo" "bar") → false
(string= "together" "frog" :start1 1 :end1 3 :start2 2) → true
(string-equal "foo" "Foo") → true
(string= "abcd" "01234abcd9012" :start2 5 :end2 9) → true
(string< "aaaa" "aaab") → 3
(string>= "aaaaa" "aaaa") → 4
(string-not-greaterp "Abcde" "abcdE") → 5
(string-lessp "012AAAA789" "01aaab6" :start1 3 :end1 7
              :start2 2 :end2 6) → 6
(string-not-equal "AAAA" "aaaA") → false
```

See Also:

char=

Notes:

equal calls **string=** if applied to two *strings*.

stringp

Function

Syntax:

stringp *object* → *generalized-boolean*

Arguments and Values:

object—an *object*.

generalized-boolean—a *generalized boolean*.

Description:

Returns *true* if *object* is of *type string*; otherwise, returns *false*.

Examples:

```
(stringp "aaaaaa") → true  
(stringp #\a) → false
```

See Also:

`typep`, `string` (*type*)

Notes:

```
(stringp object) ≡ (typep object 'string)
```

make-string

Function

Syntax:

```
make-string size &key initial-element element-type → string
```

Arguments and Values:

size—a *valid array dimension*.

initial-element—a *character*. The default is *implementation-dependent*.

element-type—a *type specifier*. The default is **character**.

string—a *simple string*.

Description:

make-string returns a *simple string* of length *size* whose elements have been initialized to *initial-element*.

The *element-type* names the *type* of the *elements* of the *string*; a *string* is constructed of the most *specialized type* that can accommodate *elements* of the given *type*.

Examples:

```
(make-string 10 :initial-element #\5) → "5555555555"  
(length (make-string 10)) → 10
```

Affected By:

The *implementation*.

Version 15.17R, X3J13/94-101R.
Fri 12-Aug-1994 6:35pm EDT
