# 9. Strings

Strings are a type of array representing a sequence of characters. The printed representation of a string is its characters enclosed in quotation marks, for example "foo bar". Strings are constants, that is, evaluating a string returns that string. Strings are the right data type to use for text-processing.

Strings are arrays of type art-string, where each element holds an eight-bit unsigned fixnum. This is because characters are represented as fixnums, and for fundamental characters only eight bits are used. A string can also be an array of type art-fat-string, where each element holds a sixteen-bit unsigned fixnum; the extra bits allow for multiple fonts or an expanded character set.

Characters are actually fixnums, as explained in section 21.1, page 362. Note that you can type in the fixnums that represent characters using "#/" and "#\"; for example, #/f reads in as the fixnum that represents the character "f", and #\return reads in as the fixnum that represents the special Return character. See page 374 for details of this syntax.

The functions described in this section provide a variety of useful operations on strings. In place of a string, most of these functions will accept a symbol or a fixnum as an argument, and will coerce it into a string. Given a symbol, its print name, which is a string, will be used. Given a fixnum, a one-character string containing the character designated by that fixnum will be used. Several of the functions actually work on any type of one-dimensional array and may be useful for other than string processing; these are the functions such as substring and string-length which do not depend on the elements of the string being characters.

Since strings are arrays, the usual array-referencing function aref is used to extract the characters of the string as fixnums. For example,

```
(aref "frob" 1) => 162  ;lower-case r
```

Note that the character at the beginning of the string is element zero of the array (rather than one); as usual in Zetalisp, everything is zero-based.

It is also legal to store into strings (using aset). As with rplaca on lists, this changes the actual object; one must be careful to understand where side-effects will propagate. When you are making strings that you intend to change later, you probably want to create an array with a fill-pointer (see page 124) so that you can change the length of the string as well as the contents. The length of a string is always computed using array-active-length, so that if a string has a fill-pointer, its value will be used as the length.

## 9.1 Characters

**character** *x*

character coerces *x* to a single character, represented as a fixnum. If *x* is a number, it is returned. If *x* is a string or an array, its first element is returned. If *x* is a symbol, the first character of its pname is returned. Otherwise an error occurs. The way characters are represented as fixnums is explained in section 21.1, page 362.

**char-equal** *ch1 ch2*

This is the primitive for comparing characters for equality; many of the string functions call it. *ch1* and *ch2* must be fixnums. The result is t if the characters are equal ignoring case and font, otherwise nil. %%ch-char is the byte-specifier for the portion of a character that excludes the font information.

**char-lessp** *ch1 ch2*

This is the primitive for comparing characters for order; many of the string functions call it. *ch1* and *ch2* must be fixnums. The result is t if *ch1* comes before *ch2* ignoring case and font, otherwise nil. Details of the ordering of characters are in section 21.1, page 362.

## 9.2 Upper and Lower Case Letters

**alphabetic-case-affects-string-comparison** *Variable*

This variable is normally nil. If it is t, char-equal, char-lessp, and the string searching and comparison functions will distinguish between upper-case and lower-case letters. If it is nil, lower-case characters behave as if they were the same character but in upper-case. It is all right to bind this to t around a string operation, but changing its global value to t will break many system functions and user interfaces and so is not recommended.

**char-upcase** *ch*

If *ch*, which must be a fixnum, is a lower-case alphabetic character its upper-case form is returned; otherwise, *ch* itself is returned. If font information is present it is preserved.

**char-downcase** *ch*

If *ch*, which must be a fixnum, is a upper-case alphabetic character its lower-case form is returned; otherwise, *ch* itself is returned. If font information is present it is preserved.

**string-upcase** *string*

Returns a copy of *string*, with all lower-case alphabetic characters replaced by the corresponding upper-case characters.

**string-downcase** *string*

Returns a copy of *string*, with all upper-case alphabetic characters replaced by the corresponding lower-case characters.

**string-capitalize-words** *string* &optional (*copy-p* t) (*spaces* t)

Puts each word in *string* into lower-case with an upper case initial, and if *spaces* is non-nil replaces each hyphen character with a space.

If *copy-p* is t, the value is a copy of *string*, and *string* itself is unchanged. Otherwise, *string* itself is returned, with its contents changed.

## 9.3 Basic String Operations

**string** *x*

string coerces *x* into a string. Most of the string functions apply this to their string arguments. If *x* is a string (or any array), it is returned. If *x* is a symbol, its pname is returned. If *x* is a non-negative fixnum less than 400 octal, a one-character-long string containing it is created and returned. If *x* is a pathname (see chapter 22, page 453), the "string for printing" is returned. Otherwise, an error is signalled.

If you want to get the printed representation of an object into the form of a string, this function is *not* what you should use. You can use format, passing a first argument of nil (see page 411). You might also want to use with-output-to-string (see page 151).

**string-length** *string*

string-length returns the number of characters in *string*. This is 1 if *string* is a number, the array-active-length (see page 130) if *string* is an array, or the array-active-length of the pname if *string* is a symbol.

**string-equal** *string1 string2* &optional (*idx1* 0) (*idx2* 0) *lim1 lim2*

string-equal compares two strings, returning t if they are equal and nil if they are not. The comparison ignores the extra "font" bits in 16-bit strings and ignores alphabetic case. equal calls string-equal if applied to two strings.

The optional arguments *idx1* and *idx2* are the starting indices into the strings. The optional arguments *lim1* and *lim2* are the final indices; the comparison stops just *before* the final index. *lim1* and *lim2* default to the lengths of the strings. These arguments are provided so that you can efficiently compare substrings.

Examples:

```
(string-equal "Foo" "foo") => t
(string-equal "foo" "bar") => nil
(string-equal "element" "select" 0 1 3 4) => t
```

**%string-equal** *string1 idx1 string2 idx2 count*

%string-equal is the microcode primitive used by string-equal. It returns t if the *count* characters of *string1* starting at *idx1* are char-equal to the *count* characters of *string2* starting at *idx2*, or nil if the characters are not equal or if *count* runs off the length of either array.

Instead of a fixnum, *count* may also be nil. In this case, %string-equal compares the substring from *idx1* to (string-length *string1*) against the substring from *idx2* to (string-length *string2*). If the lengths of these substrings differ, then they are not equal and nil

is returned.

Note that *string1* and *string2* must really be strings; the usual coercion of symbols and fixnums to strings is not performed. This function is documented because certain programs which require high efficiency and are willing to pay the price of less generality may want to use %string-equal in place of string-equal.

Examples:

To compare the two strings *foo* and *bar*:

```
(%string-equal foo 0 bar 0 nil)
```

To see if the string *foo* starts with the characters "bar":

```
(%string-equal foo 0 "bar" 0 3)
```

**string-lessp** *string1 string2*

string-lessp compares two strings using dictionary order (as defined by char-lessp). The result is t if *string1* is the lesser, or nil if they are equal or *string2* is the lesser.

**string-compare** *string1 string2* &optional (*idx1 0*) (*idx2 0*) *lim1 lim2*

string-compare compares two strings using dictionary order (as defined by char-lessp). The arguments are interpreted as in string-equal. The result is 0 if the strings are equal, a negative number if *string1* is less than *string2*, or a positive number if *string1* is greater than *string2*. If the strings are not equal, the absolute value of the number returned is one greater than the index (in *string1*) where the first difference occurred.

**substring** *string start* &optional *end area*

This extracts a substring of *string*, starting at the character specified by *start* and going up to but not including the character specified by *end*. *start* and *end* are 0-origin indices. The length of the returned string is *end* minus *start*. If *end* is not specified it defaults to the length of *string*. The area in which the result is to be consed may be optionally specified.

Example:

```
(substring "Nebuchadnezzar" 4 8) => "chad"
```

**nsubstring** *string start* &optional *end area*

nsubstring is the same as substring except that the substring is not copied; instead an indirect array (see page 125) is created which shares part of the argument *string*. Modifying one string will modify the other.

Note that nsubstring does not necessarily use less storage than substring; an nsubstring of any length uses at least as much storage as a substring 12 characters long. So you shouldn't use this just "for efficiency"; it is intended for uses in which it is important to have a substring which, if modified, will cause the original string to be modified too.

**string-append** &rest *strings*

Any number of strings are copied and concatenated into a single string. With a single argument, string-append simply copies it. If there are no arguments, the value is an empty string. In fact, arrays of any type may be used as arguments, and the value will be of the same type as the first argument. Thus string-append can be used to copy and concatenate any type of 1-dimensional array. If the first argument is not an array (for

example, if it is a character), the value is a string.
Example:

```
(string-append #/! "foo" #/!) => "!foo!"
```

**string-nconc** *modified-string* &rest *strings*

string-nconc is like string-append except that instead of making a new string containing the concatenation of its arguments, string-nconc modifies its first argument. *modified-string* must have a fill-pointer so that additional characters can be tacked onto it. Compare this with array-push-extend (page 134). The value of string-nconc is *modified-string* or a new, longer copy of it; in the latter case the original copy is forwarded to the new copy (see adjust-array-size, page 132). Unlike nconc, string-nconc with more than two arguments modifies only its first argument, not every argument but the last.

**string-trim** *char-set string*

This returns a substring of *string*, with all characters in *char-set* stripped off the beginning and end. *char-set* is a set of characters, which can be represented as a list of characters, a string of characters or a single character.
Example:

```
(string-trim '(#\sp) "  Dr. No  ") => "Dr. No"
(string-trim "ab" "abbafooabb") => "foo"
```

**string-left-trim** *char-set string*

This returns a substring of *string*, with all characters in *char-set* stripped off the beginning. *char-set* is a set of characters, which can be represented as a list of characters, a string of characters or a single character.

**string-right-trim** *char-set string*

This returns a substring of *string*, with all characters in *char-set* stripped off the end. *char-set* is a set of characters, which can be represented as a list of characters, a string of characters or a single character.

**string-remove-fonts** *string*

Returns a copy of *string* with each character truncated to 8 bits; that is, changed to font zero.

If *string* is an ordinary string of array type art-string, this does not change anything, but it makes a difference if *string* is an art-fat-string.

**string-reverse** *string*

Returns a copy of *string* with the order of characters reversed. This will reverse a one-dimensional array of any type.

**string-nreverse** *string*

Returns *string* with the order of characters reversed, smashing the original string rather than creating a new one. If *string* is a number, it is simply returned without consing up a string. This will reverse a one-dimensional array of any type.

**string-pluralize** *string*

string-pluralize returns a string containing the plural of the word in the argument *string*. Any added characters go in the same case as the last character of *string*.

Example:

```
(string-pluralize "event") => "events"
(string-pluralize "Man") => "Men"
(string-pluralize "Can") => "Cans"
(string-pluralize "key") => "keys"
(string-pluralize "TRY") => "TRIES"
```

For words with multiple plural forms depending on the meaning, string-pluralize cannot always do the right thing.

## 9.4 String Searching

**string-search-char** *char string* &optional (*from 0*) *to*

string-search-char searches through *string* starting at the index *from*, which defaults to the beginning, and returns the index of the first character that is char-equal to *char*, or nil if none is found. If the *to* argument is supplied, it is used in place of (string-length *string*) to limit the extent of the search.

Example:

```
(string-search-char #/a "banana") => 1
```

**%string-search-char** *char string from to*

%string-search-char is the microcode primitive called by string-search-char and other functions. *string* must be an array and *char*, *from*, and *to* must be fixnums. Except for this lack of type-coercion, and the fact that none of the arguments is optional, %string-search-char is the same as string-search-char. This function is documented for the benefit of those who require the maximum possible efficiency in string searching.

**string-search-not-char** *char string* &optional (*from 0*) *to*

string-search-not-char searches through *string* starting at the index *from*, which defaults to the beginning, and returns the index of the first character that is *not* char-equal to *char*, or nil if none is found. If the *to* argument is supplied, it is used in place of (string-length *string*) to limit the extent of the search.

Example:

```
(string-search-not-char #/b "banana") => 1
```

**string-search** *key string* &optional (*from 0*) *to*

string-search searches for the string *key* in the string *string*. The search begins at *from*, which defaults to the beginning of *string*. The value returned is the index of the first character of the first instance of *key*, or nil if none is found. If the *to* argument is supplied, it is used in place of (string-length *string*) to limit the extent of the search.

Example:

```
(string-search "an" "banana") => 1
(string-search "an" "banana" 2) => 3
```

**string-search-set** *char-set string* &optional (*from 0*) *to*

string-search-set searches through *string* looking for a character that is in *char-set*. The search begins at the index *from*, which defaults to the beginning. It returns the index of the first character that is char-equal to some element of *char-set*, or nil if none is found. If the *to* argument is supplied, it is used in place of (string-length *string*) to limit the extent of the search. *char-set* is a set of characters, which can be represented as a list of characters, a string of characters or a single character.

Example:

```
(string-search-set '(#/n #/o) "banana") => 2
(string-search-set "no" "banana") => 2
```

**string-search-not-set** *char-set string* &optional (*from 0*) *to*

string-search-not-set searches through *string* looking for a character which is not in *char-set*. The search begins at the index *from*, which defaults to the beginning. It returns the index of the first character which is not char-equal to any element of *char-set*, or nil if none is found. If the *to* argument is supplied, it is used in place of (string-length *string*) to limit the extent of the search. *char-set* is a set of characters, which can be represented as a list of characters, a string of characters or a single character.

Example:

```
(string-search-not-set '(#/a #/b) "banana") => 2
```

**string-reverse-search-char** *char string* &optional *from* (*to 0*)

string-reverse-search-char searches through *string* in reverse order, starting from the index one less than *from*, which defaults to the length of *string*, and returns the index of the first character which is char-equal to *char*, or nil if none is found. Note that the index returned is from the beginning of the string, although the search starts from the end. If the *to* argument is supplied, it limits the extent of the search.

Example:

```
(string-reverse-search-char #/n "banana") => 4
```

**string-reverse-search-not-char** *char string* &optional *from* (*to 0*)

string-reverse-search-not-char searches through *string* in reverse order, starting from the index one less than *from*, which defaults to the length of *string*, and returns the index of the first character that is *not* char-equal to *char*, or nil if none is found. Note that the index returned is from the beginning of the string, although the search starts from the end. If the *to* argument is supplied, it limits the extent of the search.

Example:

```
(string-reverse-search-not-char #/a "banana") => 4
```

**string-reverse-search** *key string* &optional *from* (*to 0*)

string-reverse-search searches for the string *key* in the string *string*. The search proceeds in reverse order, starting from the index one less than *from*, which defaults to the length of *string*, and returns the index of the first (leftmost) character of the first instance found, or nil if none is found. Note that the index returned is from the beginning of the string, although the search starts from the end. The *from* condition, restated, is that the instance of *key* found is the rightmost one whose rightmost character is before the *from* th character of *string*. If the *to* argument is supplied, it limits the extent of the search.

Example:

```
(string-reverse-search "na" "banana") => 4
```

**string-reverse-search-set** *char-set string* &optional *from* (*to 0*)

string-reverse-search-set searches through *string* in reverse order, starting from the index one less than *from*, which defaults to the length of *string*, and returns the index of the first character which is char-equal to some element of *char-set*, or nil if none is found. Note that the index returned is from the beginning of the string, although the search starts from the end. If the *to* argument is supplied, it limits the extent of the search. *char-set* is a set of characters, which can be represented as a list of characters, a string of characters or a single character.

```
(string-reverse-search-set "ab" "banana") => 5
```

**string-reverse-search-not-set** *char-set string* &optional *from* (*to 0*)

string-reverse-search-not-set searches through *string* in reverse order, starting from the index one less than *from*, which defaults to the length of *string*, and returns the index of the first character which is not char-equal to any element of *char-set*, or nil if none is found. Note that the index returned is from the beginning of the string, although the search starts from the end. If the *to* argument is supplied, it limits the extent of the search. *char-set* is a set of characters, which can be represented as a list of characters, a string of characters or a single character.

```
(string-reverse-search-not-set '(#/a #/n) "banana") => 0
```

**string-subst-char** *new-char old-char string*

Returns a copy of *string* in which all occurrences of *old-char* have been replaced by *new-char*.

**substring-after-char** *char string* &optional *start end area*

Returns a copy of the portion of *string* that follows the next occurrence of *char* after index *start*. The portion copied ends at index *end*. If *char* is not found before *end*, a null string is returned.

The value is consed in area *area*, or in default-cons-area, unless it is a null string. *start* defaults to zero, and *end* to the length of *string*.

See also intern (page 512), which given a string will return "the" symbol with that print name.

## 9.5 I/O to Strings

The special forms in this section allow you to create I/O streams that input from or output to the contents of a string. See section 21.5, page 391 for documentation of I/O streams.

**with-input-from-string** (*var string* [*index*] [*limit*]) *body...* *Special Form*

The form

```
(with-input-from-string (var string)
    body)
```

evaluates the forms in *body* with the variable *var* bound to a stream which reads

characters from the string which is the value of the form *string*. The value of the special form is the value of the last form in its body.

The stream is a function that only works inside the **with-input-from-string** special form, so be careful what you do with it. You cannot use it after control leaves the body, and you cannot nest two **with-input-from-string** special forms and use both streams since the special-variable bindings associated with the streams will conflict.

After *string* you may optionally specify two additional arguments. The first is *index*:

```
(with-input-from-string (var string index)
    body)
```

uses *index* as the starting index into the string, and sets *index* to the index of the first character not read when **with-input-from-string** returns. If the whole string is read, *index* will be set to the length of the string. Since it is updated, *index* may not be a general expression; it must be a variable or a **setf**-able reference. *index* is not updated in the event of an abnormal exit from the body, such as a ***throw**. The value of *index* is not updated until **with-input-from-string** returns, so you can't use its value within the body to see how far the reading has gotten.

Currently, use of the *index* feature prevents multiple values from being returned out of the body.

```
(with-input-from-string (var string index limit)
    body)
```

uses the value of the form *limit*, if the value is not **nil**, in place of the length of the string. If you want to specify a *limit* but not an *index*, write **nil** for *index*.

**with-output-to-string** (*var* [*string*] [*index*]) *body...* *Special Form*

This special form provides a variety of ways to send output to a string through an I/O stream.

```
(with-output-to-string (var)
  body)
```

evaluates the forms in *body* with *var* bound to a stream which saves the characters output to it in a string. The value of the special form is the string.

```
(with-output-to-string (var string)
  body)
```

will append its output to the string which is the value of the form *string*. (This is like the **string-nconc** function; see page 147.) The value returned is the value of the last form in the body, rather than the string. Multiple values are not returned. *string* must have an array-leader; element 0 of the array-leader will be used as the fill-pointer. If *string* is too small to contain all the output, **adjust-array-size** will be used to make it bigger.

```
(with-output-to-string (var string index)
  body)
```

is similar to the above except that *index* is a variable or **setf**-able reference which contains the index of the next character to be stored into. It must be initialized outside the **with-**

output-to-string and will be updated upon normal exit. The value of *index* is not updated until with-output-to-string returns, so you can't use its value within the body to see how far the writing has gotten. The presence of *index* means that *string* is not required to have a fill-pointer; if it does have one it will be updated.

The stream is a "downward closure" simulated with special variables, so be careful what you do with it. You cannot use it after control leaves the body, and you cannot nest two with-output-to-string special forms and use both streams since the special-variable bindings associated with the streams will conflict.

It is OK to use a with-input-from-string and with-output-to-string nested within one another, so long as there is only one of each.

Another way of doing output to a string is to use the format facility (see page 411).

## 9.6 Maclisp-Compatible Functions

The following functions are provided primarily for Maclisp compatibility.

**alphalessp** *string1 string2*

(alphalessp *string1 string2*) is equivalent to (string-lessp *string1 string2*).

**getchar** *string index*

Returns the *index* th character of *string* as a symbol. Note that 1-origin indexing is used. This function is mainly for Maclisp compatibility; aref should be used to index into strings (but aref will not coerce symbols or numbers into strings).

**getcharn** *string index*

Returns the *index* th character of *string* as a fixnum. Note that 1-origin indexing is used. This function is mainly for Maclisp compatibility; aref should be used to index into strings (but aref will not coerce symbols or numbers into strings).

**ascii** *x*

ascii is like character, but returns a symbol whose printname is the character instead of returning a fixnum.

Examples:

```
(ascii 101) => A
(ascii 56) => /.
```

The symbol returned is interned in the current package (see chapter 24, page 506).

**maknam** *char-list*

maknam returns an uninterned symbol whose print-name is a string made up of the characters in *char-list*.

Example:

```
(maknam '(a b #/0 d)) => ab0d
```

**implode** *char-list*

implode is like maknam except that the returned symbol is interned in the current package.

The samepnamep function is also provided; see page 100.