10. Characters and Strings

A string is a one-dimensional 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.

Individual characters can be represented by character objects or by fixnums. A character object is actually the same as a fixnum except that it has a recognizably different data type and prints differently. Without escaping, a character object is printed by outputting the character it represents. With escaping, a character object prints as #\char in Common Lisp syntax or as #*\char in traditional syntax; see section 10.1.1, page 205 and page 522. By contrast, a fixnum would in all cases print as a sequence of digits. Character objects are accepted by most numeric functions in place of fixnums, and may be used as array indices. When evaluated, they are constants.

The character object data type was introduced recently for Common Lisp support. Traditionally characters were always represented as fixnums, and nearly all system and user code still does so. Character objects are interchangeable with fixnums in most contexts, but not in eq, which is often used to compare the result of the stream input operations such as :ti, since that might be nil. Therefore, the stream input operations still return fixnums that represent characters. Aside from this, Common Lisp functions that return a character return a character object, while traditional functions return a fixnum. The fixnum which is the character code representing char can be written as #/char in traditional syntax. This is equivalent to writing the fixnum using digits, but does not require you to know the character code.

Most strings are arrays of type art-string, where each element is stored in eight bits. Only characters with character code less than 256 can be stored in an ordinary string; these characters form the type string-char. 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.

Since strings are arrays, the usual array-referencing function aref is used to extract characters from strings. For example, (aref "frob" 1) returns the representation of lower case r. The first character is at index zero.

Conceptually, the elements of a string are character objects. This is what Common Lisp programs expect to see when they do aref (or char, which on the Lisp Machine is synonymous with aref) on a string. But nearly all Lisp Machine programs are traditional, and expect elements of strings to be fixnums. Therefore, aref of a string actually returns a fixnum. A distinct version of aref exists for Common Lisp programs. It is cli:aref and it does return character objects if given a string. For all other kinds of arrays, aref and cli:aref are equivalent.

(aref "Foo" 1) => #0157
(cli:aref "Foo" 1) => #*\0

It is also legal to store into strings, for example using setf of aref. As with rplaca on lists, this changes the actual object; you must be careful to understand where side-effects will propagate. It makes no difference whether a character object or a fixnum is stored. When you
are making strings that you intend to change later, you probably want to create an array with a fill-pointer (see page 166) 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 is used as the length.

The functions described in this section provide a variety of useful operations on strings. In place of a string, most of these functions accept a symbol or a fixnum as an argument, coercing it into a string. Given a symbol, its print name, which is a string, is used. Given a fixnum, a one-character string containing the character designated by that fixnum is 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.

The generic sequence functions in chapter 9 may also be used on strings.

10.1 Characters

The Lisp Machine data type for character objects is a recent addition to the system. Most programs still use fixnums to represent characters.

Common Lisp programs typically work with actual character objects but programs traditionally use fixnums to represent characters. The new Common Lisp functions for operating with characters have been implemented to accept fixnums as well, so that they can be used equally well from traditional programs.

`characterp object`
\[ \text{t if object is a character object; nil otherwise. In particular, it is nil if object is a fixnum such as traditional programs use to represent characters.} \]

`character object`
Coerces object to a single character, represented as a fixnum. If object is a number, it is returned. If object is a string or an array, its first element is returned. If object 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 10.1.1, page 205.

`cl:character object`
Coerces object into a character and returns the character as a character object for Common Lisp programs.

`int-char fixnum`
Converts fixnum, regarded as representing a character, to a character object. This is a special case of `cl:character`. `(int-char #\010)` is the character object for A. If a character object is given as an argument, it is returned unchanged.
char-int char

Converts char, a character object, to the fixnum which represents the same character.
This is the inverse of int-char. It may also be given a fixnum as argument, in which
case the value is the same fixnum.

10.1.1 Components of a Character

A character object, or a fixnum which is interpreted as a character, contains three separate
pieces of information: the character code, the font number, and the modifier bits. Each of these
things is an integer from a fixed range. The character code ranges from 0 to 377 (octal), the font
number from 0 to 377 (octal), and the modifier bits from 0 to 17 (octal). These numeric
constants should not appear in programs; instead, use the constant symbols char-code-limit, and
so on, described below.

Ordinary strings can hold only characters whose font number and modifier bits are zero. Fat
strings can hold characters with any font number, but the modifier bits must still be zero.

Character codes less than 200 octal are printing graphics; when output to a device they are
assumed to print a character and move the cursor one character position to the right. (All
software provides for variable-width fonts, so the term “character position” shouldn’t be taken too
literally.)

Character codes 200 through 236 octal are used for special characters. Character 200 is a
“null character”, which does not correspond to any key on the keyboard. The null character is
not used for anything much: fastload uses it internally. Characters 201 through 236 correspond to
the special function keys on the keyboard such as Return and Call. The remaining character
codes 237 through 377 octal are reserved for future expansion.

Most of the special characters do not normally appear in files (although it is not forbidden for
files to contain them). These characters exist mainly to be used as “commands” from the
keyboard. A few special characters, however, are “format effectors” which are just as legitimate
as printing characters in text files. The names and meanings of these characters are:

Return The “newline” character, which separates lines of text. We do not use the PDP-
10 convention which separates lines by a pair of characters, a “carriage return”
and a “linefeed”.

Page The “page separator” character, which separates pages of text.

Tab The “tabulation” character, which spaces to the right until the next “tab stop”.
Tab stops are normally every 8 character positions.

The space character is considered to be a printing character whose printed image happens to
be blank, rather than a format effector.

When a letter is typed with any of the modifier bit keys (Control, Meta, Super, or Hyper),
the letter is normally upper-case. If the Shift key is pressed as well, then the letter becomes
lower-case. This is exactly the reverse of what the Shift key does to letters without control bits.
(The Shift-lock key has no effect on letters with control bits.)
char-code char
char-font char
char-bits char

Return the character code of char, the font number of char, and the modifier bits value of char. char may be a fixnum or a character object; the value is always a fixnum.

These used to be written as

(ldb \%ch-char char)
(ldb \%ch-font char)
(ldb \%ch-control-meta char)

Such use of ldb is frequent but obsolete.

char-code-limit Constant

A constant whose value is a bound on the maximum code of any character. In the Lisp Machine, currently, it is 400 (octal).

char-font-limit Constant

A constant whose value is a bound on the maximum font number value of any character. In the Lisp Machine, currently, it is 400 (octal).

char-bits-limit Constant

A constant whose value is a bound on the maximum modifier bits value of any character. In the Lisp Machine, currently, it is 20 (octal). Thus, there are four modifier bits. These are just the familiar Control, Meta, Super and Hyper bits.

char-control-bit Constant
char-meta-bit Constant
char-super-bit Constant
char-hyper-bit Constant

Constants with values 1, 2, 4 and 8. These give the meanings of the bits within the bits-field of a character object. Thus, (bit-test char-meta-bit (char-bits char)) would be non-nil if char is a meta-character. (This can also be tested with char-bit.)

char-bit char name

t if char has the modifier bit named by name. name is one of the following four symbols: :control, :meta, :super, and :hyper.

(char-bit #\meta-x :meta) => t.

set-char-bit char name newvalue

Returns a character like char except that the bit specified by name is present if newvalue is non-nil, absent otherwise. Thus,

(set-char-bit #\x :meta t) => #\meta-x.

The value is a fixnum if char is one; a character object if char is one.

Until recently the only way to access the character code, font and modifier bits was with ldb, using the byte field names listed below. Most code still uses that method, but it is obsolete; char-bit should be used instead.

%%kbd-char
%%ch-char Specifies the byte containing the character code.
%ch-font  Specifies the byte containing the font number.
%kbd-control  Specifies the byte containing the Control bit.
%kbd-meta  Specifies the byte containing the Meta bit.
%kbd-sup  Specifies the byte containing the Super bit.
%kbd-hyper  Specifies the byte containing the Hyper bit.
%kbd-control-meta  Specifies the byte containing all the modifier bits.

Characters are sometimes used to represent mouse clicks. The character says which button was pressed and how many times. Refer to the Window System manual for an explanation of how these characters are generated.

tv:kbd-mouse-p char
  t if char is a character used to represent a mouse click. Such characters are always distinguishable from characters that represent keyboard input.

%kbd-mouse-button  Constant
  The value of %kbd-mouse-button is a byte specifier for the field in a mouse signal that says which button was clicked. The byte contains 0, 1, or 2 for the left, middle, or right button, respectively.

%kbd-mouse-n-clicks  Constant
  The value of %kbd-mouse-n-clicks is a byte specifier for the field in a mouse signal that says how many times the button was clicked. The byte contains one less than the number of times the button was clicked.

10.1.2 Constructing Character Objects

code-char code &optional (bits 0) (font 0)
make-char code &optional (bits 0) (font 0)

Returns a character object made from code, bits and font. Common Lisp says that not all combinations may be valid, and that nil is returned for an invalid combination. On the Lisp Machine, any combination is valid if the arguments are valid individually.

According to Common Lisp, code-char requires a number as a first argument, whereas make-char requires a character object, whose character code is used. On the Lisp Machine, either function may be used in either way.

digit-char weight &optional (radix 10.) (font 0)

Returns a character object which is the digit with the specified weight, and with font as specified. However, if there is no suitable character which has weight weight in the specified radix, the value is nil. If the "digit" is a letter (which happens if weight is greater than 9), it is returned in upper case.
tv:make-mouse-char  button  n-clicks

Returns the fixnum character code that represents a mouse click in the standard way. tv:mouse-char-p of this value is t.  button is 0 for the left button, 1 for the middle button, or 2 for the right button.  n-clicks is one less than the number of clicks (1 for a double click, 0 normally).

10.1.3 The Character Set

Here are the numerical values of the characters in the Zetalisp character set. It should never be necessary for a user or a source program to know these values. Indeed, they are likely to be changed in the future. There are symbolic names for all characters: see the section on character names, below.

It is worth pointing out that the Zetalisp character set is different from the ASCII character set. File servers operating on hosts that use ASCII for storing text files automatically perform character set conversion when text files are read or written. The details of the mapping are explained in section 25.8, page 607.
<table>
<thead>
<tr>
<th>Oct Code</th>
<th>Character</th>
<th>Decimal Value</th>
<th>Hex Value</th>
<th>ASCII Value</th>
<th>Unicode Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>center-dot (·)</td>
<td>040</td>
<td>100 @</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>down arrow (↓)</td>
<td>041 !</td>
<td>101 A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>002</td>
<td>alpha (α)</td>
<td>042 &quot;</td>
<td>102 B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>003</td>
<td>beta (β)</td>
<td>043 #</td>
<td>103 C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>004</td>
<td>and-sign (∧)</td>
<td>044 $</td>
<td>104 D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>005</td>
<td>not-sign (¬)</td>
<td>045 %</td>
<td>105 E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>006</td>
<td>epsilon (ε)</td>
<td>046 &amp;</td>
<td>106 F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>007</td>
<td>pi (π)</td>
<td>047 '</td>
<td>107 G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>lambda (λ)</td>
<td>050 (</td>
<td>110 H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>gamma (γ)</td>
<td>051 )</td>
<td>111 I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>012</td>
<td>delta (δ)</td>
<td>052 *</td>
<td>112 J</td>
<td></td>
<td></td>
</tr>
<tr>
<td>013</td>
<td>uparrow (↑)</td>
<td>053 +</td>
<td>113 K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>014</td>
<td>plus-minus (±)</td>
<td>054 ,</td>
<td>114 L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>015</td>
<td>circle-plus (⊕)</td>
<td>055 -</td>
<td>115 M</td>
<td></td>
<td></td>
</tr>
<tr>
<td>016</td>
<td>infinity (∞)</td>
<td>056 .</td>
<td>116 N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>017</td>
<td>partial delta (∂)</td>
<td>057 /</td>
<td>117 O</td>
<td></td>
<td></td>
</tr>
<tr>
<td>020</td>
<td>left horseshoe (⟨)</td>
<td>060 0</td>
<td>120 P</td>
<td></td>
<td></td>
</tr>
<tr>
<td>021</td>
<td>right horseshoe (⟩)</td>
<td>061 1</td>
<td>121 Q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>022</td>
<td>up horseshoe (↑)</td>
<td>062 2</td>
<td>122 R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>023</td>
<td>down horseshoe (↓)</td>
<td>063 3</td>
<td>123 S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>024</td>
<td>universal quantifier (∀)</td>
<td>064 4</td>
<td>124 T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>025</td>
<td>existential quantifier (∃)</td>
<td>065 5</td>
<td>125 U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>026</td>
<td>circle-X (●)</td>
<td>066 6</td>
<td>126 V</td>
<td></td>
<td></td>
</tr>
<tr>
<td>027</td>
<td>double-arrow (→)</td>
<td>067 7</td>
<td>127 W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>030</td>
<td>left arrow (←)</td>
<td>070 8</td>
<td>130 X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>031</td>
<td>right arrow (→)</td>
<td>071 9</td>
<td>131 Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>032</td>
<td>not-equals (≠)</td>
<td>072 :</td>
<td>132 Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>033</td>
<td>diamond (altmode) (*)</td>
<td>073 ;</td>
<td>133 [</td>
<td></td>
<td></td>
</tr>
<tr>
<td>034</td>
<td>less-or-equal (≤)</td>
<td>074 &lt;</td>
<td>134 \</td>
<td></td>
<td></td>
</tr>
<tr>
<td>035</td>
<td>greater-or-equal (≥)</td>
<td>075 =</td>
<td>135 ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>036</td>
<td>equivalence (≡)</td>
<td>076 &gt;</td>
<td>136 ^</td>
<td></td>
<td></td>
</tr>
<tr>
<td>037</td>
<td>or (v)</td>
<td>077 ?</td>
<td>137 -</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>Null character</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>201</td>
<td>Break</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>202</td>
<td>Clear</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>203</td>
<td>Call</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>204</td>
<td>Terminal escape</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>205</td>
<td>Macro/backnext</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>206</td>
<td>Help</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>207</td>
<td>Rubout</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

237-377 reserved for the future

The Lisp Machine Character Set
(all numbers in octal)
10.1.4 Classifying Characters

`string-char-p char`
- `t` if `char` is a character that can be stored in a string. On the Lisp Machine, this is true if the font and modifier bits of `char` are zero.

`standard-char-p char`
- `t` if `char` is a standard Common Lisp character: any of the 95 ASCII printing characters (including Space), and the Return character. Thus `(standard-char-p #\end)` is `nil`.

`graphic-char-p char`
- `t` if `char` is a graphic character: one which has a printed shape. A, -, Space and © are all graphic characters; Return, End and Abort are not. A character whose modifier bits are nonzero is never graphic.

Ordinary output to windows prints graphic characters using the current font. Nongraphic characters are printed using lozenges unless they have special formatting meanings (as Return does).

`alpha-char-p char`
- `t` if `char` is a letter with zero modifier bits.

`digit-char-p char &optional (radix 10.)`
- If `char` is a digit available in the specified radix, returns the weight of that digit. Otherwise, it returns `nil`. If the modifier bits of `char` are nonzero, the value is always `nil`. (It would be more useful to ignore the modifier bits, but this decision provides Common Lisp with a foolish consistency.) Examples:
  - `(digit-char-p #\8 8) => nil`
  - `(digit-char-p #\8 9) => 8`
  - `(digit-char-p #\F 16.) => 15.`
  - `(digit-char-p #\c-8 anything) => nil`

`alphanumericp char`
- `t` if `char` is a letter or a digit 0 through 9, with zero modifier bits.

10.1.5 Comparing Characters

`char-equal &rest chars`
- This is the primitive for comparing characters for equality; many of the string functions call it. The arguments may be fixnums or character objects indiscriminately. The result is `t` if the characters are equal ignoring case, font and modifier bits, otherwise `nil`.

`char-not-equal &rest chars`
- `t` if the arguments are all different as characters, ignoring case, font and modifier bits.
char-lessp &rest chars
char-greaterp &rest chars
char-not-lessp &rest chars
char-not-greaterp &rest chars

Ordered comparison of characters, ignoring case, font and modifier bits. These are the primitives for comparing characters for order; many of the string functions call it. The arguments may be fixnums or character objects. The result is t if the arguments are in strictly increasing (strictly decreasing, nonincreasing, nondecreasing) order. Details of the ordering of characters are in section 10.1.1, page 205.

char= char1 &rest chars
char//= char1 &rest chars
char> char1 &rest chars
char< char1 &rest chars
char>= char1 &rest chars
char<= char1 &rest chars

These are the Common Lisp functions for comparing characters and including the case, font and bits in the comparison. On the Lisp Machine they are synonyms for the numeric comparison functions =, >, etc. Note that in Common Lisp syntax you would write char//=, not char//=.

10.1.6 Character Names

Characters can sometimes be referred to by long names; as, for example, in the #\ construct in Lisp programs. Every basic character (zero modifier bits) which is not a graphic character has one or more standard names. Some graphic characters have standard names too. When a non-graphic character is output to a window, it appears as a lozenge containing the character's standard name.

char-name char

Returns the standard name (or one of the standard names) of char, or nil if there is none. The name is returned as a string. (char-name #\space) is the string "SPACE".

If char has nonzero modifier bits, the value is nil. Compound names such as Control-X are not constructed by this function.

name-char name

Returns (as a character object) the character for which name is a name, or returns nil if name is not a recognized character name. name may be a symbol or a string. Compound names such as Control-X are not recognized.

read uses this function to process the #\ construct when a character name is encountered.

The following are the recognized special character names, in alphabetical order except with synonyms together. Character names are encoded and decoded by the functions char-name and name-char (page 211).
First a list of the special function keys.

| abort     | break   | call     | clear-input, clear |
| delete    | end     | hand-down| hand-left          |
| hand-right| hand-up | help     | hold-output        |
| line, if  | macro, back-next | network |                  |
| overstrike, backspace, bs | resume | page, form, clear-screen |                  |
| quote     | roman-i | return, cr|                  |
| roman-ii  | roman-iii | roman-iv |                  |
| rubout    | space, sp | status  |                  |
| system    | tab     | terminal, esc |                  |

These are printing characters that also have special names because they may be hard to type on the hosts that are used as file servers.

| altmode      | circle-plus | delta | gamma |
| integral     | lambda      | plus-minus | uparrow |
| center-dot   | down-arrow  | alpha | beta  |
| and-sign     | not-sign    | epsilon | pi    |
| lambda       | gamma       | delta | up-arrow |
| plus-minus   | circle-plus | infinity | partial-delta |
| left-horseshoe | right-horseshoe | up-horseshoe | down-horseshoe |
| universal-quantifier | double-arrow | left-arrow | right-arrow |
| circle-x     | not-equal | equivalence | or-sign |

The following names are for special characters sometimes used to represent single and double mouse clicks. The buttons can be called either l, m, r or 1, 2, 3 depending on stylistic preference.

| mouse-l-1 or mouse-1-1 | mouse-l-2 or mouse-1-2 |
| mouse-m-1 or mouse-2-1 | mouse-m-2 or mouse-2-2 |
| mouse-r-1 or mouse-3-1 | mouse-r-2 or mouse-3-2 |

10.2 Conversion to Upper or Lower Case

\textbf{upper-case-p \texttt{char}}

\texttt{t} if \texttt{char} is an upper case letter with zero modifier bits.

\textbf{lower-case-p \texttt{char}}

\texttt{t} if \texttt{char} is an lower case letter with zero modifier bits.

\textbf{both-case-p \texttt{char}}

This Common Lisp function is defined to return \texttt{t} if \texttt{char} is a character which has distinct upper and lower case forms. On the Lisp Machine it returns \texttt{t} if \texttt{char} is a letter with zero modifier bits.
char-upcase char
If char, is a lower-case alphabetic character its upper-case form is returned; otherwise, char itself is returned. If font information or modifier bits are present, they are preserved. If char is a fixnum, the value is a fixnum. If char is a character object, the value is a character object.

cchar-downcase char
Similar, but converts to lower case.

string-upcase string &key (start 0) end
Returns a string like string, with all lower-case alphabetic characters replaced by the corresponding upper-case characters. If start or end is specified, only the specified portion of the string is converted, but in any case the entire string is returned.

The result is a copy of string unless no change is necessary. string itself is never modified.

string-downcase string &key (start 0) end
Similar, but converts to lower case.

string-capitalize string &key (start 0) end
Returns a string like string in which all, or the specified portion, has been processed by capitalizing each word. For this function, a word is any maximal sequence of letters or digits. It is capitalized by putting the first character (if it is a letter) in upper case and any letters in the rest of the word in lower case.

The result is a copy of string unless no change is necessary. string itself is never modified.

nstring-upcase string &key (start 0) end
nstring-downcase string &key (start 0) end
nstring-capitalize string &key (start 0) end
Like the previous functions except that they modify string itself and return it.

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.

This function is somewhat obsolete. One can use string-capitalize followed optionally by string-subst-char.

See also the format operation ~(...~) on page 488.
10.3 Basic String Operations

\texttt{make-string} size &key \texttt{(initial-element 0)}  
Creates and returns a string of length \textit{size}, with each element initialized to \textit{initial-element},  
which may be a fixnum or a character.

\texttt{string} x  
Coerces \textit{x} into a string. Most of the string functions apply this to their string arguments.  
If \textit{x} is a string (or any array), it is returned. If \textit{x} is a symbol, its p-name is returned. If \textit{x} is a non-negative fixnum less than 400 octal, a one-character-long string containing it is created and returned. If \textit{x} is an instance that supports the \texttt{string-for-printing} operation (such as, a pathname) then the result of that operation is returned. Otherwise, an error is signaled.

If you want to get the printed representation of an object into the form of a string, this function is \textit{not} what you should use. You can use \texttt{format}, passing a first argument of \texttt{nil} (see page 483). You might also want to use \texttt{with-output-to-string} (see page 474).

\texttt{string-length} string  
Returns the number of characters in \textit{string}. This is 1 if \textit{string} is a number or character object, the \texttt{array-active-length} (see page 174) if \textit{string} is an array, or the \texttt{array-active-length} of the p-name if \textit{string} is a symbol.

\texttt{string-equal} string1 string2 &key \texttt{(start1 0) (start2 0) end1 end2}  
Compares two strings, returning \texttt{t} if they are equal and \texttt{nil} if they are not. The comparison ignores the font and case of the characters. \texttt{equal} calls \texttt{string-equal} if applied to two strings.

The keyword arguments \textit{start1} and \textit{start2} are the starting indices into the strings. \textit{end1} and \textit{end2} are the final indices; the comparison stops just \textit{before} the final index. \texttt{nil} for \textit{end1} or \textit{end2} means stop at the end of the string.

Examples:

\begin{verbatim}
(string-equal "Foo" "foo") => t
(string-equal "foo" "bar") => nil
(string-equal "element" "select" 0 1 3 4) => t
\end{verbatim}

An older calling sequence in which the \textit{start} and \textit{end} arguments are positional rather than keyword is still supported. The arguments come in the order \textit{start1} \textit{start2} \textit{end1} \textit{end2}. This calling sequence is obsolete and should be changed whenever found.

\texttt{string-not-equal} string1 string2 &key \texttt{(start1 0) (start2 0) end1 (end2 end2)}  
\texttt{(not (string-equal ...))}

\texttt{string=} string1 string2 &key \texttt{(start1 0) (start2 0) end1 end2}  
is like \texttt{string-equal} except that case is significant.

\begin{verbatim}
(string= "A" "a") => nil
\end{verbatim}
string <= string1 string2 &key (start1 0) end1 (start2 0) end2
string //= string1 string2 &key (start1 0) end1 (start2 0) end2
        (not (string= ...)). Note that in Common Lisp syntax you would write string/=, not
        string//=.

string<lessp string1 string2 &key (start1 0) end1 (start2 0) end2
string>greaterp string1 string2 &key (start1 0) end1 (start2 0) end2
string<not-greaterp string1 string2 &key (start1 0) end1 (start2 0) end2
string<not-lessp string1 string2 &key (start1 0) end1 (start2 0) end2
        Compare all or the specified portions of string1 and string2 using dictionary order.
        Characters are compared using char-lessp and char-equal so that font and alphabetic
        case are ignored.

You can use these functions as predicates, but they do more. If the strings fit the
        condition (e.g. string1 is strictly less in string-lessp) then the value is a number, the
        index in string1 of the first point of difference between the strings. This equals the length
        of string1 if the strings match. If the condition is not met, the value is nil.
        (string-lessp "aa" "Ab") => 1
        (string-lessp "aa" "Ab" :end1 1 :end2 1) => nil
        (string-not-greaterp "Aa" "Ab" :end1 1 :end2 1) => 1

string< string1 string2 &key (start1 0) end1 (start2 0) end2
string> string1 string2 &key (start1 0) end1 (start2 0) end2
string>= string1 string2 &key (start1 0) end1 (start2 0) end2
string<= string1 string2 &key (start1 0) end1 (start2 0) end2
string> string1 string2 &key (start1 0) end1 (start2 0) end2
string<= string1 string2 &key (start1 0) end1 (start2 0) end2
        Like string-lessp, etc., but treat case and font as significant when comparing characters.
        (string< "AA" "aa") => 0
        (string-lessp "AA" "aa") => nil

string-compare string1 string2 &optional (start1 0) (start2 0) end1 end2
        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
        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"
нибудь string start &optional end area
Is like substring except that the substring is not copied; instead an indirect array (see page 167) is created which shares part of the argument string. Modifying one string will modify the other.

Note that nsub_string does not necessarily use less storage than substring: an nsub_string of any length uses at least as much storage as a substring 12 characters long. So you shouldn’t use this 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
Copies and concatenates any number of strings 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, vectors of any type may be used as arguments, and the value is a vector capable of holding all the elements of all the arguments. Thus string-append can be used to copy and concatenate any type of vector. If the first argument is not an array (for example, if it is a character), the value is a string.
Example:

\[ (\text{string-append}\#\!\!"foo"\#\!\!) \Rightarrow \"I\!fool\!" \]

string-nconc modified-string &rest strings
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 178). 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 176). 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
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:

\[ (\text{string-trim}\,(\#\!\!sp)\,"Dr. No") \Rightarrow \"Dr. No\!" \]
\[ (\text{string-trim}\,\"ab\,"abbafooabb") \Rightarrow \"foo\!" \]

string-left-trim char-set string
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
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
string-nreverse string
Like reverse and nreverse, but on strings only (see page 190). There is no longer any reason to use these functions except that they coerce numbers and symbols into strings like the other string functions.

string-pluralize string
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 "trufan") => "trufen"
(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.

string-select-a-or-an word
Returns "a" or "an" according to the string word; whichever one appears to be correct to use before word in English.

string-append-a-or-an word
Returns the result of appending "a " or "an ", whichever is appropriate, to the front of word.

%string-equal string1 start1 string2 start2 count
%string-equal is the microcode primitive used by string-equal. It returns t if the count characters of string1 starting at start1 are char-equal to the count characters of string2 starting at start2, 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 start1 to (string-length string1) against the substring from start2 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)
```

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

*Variable*

If this variable is `t`, the functions `%string-equal` and `%string-search` consider case (and font) significant in comparing characters. Normally this variable is `nil` and those primitives ignore differences of case.

This variable may be bound by user programs around calls to `%string-equal` and `%string-search-char`, but do not set it globally, for that may cause system malfunctions.

### 10.4 String Searching

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

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 `to` is non-`nil`, it is used in place of `(string-length string)` to limit the extent of the search.

Example:

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

Case (and font) is significant in comparison of characters if `consider-case` is non-`nil`. In other words, characters are compared using `char=` rather than `char-equal`.

```
(string-search-char \A "BAanana" 0 nil t) => 3
```

**%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. The arguments are all required. Case-sensitivity is controlled by the value of the variable `alphabetic-case-affects-string-comparison` rather than by an argument. Except for these these differences, `%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 consider-case`

Like `string-search-char` but searches `string` for a character different from `char`.

Example:

```
(string-search-not-char \B "banana") => 1
```

```
(string-search-not-char \b "banana" 0 nil t) => 0
```

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

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 `to` is non-nil, it is used in place of `(string-length string)` to limit the extent of the search.
The arguments key-from and key-to can be used to specify the portion of key to be searched for, rather than all of key.

Case and font are significant in character comparison if consider-case is non-nil.

Example:

```
(string-search "an" "banana") => 1
(string-search "an" "banana" 2) => 3
(string-search "tank" "banana" 2 nil 1 3) => 3
(string-search "an" "BAnaNA" 0 nil 0 nil t) => nil
```

```
string-search-set char-set string &optional (from0) to consider-case
```

Searches through string looking for a character that is in char-set. char-set is a set of characters, which can be represented as a sequence of characters or a single character.

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 to is non-nil, it is used in place of (string-length string) to limit the extent of the search.

Case and font are significant in character comparison if consider-case is non-nil.

Example:

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

```
string-search-not-set char-set string &optional (from0) to consider-case
```

Like string-search-set but searches for a character that is not in char-set.

Example:

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

```
string-reverse-search-char char string &optional from (to0) consider-case
```

Searches through string in reverse order, starting from the index one less than from (nil for from starts at the end 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. The last (leftmost) character of string examined is the one at index to.

Case and font are significant in character comparison if consider-case is non-nil. In this case, char= is used for the comparison rather than char-equal.

Example:

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

```
string-reverse-search-not-char char string &optional from (to0) consider-case
```

Like string-reverse-search-char but searches for a character in string that is different from char.

Example:

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

; 4 is the index of the second "n"
string-reverse-search  key string &optional from (to 0) (key-from 0) key-to consider-case
Searches for the string key in the string string. The search proceeds in reverse order, starting from the index one less than from, 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. nil for from means the search starts at the end of string. The last (leftmost) character of string examined is the one at index to.

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

The arguments key-from and key-to can be used to specify the portion of key to be searched for, rather than all of key. Case and font are significant in character comparison if consider-case is non-nil.

string-reverse-search-set  char-set string &optional from (to 0) consider-case
Searches through string in reverse order for a character which is char-equal to some element of char-set. char-set is a set of characters, which can be represented as a list of characters, a string of characters or a single character.

The search starts from an index one less than from, and returns the index of the first suitable character found, or nil if none is found. nil for from means the search starts at the end of string. Note that the index returned is from the beginning of the string, although the search starts from the end. The last (leftmost) character of string examined is the one at index to.

Case and font are significant in character comparison if consider-case is non-nil. In this case, char= is used for the comparison rather than char-equal.

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

string-reverse-search-not-set  char-set string &optional from (to 0) consider-case
Like string-reverse-search-set but searches for a character which is not in char-set.

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

string-subst-char  new-char old-char string (copy-p) (retain-font-p)
Returns a copy of string in which all occurrences of old-char have been replaced by new-char.

Case and font are ignored in comparing old-char against characters of string. Normally the font information of the character replaced is preserved, so that an old-char in font 3 is replaced by a new-char in font 3. If retain-font-p is nil, the font specified in new-char is stored whenever a character is replaced.

If copy-p is nil, string is modified destructively and returned. No copy is made.
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 make-symbol (page 133), which given a string makes a new uninterned symbol with
that print name, and intern (page 645), which given a string returns the one and only symbol (in
the current package) with that print name.

10.5 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).

samepnamep sym1 sym2
    This predicate is equivalent to string=.

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 does 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 does not coerce symbols or numbers into strings).

ascii x
    Like character, but returns a symbol whose printname is the character instead of
    returning a fixnum.
    Examples:
    (ascii #\010) => A
    (ascii #\066) => /.
    The symbol returned is interned in the current package (see chapter 27, page 636).

maknam char-list
    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.