4. STRINGS

A string represents a sequence of characters. Interlisp strings are a subtype of Common Lisp strings. Medley provides functions for creating strings, concatenating strings, and creating sub-strings of a string; all accepting or producing Common Lisp-acceptable strings.

A string is typed as a double quote (`"`), followed by a sequence of any characters except double quote and `%`, terminated by a double quote. To include `%` or `"` in a string, type `%` in front of them:

```
"A string"
"A string with %" in it, and a %%."
""      ; an empty string
```

Strings are printed by `PRINT` and `PRIN2` with initial and final double quotes, and `%`s inserted where necessary for it to read back in properly. Strings are printed by `PRIN1` without the double quotes and extra `%`s. The null string is printed by `PRINT` and `PRIN2` as `""`. `(PRIN1 "")` doesn't print anything.

Internally, a string is stored in two parts: a "string header" and the sequence of characters. Several string headers may refer to the the same character sequence, so a substring can be made by creating a new string header, without copying any characters. Functions that refer to "strings" actually manipulate string headers. Some functions take an "old string" argument, and re-use the string pointer.

(**STRINGP** *X*) [Function]

Returns *X* if *X* is a string, `NIL` otherwise.

(**STREQUAL** *X Y*) [Function]

Returns `T` if *X* and *Y* are both strings and they contain the same sequence of characters, otherwise `NIL`. `EQUAL` uses `STREQUAL`. Note that strings may be `STREQUAL` without being `EQ`. For instance,

```
(STREQUAL "ABC" "ABC")  =>  T
(EQ "ABC" "ABC")  =>  NIL
```

`STREQUAL` returns `T` if *X* and *Y* are the same string pointer, or two different string pointers which point to the same character sequence, or two string pointers which point to different character sequences which contain the same characters. Only in the first case would *X* and *Y* be `EQ`.

(**STRING-EQUAL** *X Y*) [Function]

Returns `T` if *X* and *Y* are either strings or symbols, and they contain the same sequence of characters, ignoring case. For instance,

```
(STRING-EQUAL "FOO" "Foo")  =>  T
(STRING-EQUAL "FOO" 'Foo)  =>  T
```

This is useful for comparing things that might want to be considered "equal" even though they're not both symbols in a consistent case, such as file names and user names.

(**STRING.EQUAL** *X Y*) [Function]

Returns `T` if the print names of *X* and *Y* contain the same sequence of characters, ignoring

case. For instance,

```
(STRING-EQUAL "320" 320)  =>  T
(STRING-EQUAL "FOO" 'Foo)  =>  T
```

This is like STRING-EQUAL, but handles numbers, etc., where STRING-EQUAL doesn't.

(**ALLOCSTRING** *N INITCHAR OLD FATFLG*) [Function]

Creates a string of length *N* characters of *INITCHAR* (which can be either a character code or something coercible to a character). If *INITCHAR* is NIL, it defaults to character code 0. if *OLD* is supplied, it must be a string pointer, which is modified and returned.

If *FATFLG* is non-NIL, the string is allocated using full 16-bit NS characters (see Chapter 2) instead of 8-bit characters. This can speed up some string operations if NS characters are later inserted into the string. This has no other effect on the operation of the string functions.

(**MKSTRING** *X FLG RDTBL*) [Function]

If *X* is a string, returns *X*. Otherwise, creates and returns a string containing the print name of *X*. Examples:

```
(MKSTRING "ABC")  =>  "ABC"
(MKSTRING '(A B C))  =>  "(A B C)"
(MKSTRING NIL)  =>  "NIL"
```

Note that the last example returns the string "NIL", not the symbol NIL.

If *FLG* is T, then the PRIN2-name of *X* is used, computed with respect to the readtable *RDTBL*. For example,

```
(MKSTRING "ABC" T)  =>  "%"ABC%""
```

(**NCHARS** *X FLG RDTBL*) [Function]

Returns the number of characters in the print name of *X*. If *FLG*=T, the PRIN2-name is used. For example,

```
(NCHARS 'ABC)  =>  3
(NCHARS "ABC" T)  =>  5
```

Note: NCHARS works most efficiently on symbols and strings, but can be given any object.

(**SUBSTRING** *X N M OLDPTR*) [Function]

Returns the substring of *X* consisting of the *N*th through *M*th characters of *X*. If *M* is NIL, the substring contains the *N*th character thru the end of *X*. *N* and *M* can be negative numbers, which are interpreted as counts back from the end of the string, as with NTHCHAR (Chapter 2). SUBSTRING returns NIL if the substring is not well defined, (e.g., *N* or *M* specify character positions outside of *X*, or *N* corresponds to a character in *X* to the right of the character indicated by *M*). Examples:

```
(SUBSTRING "ABCDEFG" 4 6)  =>  "DEF"
(SUBSTRING "ABCDEFG" 3 3)  =>  "C"
(SUBSTRING "ABCDEFG" 3 NIL)  =>  "CDEFG"
(SUBSTRING "ABCDEFG" 4 -2)  =>  "DEF"
(SUBSTRING "ABCDEFG" 6 4)  =>  NIL
```

```
(SUBSTRING "ABCDEFG" 4 9)  =>  NIL
```

If *X* is not a string, it is converted to one. For example,

```
(SUBSTRING '(A B C) 4 6)  =>  "B C"
```

SUBSTRING does not actually copy any characters, but simply creates a new string pointer to the characters in *X*. If *OLDPTR* is a string pointer, it is modified and returned.

(**GNC** *X*) [Function]

"Get Next Character." Returns the next character of the string *X* (as a symbol); also removes the character from the string, by changing the string pointer. Returns NIL if *X* is the null string. If *X* isn't a string, a string is made. Used for sequential access to characters of a string. Example:

```
_(SETQ FOO "ABCDEFG")
   "ABCDEFG"
_(GNC FOO)
   A
_(GNC FOO)
   B
_FOO
   "CDEFG"
```

Note that if *A* is a substring of B, (GNC A) does not remove the character from B.

(**GLC** *X*) [Function]

"Get Last Character." Returns the last character of the string *X* (as a symbol); also removes the character from the string. Similar to GNC. Example:

```
_(SETQ FOO "ABCDEFG")
   "ABCDEFG"
_(GLC FOO)
   G
_(GLC FOO)
   F
_FOO
   "ABCDE"
```

(**CONCAT** X_1 X_2 ... X_N) [NoSpread Function]

Returns a new string which is the concatenation of (copies of) its arguments. Any arguments which are not strings are transformed to strings. Examples:

```
(CONCAT "ABC" 'DEF "GHI")  =>  "ABCDEFGHI"
(CONCAT '(A B C) "ABC")  =>  "(A B C)ABC"
```

(CONCAT) returns the null string, ""

(**CONCATLIST** *L*) [Function]

L is a list of strings and/or other objects. The objects are transformed to strings if they aren't strings. Returns a new string which is the concatenation of the strings. Example:

```
(CONCATLIST '(A B (C D) "EF"))  =>  "AB(C D)EF"
```

(**RPLSTRING** *X N Y*) [Function]

Replaces the characters of string *X* beginning at character position *N* with string *Y*. *X* and *Y* are converted to strings if they aren't already. *N* may be positive or negative, as with SUBSTRING. Characters are smashed into (converted) *X*. Returns the string *X*. Examples:

```
(RPLSTRING "ABCDEF" -3 "END")  =>  "ABCEND"
(RPLSTRING "ABCDEFGHIJK" 4 '(A B C))  =>  "ABC(A B C)K"
```

Generates an error if there is not enough room in *X* for *Y*, i.e., the new string would be longer than the original. If *Y* was not a string, *X* will already have been modified since RPLSTRING does not know whether *Y* will "fit" without actually attempting the transfer.

Warning: In some implementations of Interlisp, if *X* is a substring of Z, Z will also be modified by the action of RPLSTRING or RPLCHARCODE. However, this is not guaranteed to be true in all cases, so programmers should not rely on RPLSTRING or RPLCHARCODE altering the characters of any string other than the one directly passed as argument to those functions.

(**RPLCHARCODE** *X N CHAR*) [Function]

Replaces the *N*th character of the string *X* with the character code *CHAR*. *N* may be positive or negative. Returns the new *X*. Similar to RPLSTRING. Example:

```
(RPLCHARCODE "ABCDE" 3 (CHARCODE F))  =>  "ABFDE"
```

(**STRPOS** *PAT STRING START SKIP ANCHOR TAIL CASEARRAY BACKWARDSFLG*) [Function]

STRPOS is a function for searching one string looking for another. *PAT* and *STRING* are both strings (or else they are converted automatically). STRPOS searches *STRING* beginning at character number *START*, (or 1 if *START* is NIL) and looks for a sequence of characters equal to *PAT*. If a match is found, the character position of the first matching character in *STRING* is returned, otherwise NIL. Examples:

```
(STRPOS "ABC" "XYZABCDEF")  =>  4
(STRPOS "ABC" "XYZABCDEF" 5)  =>  NIL
(STRPOS "ABC" "XYZABCDEFABC" 5)  =>  10
```

SKIP can be used to specify a character in *PAT* that matches any character in *STRING*. Examples:

```
(STRPOS "A&C&" "XYZABCDEF" NIL '&)  =>  4
(STRPOS "DEF&" "XYZABCDEF" NIL '&)  =>  NIL
```

If *ANCHOR* is T, STRPOS compares *PAT* with the characters beginning at position *START* (or 1 if *START* is NIL). If that comparison fails, STRPOS returns NIL without searching any further down *STRING*. Thus it can be used to compare one string with some *portion* of another string. Examples:

```
(STRPOS "ABC" "XYZABCDEF" NIL NIL T)  =>  NIL
(STRPOS "ABC" "XYZABCDEF" 4 NIL T)  =>  4
```

If *TAIL* is T, the value returned by STRPOS if successful is not the starting position of the sequence of characters corresponding to *PAT*, but the position of the first character after that, i.e., the starting position plus (NCHARS *PAT*). Examples:

```
(STRPOS "ABC" "XYZABCDEFABC" NIL NIL NIL T)  =>  7
```

```
(STRPOS "A" "A" NIL NIL NIL T)  =>  2
```

If *TAIL* = NIL, STRPOS returns NIL, or a character position within *STRING* which can be passed to SUBSTRING. In particular, (STRPOS "" "") => NIL. However, if *TAIL* = T, STRPOS may return a character position outside of *STRING*. For instance, note that the second example above returns 2, even though "A" has only one character.

If *CASEARRAY* is non-NIL, this should be a casearray like that given to FILEPOS (Chapter 25). The casearray is used to map the string characters before comparing them to the search string.

If *BACKWARDSFLG* is non-NIL, the search is done backwards from the end of the string.

(**STRPOSL** *A STRING START NEG BACKWARDSFLG*) [Function]

STRING is a string (or is converted automatically to a string), *A* is a list of characters or character codes. STRPOSL searches *STRING* beginning at character number *START* (or 1 if *START* = NIL) for one of the characters in *A*. If one is found, STRPOSL returns as its value the corresponding character position, otherwise NIL. Example:

```
(STRPOSL '(A B C) "XYZBCD")  =>  4
```

If *NEG* = T, STRPOSL searches for a character *not* on *A*. Example:

```
(STRPOSL '(A B C) "ABCDEF" NIL T)  =>  4
```

If any element of *A* is a number, it is assumed to be a character code. Otherwise, it is converted to a character code via CHCON1. Therefore, it is more efficient to call STRPOSL with *A* a list of character *codes.*

If *A* is a bit table, it is used to specify the characters (see MAKEBITTABLE below)

If *BACKWARDSFLG* is non-NIL, the search is done backwards from the end of the string.

STRPOSL uses a "bit table" data structure to search efficiently. If *A* is not a bit table, it is converted to a bit table using MAKEBITTABLE. If STRPOSL is to be called frequently with the same list of characters, a considerable savings can be achieved by converting the list to a bit table *once*, and then passing the bit table to STRPOSL as its first argument.

(**MAKEBITTABLE** *L NEG A*) [Function]

Returns a bit table suitable for use by STRPOSL. *L* is a list of characters or character codes, *NEG* is the same as described for STRPOSL. If *A* is a bit table, MAKEBITTABLE modifies and returns it. Otherwise, it will create a new bit table.

Note: If *NEG* = T, STRPOSL must call MAKEBITTABLE whether *A* is a list *or* a bit table. To obtain bit table efficiency with *NEG*=T, MAKEBITTABLE should be called with *NEG*=T, and the resulting "inverted" bit table should be given to STRPOSL with *NEG*=NIL.

[This page intentionally left blank]