A litatom (for "literal atom") is an object that conceptually consists of a print name, a value, a function definition, and a property list. Litatoms are also known as "symbols" in Common Lisp. For clarity, we will use the term "symbol".

A symbol is read as any string of non-delimiting characters that cannot be interpreted as a number. The syntactic characters that delimit symbols are called "separator" or "break" characters (see Chapter 25) and normally are space, end-of-line, line-feed, left parenthesis `(`, right parenthesis `)`, double quote `"`, left square bracket `[`, and right square bracket `]`. However, any character may be included in a symbol by preceding it with the character `%`. Here are some examples of symbols:

```
A wxyz 23SKIDDOO %]
Long% Litatom% With% Embedded% Spaces
```

(**LITATOM** *X*) [Function]

Returns `T` if *X* is a symbol, `NIL` otherwise. Note that a number is not a symbol.

```
(LITATOM NIL) = T
```

(**ATOM** *X*) [Function]

Returns `T` if *X* is an atom (i.e., a symbol or a number) or `NIL` (e.g. `(ATOM NIL) = T`); otherwise returns `NIL`.

**Warning:** `(ATOM X)` is `NIL` if `X` is an array, string, etc. In Common Lisp, the function `CL:ATOM` is defined equivalent to the Interlisp function `NLISTP`.

Each symbol has a print name, a string of characters that uniquely identifies that symbol: Those characters that are output when the symbol is printed using `PRIN1`, e.g., the print name of the symbol `ABC%(D` consists of the five characters `ABC(D`.

Symbols are unique: If two symbols print the same, they will always be `EQ`. Note that this is not true for strings, large integers, floating-point numbers, etc.; they all can print the same without being `EQ`. Thus, if `PACK` or `MKATOM` is given a list of characters corresponding to a symbol that already exists, they return a pointer to that symbol, and do not make a new symbol. Similarly, if the read program is given as input a sequence of characters for which a symbol already exists, it returns a pointer to that symbol.

Symbol names are limited to 255 characters. Attempting to create a larger symbol will cause an error: `Atom too long`.

Sometimes we'll refer to a "`PRIN2`-name". The `PRIN2`-name of a symbol is those characters output when it is printed using `PRIN2`. So the `PRIN2`-name of the symbol `ABC%(D` is the six characters `ABC%(D`. The `PRIN2`-name depends on what readtable is being used (see Chapter 25), since this determines where `%`s will be inserted. Many of the functions below allow either print names or `PRIN2`-names to be used, as specified by `FLG` and `RDTBL` arguments. If `FLG` is `NIL`, print names are used. Otherwise, `PRIN2`-names are used, computed with respect to the readtable `RDTBL` (or the current readtable, if `RDTBL` = `NIL`).

(**MKATOM** *X*) [Function]

Creates and returns a symbol whose print name is the name as that of the string *X* or, if *X* is not a string, the same as that of (MKSTRING X). Examples:

```
(MKATOM '(A B C)) => %(A% B% C%)
(MKATOM "1.5") => 1.5
```

Note that the last example returns a number, not a symbol. It is a deeply-ingrained feature of Interlisp that no symbol can have the print name of a number.

(**SUBATOM** *X N M*) [Function]

Returns a symbol made from the *N*th through *M*th characters of the print name of *X*. If *N* or *M* are negative, they specify positions counting backwards from the end of the print name. Equivalent to (MKATOM (SUBSTRING X N M)). Examples:

```
(SUBATOM "FOO1.5BAR" 4 6) => 1.5
(SUBATOM '(A B C) 2 -2) => A% B% C
```

(**PACK** *X*) [Function]

If *X* is a list of symbols, PACK returns a single symbol whose print name is the concatenation of the print names of the symbols in *X*. If the concatenated print name is the same as that of a number, PACK returns that number. For example:

```
(PACK '(A BC DEF G)) => ABCDEFG
(PACK '(1 3.4)) => 13.4
(PACK '(1 E -2)) => .01
```

Although X is usually a list of symbols, it can be a list of arbitrary objects. The value of PACK is still a single symbol whose print name is the concatenation of the print names of all the elements of *X*, e.g.,

```
(PACK '((A B) "CD")) => %(A% B%)CD
```

If *X* is not a list or NIL, PACK generates the error Illegal arg.

(**PACK*** $X_1$ $X_2$... $X_N$ ) [NoSpread Function]

Version of PACK that takes an arbitrary number of arguments, instead of a list. Examples:

```
(PACK* 'A 'BC 'DEF 'G => ABCDEFG
(PACK* 1 3.4)) => 13.4
```

(**GENSYM** *PREFIX* — — — — ) [Function]

Returns a symbol of the form Xnnnn, where X = *PREFIX* (or A if *PREFIX* is NIL) and nnnn is an integer. Thus, the first one generated is A0001, the second A0002, etc. The integer suffix is always at least four characters long, but it can grow beyond that. For example, the next symbol produced after A9999 would be A10000. GENSYM provides a way of generating symbols for various uses within the system .

Note: The Common Lisp function CL:GENSYM is not the same as Interlisp's GENSYM. Interlisp always creates interned symbols whereas CL:GENSYM creates uninterned symbols.

**GENNUM** [Variable]

The value of GENNUM, initially 0, determines the next GENSYM, e.g., if GENNUM is set to 23, (GENSYM) = A0024.

The term "gensym" is used to indicate a symbol that was produced by the function GENSYM. Symbols generated by GENSYM are the same as any other symbols: they have property lists, and can be given function definitions. The symbols are not guaranteed to be new. For example, if the user has previously created A0012, either by typing it in, or via PACK or GENSYM itself, then if GENNUM is set to 11, the next symbol returned by GENSYM will be the A0012 already in existence.

(**MAPATOMS** *FN*) [Function]

Applies *FN* (a function or lambda expression) to every symbol in the system. Returns NIL. For example:

```
(MAPATOMS (FUNCTION (LAMBDA(X) (if (GETD X) then (PRINTX)]
```

will print every symbol with a function definition.

**Warning**: Be careful if *FN* is a lambda expression or an interpreted function: since NOBIND is a symbol, it will eventually be passed as an argument. The first reference to that argument within the function will signal an error.

A way around this problem is to use a Common Lisp function, so that the Common Lisp interpreter will be invoked. It will treat the argument as local, not special and no error will be signaled. An alternative solution is to include the argument to the Interlisp function in a LOCALVARS declaration and then compile the function before passing it to MAPATOMS. This will significantly speed up MAPATOMS.

(**APROPOS** *STRING ALLFLG QUITFLG OUTPUT*) [Function]

APROPOS scans all symbols in the system for those which have *STRING* as a substring and prints them on the terminal along with a line for each relevant item defined for each selected symbol. Relevant items are:

- function definitions, for which only the arglist is printed
- dynamic variable values
- non-null property lists

PRINTLEVEL (see Chapter 25) is set to (3 . 5) when APROPOS is printing.

If *ALLFLG* is NIL, then symbols with no relevant items and "internal" symbols are omitted ("internal" currently means those symbols whose print name begins with a \ or those symbols produced by GENSYM). If *ALLFLG* is a function, it is used as a predicate on symbols selected by the substring match, with value NIL meaning to omit the symbol. If *ALLFLG* is any other non-NIL value, then no symbols are omitted.

Note: Unlike CL:APROPOS which lets you designate the package to search, APROPOS searches *all* packages.

## Using Symbols as Variables

Symbols are commonly used as variable names. Each symbol has a "top level" value, which can be an arbitrary object. Symbols may also be given special variable bindings within PROGs or functions, which only exist for the duration of the function. When a symbol is evaluated, the "current" variable binding is returned. This is the most recent special variable binding, or the top-level binding if the symbol hasn't been rebound. SETQ is used to change the current binding. For more information on variable bindings in Interlisp, see Chapter 11.

A symbol whose top-level value is the symbol NOBIND is considered to have no value. If a symbol has no local bindings, and its top-level value is NOBIND, trying to evaluate it will cause an unbound-atom error. In addition, if a symbol's local binding is to NOBIND, trying to evaluate it will cause an error.

The symbols T and NIL always evaluate to themselves. Attempting to change the value of T or NIL with the functions below will generate the error; `Attempt to set T` or `Attempt to set NIL`.

The following functions (except BOUNDP) will also generate the error `Arg not litatom`, if not given a symbol.

(**BOUNDP** *VAR*) [Function]

Returns T if *VAR* has a special variable binding, or if *VAR* has a top-level value other than NOBIND; otherwise NIL. That is, if X is a symbol, (EVAL X) will cause an `Unbound atom` error if and only if (BOUNDP X) returns NIL.

Note: The Interlisp interpreter has been modified so that it will generate an `Unbound Variable` error when it encounters any symbol bound to NOBIND. This is a change from previous releases that only signaled an error when a symbol had a top-level binding of NOBIND in addition to no dynamic binding.

(**SET** *VAR VALUE*) [NoSpread Function]

Sets the "current" value of *VAR* to *VALUE*, and returns *VALUE*.

SET is a normal function, so both *VAR* and *VALUE* are evaluated before it is called. Thus, if the value of X is B, and value of Y is C, then (SET X Y) would result in B being set to C, and C being returned as the value of SET.

(**SETQ** *VAR VALUE*) [NoSpread Function]

Like SET, but *VAR* is not evaluated, *VALUE* is. Thus, if the value of X is B and the value of Y is C, (SETQ X Y) would result in X (not B) being set to C, and C being returned.

Actually, neither argument is evaluated during the calling process. However, SETQ itself calls EVAL on its second argument. As a result, typing (SETQ VAR FORM) and SETQ (VAR FORM) to the Interlisp Executive are equivalent: in both cases VAR is not evaluated, and FORM is.

(**SETQQ** *VAR VALUE*) [NoSpread Function]

Like SETQ, but neither argument is evaluated, e.g., (SETQQ X (A B C)) sets X to (A B C).

(**PSETQ** *VAR$_1$ VALUE$_1$ ... VAR$_N$ VALUE$_N$*) [Macro]

Does a SETQ in parallel of *VAR$_1$* (unevaluated) to *VALUE$_1$*, *VAR$_2$* to *VALUE$_2$*, etc. All of the *VALUE$_i$* terms are evaluated before any of the assignments. Therefore, (PSETQ A B B A) can be used to swap the values of the variables A and B.

(**GETTOPVAL** *VAR*) [Function]

Returns the top level value of *VAR* (even if NOBIND), regardless of any intervening local bindings.

(**SETTOPVAL** *VAR VALUE*) [Function]

Sets the top level value of *VAR* to *VALUE*, regardless of any intervening bindings, and returns *VALUE*.

(**GETATOMVAL** *VAR*) [Function]

Same as (GETTOPVAL VAR).

(**SETATOMVAL** *VAR VALUE*) [Function]

Same as SETTOPVAL.

Note: The compiler (see Chapter 18) treats variables somewhat differently from the interpreter, and you need to be aware of these differences when writing functions that will be compiled. For example, variable references in compiled code are not checked for NOBIND, so compiled code will not generate unbound-atom errors. In general, it is better to debug interpreted code, before compiling it for speed. The compiler offers some facilities to increase the efficiency of variable use in compiled functions: Global variables can be defined so that the entire stack is not searched at each variable reference. Local variables have bindings that are not visible outside the function, which reduces variable conflicts and makes variable lookup faster.

## Function Definition Cells

Each symbol has a function-definition cell, which is accessed when that symbol is used as a function. This is described in detail in Chapter 10.

## Property Lists

Each symbol has an associated property list, which allows a set of named objects to be associated with the symbol. A property list associates a name (known as a "property name" or "property") with an arbitrary object (the "property value" or "value"). Sometimes the phrase "to store on the property X" is used, meaning to place the indicated information on a property list under the property name X.

Property names are usually symbols or numbers, although no checks are made. However, the standard property list functions all use EQ to search for property names, so they may not work with non-atomic property names. The same object can be used as both a property name and a property value.

Many symbols in the system already have property lists, with properties used by the compiler, the break package, DWIM, etc. Be careful not to clobber such system properties. The variable SYSPROPS is a list of property names used by the system.

The functions below are used to manipulate the property lists of symbols. Except when indicated, they generate the error *ATM* `is not a SYMBOL`, if given an object that is not a symbol.

(**GETPROP** *ATM PROP*) [Function]

Returns the property value for *PROP* from the property list of *ATM*. Returns `NIL` if *ATM* is not a symbol, or *PROP* is not found. `GETPROP` also returns `NIL` if there is an occurrence of *PROP* but the corresponding property value is `NIL`. This can be a source of program errors.

Note: `GETPROP` used to be called `GETP`.

(**PUTPROP** *ATM PROP VAL*) [Function]

Puts the property *PROP* with value *VAL* on the property list of *ATM*. *VAL* replaces any previous value for the property *PROP* on this property list. Returns *VAL*.

(**ADDPROP** *ATM PROP NEW FLG*) [Function]

Adds the value *NEW* to the list which is the value of property *PROP* on the property list of the *ATM*. If *FLG* is `T`, *NEW* is `CONS`ed onto the front of the property value of *PROP*; otherweise, it is `NCONC`ed on the end (using `NCONC1`). If *ATM* does not have a property *PROP*, or the value is not a list, then the effect is the same as `(PUTPROP ATM PROP (LIST NEW))`. `ADDPROP` returns the (new) property value. Example:

```
←(PUTPROP 'POCKET 'CONTENTS NIL)
 (NIL)
←(ADDPROP 'POCKET 'CONTENTS 'COMB)
 (COMB)
←(ADDPROP 'POCKET 'CONTENTS 'WALLET)
 (COMB WALLET)
```

(**REMPROP** *ATM PROP*) [Function]

Removes all occurrences of the property *PROP* (and its value) from the property list of *ATM*. Returns *PROP* if any were found (`T` if *PROP* is `NIL`), otherwise `NIL`.

(**CHANGEPROP** *X PROP1 PROP2*) [Function]

Changes the property name of property *PROP1* to *PROP2* on the property list of *X* (but does not affect the value of the property). Returns *X*, unless *PROP1* is not found, in which case it returns `NIL`.

(**PROPNAMES** *ATM*) [Function]

Returns a list of the property names on the property list of *ATM*.

(**DEFLIST** *L PROP*) [Function]

Used to put values under the same property name on the property lists of several symbols. *L* is a list of two-element lists. The first element of each is a symbol, and the second element is the property vqalue of the property *PROP*. Returns `NIL`. For example:

```
(DEFLIST '((FOO MA)(BAR CA)(BAZ RI)) 'STATE)
```

puts MA on FOO's STATE property, CA on BAR's STATE property, and RI on BAZ's STATE property.

Property lists are conventionally implemented as lists of the form

($NAME_1$ $VALUE_1$ $NAME_2$ $VALUE_2$...)

although the user can store anything as the property list of a symbol. However, thge functions which manipulate property lists observe this convention by searching down the property lists two CDRs at a time. Most of these functions also generate the error `Arg not litatom` if given an argument which is not a symbol, so they cannot be used directly on lists. (LISTPUT, LISTPUT1, LISTGET, and LISTGET1 are functions similar to PUTPROP and GETPROP that work directly on lists (see Chapter 3) . The property lists of symbols can be directly accessed with the following functions.

(**GETPROPLIST** *ATM*) [Function]

Returns the property list of *ATM*.

(**SETPROPLIST** *ATM LST*) [Function]

If *ATM* is a symbol, sets the property list of *ATM* to be *LST*, and returns *LST* as its value.

(**GETLIS** *X PROPS*) [Function]

Searches the property list of *X*, and returns the property list as of the first property on *PROPS* that it finds. For example:

```
←(GETPROPLIST 'X)
 (PROP1 A PROP3 B A C)
←(GETLIS 'X '(PROP2 PROP3))
 (PROP3 B A C)
```

Returns NIL if no element on props is found. *X* can also be a list itself, in which case it is searched as described above. If *X* is not a symbol or a list, returns NIL.

(**REMPROPLIST** *ATM PROPS*) [Function]

Removes all occurrences of all properties on the list *PROPS* (and their corresponding property values) from the property list of *ATM*. Returns NIL.

## Print Names

The term "print name" has an extended meaning: The characters that are output when *any object* is printed. In Medley, all objects have print names, although only symbols and strings have their print names explicitly stored. Symbol print names are limited to 255 characters.

This section describes a set of functions that can be used to access and manipulate the print names of any object, though they are primarily used with the print names of symbols. In Medley, print functions qualify symbol names with a package prefix if the symbol is not accessible in the current package. The exception is Interlisp's PRIN1, which does not include a package prefix.

The print name of an object is those characters that are output when the object is printed using PRIN1, e.g., the print name of the list `(A B "C")` consists of the seven characters `(A B C)` (two of the characters are spaces).

The PRIN2-name of an object is those characters output when the object is printed using PRIN2. Thus the PRIN2-name of the list (A B "C") is the 9 characters (A B "C") (including the two spaces). The PRIN2-name depends on what readtable is being used (see Chapter 25), since this determines where %s will be inserted. Many of the functions below allow either print names of PRIN2-names to be used, as specified by FLG and RDTBL arguments. If FLG is NIL, print names are used. Otherwise, PRIN2-names are used, computed with respect to the readtable RDTBL (or the current readtable, if RDTBL = NIL).

The print name of an integer depends on the setting of RADIX (see Chapter 25). The functions described in this section (UNPACK, NCHARS, etc.) define the print name of an integer as though the radix was 10, so that (PACK (UNPACK 'X9)) will always be X9 (and not X11, if RADIX is set to 8). However, integers will still be printed by PRIN1 using the current radix. The user can force these functions to use print names in the current radix by changing the setting of the variable PRXFLG (see Chapter 25).

(**CL:SYMBOL-NAME** *SYM*) [Common Lisp Function]

Returns a string displaced to the *SYM* print name. Strings returned from CL:SYMBOL-NAME may be destructively modified without affecting *SYM*'s print name.

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

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

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

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

(**NTHCHAR** *X N FLG RDTBL*) [Function]

Returns *X*, if *X* is a tail of the list Y; otherwise NIL. *X* is a tail of Y if it is EQ to 0 or more CDRs of Y.

```
(NTHCHAR 'ABC 2)  =>  B
(NTHCHAR 15.6 2)  =>  5
(NTHCHAR 'ABC%(D -3 T)  =>  %%
(NTHCHAR "ABC" 2)  =>  B
(NTHCHAR "ABC" 2 T)  =>  A
```

NTHCAR and NCHARS work much faster on objects that actually have an internal representation of their print name, i.e., symbols and strings, than they do on numbers and lists, since they don't have to simulate printing.

(**L-CASE** *X FLG*) [Function]

Returns a lowercase version of *X*. If *FLG* is T, the first letter is capitalized. If *X* is a string, the value of L-CASE is also a string. If *X* is a list, L-CASE returns a new list in which L-CASE is computed for each corresponding element and non-NIL tail of the original list. Examples:

```
(L-CASE 'FOO) => foo
(L-CASE 'FOO T) => Foo
(L-CASE "FILE NOT FOUND" T) => "File not found"
```

```
(L-CASE '(JANUARY FEBRUARY (MARCH "APRIL")) T) =>
   '(January February (March "April"))
```

(**U-CASE** *X*) [Function]

Like L-CASE, but returns the uppercase version of *X*.

(**U-CASEP** *X*) [Function]

Returns T if *X* contains no lowercase letters; NIL otherwise.

## Characters and Character Codes

Characters are represented 3 different ways in Medley. In Interlisp they are single-character symbols or integer character codes. In Common Lisp they are instances of the CHARACTER datatype. In general Interlisp character functions don't accept Common Lisp characters and vice versa. The only exceptions are Interlisp string-manipulation functions that accept "string or symbol" types as arguments.

You can convert between Interlisp and Common Lisp characaters by using the functions CL:CODE-CHAR, CL:CHAR-CODE, and CHARCODE (see below).

Medley uses the 16-bit NS character set, described in the document Character Code Standard (Xerox System Integration Standards, XSIS 058404, April 1984). Legal character codes range from 0 to 65535. The NS (Network Systems) character encoding encompasses a much wider set of available characters than the 8-bit character standards (such as ASCII), including characters comprising many foreign alphabets and special symbols. For instance, Medley supports the display and printing of the following:

- Le système d'informtion Medley est remarqueablement polyglotte
- Das Medley Kommunikationssystem bietet merkwürdige multilinguale Nutzmöglichkeiten
- M ⊆ ❑ [w] ⇔ ∀ v with Rwv: M ⊆ [v]

These characters can be used in strings, symbol print names, symbolic files, or anywhere else 8-bit characters could be used. All of the standard string and print name functions (RPLSTRING, GNC, NCHARS, STRPOS, etc.) accept symbols and strings containing NS characters. For example:

```
←(STRPOS "char""this is an 8-bit character string")
   18
←(STRPOS "char""celui-ci comports des characteres NS")
   23
```

In almost all cases, a program does not have to distinguish between NS characters or 8-bit characters. The exception to this rule is the handling of input/output operations (see Chapter 25).

The function CHARCODE (see below) provides a simple way to create individual NS character codes. The VirtualKeyboards library module provides a set of virtual keyboards that allows keyboard or mouse entry of NS characters.

(**PACKC** *X*) [Function]

Like PACK except *X* is a list of character codes. For example,

```
(PACKC '(70 79 79))  => FOO
```

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

Like UNPACK, but returns the print name of *X* as a list of character codes. If *FLG* = T, the PRIN2-name is used. For example:

```
(CHCON 'FOO)  => (70 79 79)
```

(**DCHCON** *X SCRATCHLIST FLG RDTBL*) [Function]

Like DUNPACK.

(**NTHCHARCODE** *X N FLG RDTBL*) [Function]

Like NTHCHAR, but returns the character code of the *N*th character of the print name of *X*. If *N* is negative, it is interpreted as a count backwards from the end of *X*. If the absolute value of *N* is greater than the number of characters in *X*, or 0, then the value of NTHCHARCODE is NIL.

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

(**CHCON1** *X*) [Function]

Returns the character code of the first character of the print name of *X*; equal to (NTHCHARCODE X 1).

(**CHARACTER** *N*) [Function]

*N* is a character code. Returns the symbol having the corresponding single character as its print name.

```
(CHARACTER 70) => F
```

(**FCHARACTER** *N*) [Function]

Fast version of CHARACTER that compiles open.

The following function makes it possible to gain the efficiency that comes from dealinig with character codes without losing the symbolic advantages of character symbols.

(**CHARCODE** *CHAR*) [Function]

Returns the character code specified by *CHAR* (unevaluated). If *CHAR* is a one-character symbol or string, the corresponding character code is simply returned. Thus, (CHARCODE A) is 65, (CHARCODE 0) is 48. If *CHAR* is a multi-character symbol or string, it specifies a character code as described below. If *CHAR* is NIL, CHARCODE simply returns NIL. Finally, if *CHAR* is a list structure, the value is a copy of *CHAR* with all the leaves replaced by the corresponding character codes. For instance, (CHARCODE (A (B C))) => (65 (66 67)).

If a character is specified by a multi-character symbol or string, CHARCODE interprets it as follows:

CR, SPACE, etc.

The variable CHARACTERNAMES contains an association list mapping special symbols to character codes. Among the characters defined this way are CR (13), LF (10), SPACE or SP (32), ESCAPE or ESC (27), BELL (7), BS (8), TAB (9), NULL (0), and DEL (127). The symbol EOL maps into the appropriate end-of-line character code in the different Interlisp implementations (31 in Interlisp-10, 13 in Interlisp-D, 10 in Interlisp-VAX). Examples:

```
(CHARCODE SPACE)  =>  32
(CHARCODE CR)  =>  13
```

CHARSET, CHARNUM, CHARSET-CHARNUM

If the character specification is a symbol or string of the form CHARSET, CHARNUM, or CHARSET-CHARNUM, the character code for the character number CHARNUM in the character set CHARSET is returned.

The 16-bit NS character encoding is divided into a large number of "character sets". Each 16-bit character can be decoded into a character set (an integer from 0 to 254 inclusive) and a character number (also an integer from 0 to 254 inclusive). CHARSET is either an octal number, or a symbol in the association list CHARACTERSETNAMES (which defines the character sets for GREEK, CYRILLIC, etc.).

CHARNUM is either an octal number, a single-character symbol, or a symbol from the association list CHARACTERNAMES. If CHARNUM is a single-digit number, it is interpreted as the character "2", rather than as the octal number 2. Examples:

```
(CHARCODE 12,6)  =>  2566
(CHARCODE 12,SPACE)  =>  2592
(CHARCODE GREEK,A) => 9793
```

↑CHARSPEC (control chars)

If the character specification is a symbol or string of one of the forms above, preceded by the character ↑, this indicates a "control character," derived from the normal character code by clearing the seventh bit of the character code (normally set). Examples:

```
(CHARCODE ↑A)  =>  1
(CHARCODE ↑GREEK,A)  =>  9729
```

#CHARSPEC (meta chars)

If the character specification is a symbol or string of one of the forms above, preceded by the charactger #, this indicates a meta character, derived from the normal character code by setting the eighth bit of the character code (normally cleared). ↑ and # can both be set at once. Examples:

```
(CHARCODE #A)  =>  193
(CHARCODE #↑GREEK,A)  =>  9857
```

A CHARCODE form can be used wherever a structure of character codes would be appropriate. For example:

```
(FMEMB (NTHCHARCODE X 1)(CHARCODE (CR LF SPACE ↑A)))
(EQ (READCCODE FOO)(CHARCODE GREEK,A))
```

There is a macro for CHARCODE which causes the character-code structure to be constructed at compile-time. Thus, the compiled code for these examples is exactly as efficient as the less readable:

```
(FMEMB (NTHCHARCODE X 1)(QUOTE (13 10 32 1)))
(EQ (READCCODE FOO)9793)
```

(**CL:CHAR-CODE** *CHAR* ) [Common Lisp Function]

Returns the Interlisp character code of *CHAR*. Use to convert a Common Lisp character to an Interlisp character code.

(**CL:CODE-CHAR** *N* ) [Common Lisp Function]

Returns a character with the given non-negative integer *N* code. Returns NIL if no character is possible with *N*. Use to convert an Interlisp character code to a Common Lisp character.

(**SELCHARQ** *E CLAUSE*$_1$*... CLAUSE*$_N$ *DEFAULT*) [Function]

Lets you branch one of several ways, based on the character code *E*. The first item in each *CLAUSE*$_N$ is a character code or list of character codes, given in the form CHARCODE would accept. If the value of *E* is a character code or NIL, and it is EQ or MEMB to the result of applying CHARCODE to the first element of a clause, the remaining forms of that clause are evaluated. Otherwise, the default is evaluated.

Thus

```
(SELCHARQ (BIN FOO))
     ((SPACE TAB)(FUM))
     ((↑D NIL)(BAR))
     (a (BAZ))
     (ZIP)))
```

is exactly equivalent to

```
(SELECTQ (BIN FOO))
     ((32 9)(FUM))
     ((4 NIL)(BAR))
     (97 (BAZ))
     (ZIP)))
```

If (BIN FOO) returned 32 (the SPACE character), the function FUM would be called.

[This page intentionally left blank]