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

Programming Language—Common Lisp

13. Characters

13.1 Character Concepts

13.1.1 Introduction to Characters

A ***character*** is an *object* that represents a unitary token (*e.g.*, a letter, a special symbol, or a "control character") in an aggregate quantity of text (*e.g.*, a *string* or a text *stream*).

Common Lisp allows an implementation to provide support for international language *characters* as well as *characters* used in specialized arenas (*e.g.*, mathematics).

The following figures contain lists of *defined names* applicable to *characters*.

Figure 13–1 lists some *defined names* relating to *character attributes* and *character predicates*.

alpha-char-p	**char-not-equal**	**char>**
alphanumericp	**char-not-greaterp**	**char>=**
both-case-p	**char-not-lessp**	**digit-char-p**
char-code-limit	**char/=**	**graphic-char-p**
char-equal	**char<**	**lower-case-p**
char-greaterp	**char<=**	**standard-char-p**
char-lessp	**char=**	**upper-case-p**

Figure 13–1. Character defined names – 1

Figure 13–2 lists some *character* construction and conversion *defined names*.

char-code	**char-name**	**code-char**
char-downcase	**char-upcase**	**digit-char**
char-int	**character**	**name-char**

Figure 13–2. Character defined names – 2

13.1.2 Introduction to Scripts and Repertoires

13.1.2.1 Character Scripts

A *script* is one of possibly several sets that form an *exhaustive partition* of the type **character**.

The number of such sets and boundaries between them is *implementation-defined*. Common Lisp does not require these sets to be *types*, but an *implementation* is permitted to define such *types* as an extension. Since no *character* from one *script* can ever be a member of another *script*, it is generally more useful to speak about *character repertoires*.

Although the term "*script*" is chosen for definitional compatibility with ISO terminology, no *conforming implementation* is required to use any particular *scripts* standardized by ISO or by any other standards organization.

Whether and how the *script* or *scripts* used by any given *implementation* are named is *implementation-dependent*.

13.1.2.2 Character Repertoires

A ***repertoire*** is a *type specifier* for a *subtype* of *type* **character**. This term is generally used when describing a collection of *characters* independent of their coding. *Characters* in *repertoires* are only identified by name, by *glyph*, or by character description.

A *repertoire* can contain *characters* from several *scripts*, and a *character* can appear in more than one *repertoire*.

For some examples of *repertoires*, see the coded character standards ISO 8859/1, ISO 8859/2, and ISO 6937/2. Note, however, that although the term "*repertoire*" is chosen for definitional compatibility with ISO terminology, no *conforming implementation* is required to use *repertoires* standardized by ISO or any other standards organization.

13.1.3 Character Attributes

Characters have only one *standardized attribute*: a *code*. A *character*'s *code* is a non-negative *integer*. This *code* is composed from a character *script* and a character label in an *implementation-dependent* way. See the *functions* **char-code** and **code-char**.

Additional, *implementation-defined attributes* of *characters* are also permitted so that, for example, two *characters* with the same *code* may differ in some other, *implementation-defined* way.

For any *implementation-defined attribute* there is a distinguished value called the ***null*** value for that *attribute*. A *character* for which each *implementation-defined attribute* has the null value for that *attribute* is called a *simple character*. If the *implementation* has no *implementation-defined attributes*, then all *characters* are *simple characters*.

13.1.4 Character Categories

There are several (overlapping) categories of *characters* that have no formally associated *type* but that are nevertheless useful to name. They include *graphic characters*, *alphabetic*$_1$ *characters*, *characters* with *case* (*uppercase* and *lowercase characters*), *numeric characters*, *alphanumeric characters*, and *digits* (in a given *radix*).

For each *implementation-defined attribute* of a *character*, the documentation for that *implementation* must specify whether *characters* that differ only in that *attribute* are permitted to differ in whether are not they are members of one of the aforementioned categories.

Note that these terms are defined independently of any special syntax which might have been enabled in the *current readtable*.

13.1.4.1 Graphic Characters

Characters that are classified as ***graphic***, or displayable, are each associated with a glyph, a visual representation of the *character*.

A *graphic character* is one that has a standard textual representation as a single *glyph*, such as `A` or `*` or `=`. *Space*, which effectively has a blank *glyph*, is defined to be a *graphic*.

Of the *standard characters*, *newline* is *non-graphic* and all others are *graphic*; see Section 2.1.3 (Standard Characters).

Characters that are not *graphic* are called ***non-graphic***. *Non-graphic characters* are sometimes informally called "formatting characters" or "control characters."

`#\Backspace`, `#\Tab`, `#\Rubout`, `#\Linefeed`, `#\Return`, and `#\Page`, if they are supported by the *implementation*, are *non-graphic*.

13.1.4.2 Alphabetic Characters

The *alphabetic*$_1$ *characters* are a subset of the *graphic characters*. Of the *standard characters*, only these are the *alphabetic*$_1$ *characters*:

`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

`a b c d e f g h i j k l m n o p q r s t u v w x y z`

Any *implementation-defined character* that has *case* must be *alphabetic*$_1$. For each *implementation-defined graphic character* that has no *case*, it is *implementation-defined* whether that *character* is *alphabetic*$_1$.

13.1.4.3 Characters With Case

The *characters* with *case* are a subset of the *alphabetic*$_1$ *characters*. A *character* with *case* has the property of being either *uppercase* or *lowercase*. Every *character* with *case* is in one-to-one correspondence with some other *character* with the opposite *case*.

13.1.4.3.1 Uppercase Characters

An uppercase *character* is one that has a corresponding *lowercase character* that is *different* (and can be obtained using **char-downcase**).

Of the *standard characters*, only these are *uppercase characters*:

`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

13.1.4.3.2 Lowercase Characters

A lowercase *character* is one that has a corresponding *uppercase character* that is *different* (and can be obtained using **char-upcase**).

Of the *standard characters*, only these are *lowercase characters*:

`a b c d e f g h i j k l m n o p q r s t u v w x y z`

13.1.4.3.3 Corresponding Characters in the Other Case

The *uppercase standard characters* `A` through `Z` mentioned above respectively correspond to the *lowercase standard characters* `a` through `z` mentioned above. For example, the *uppercase character* `E` corresponds to the *lowercase character* `e`, and vice versa.

13.1.4.3.4 Case of Implementation-Defined Characters

An *implementation* may define that other *implementation-defined graphic characters* have *case*. Such definitions must always be done in pairs—one *uppercase character* in one-to-one *correspondence* with one *lowercase character*.

13.1.4.4 Numeric Characters

The *numeric characters* are a subset of the *graphic characters*. Of the *standard characters*, only these are *numeric characters*:

`0 1 2 3 4 5 6 7 8 9`

For each *implementation-defined graphic character* that has no *case*, the *implementation* must define whether or not it is a *numeric character*.

13.1.4.5 Alphanumeric Characters

The set of *alphanumeric characters* is the union of the set of *alphabetic*$_1$ *characters* and the set of *numeric characters*.

13.1.4.6 Digits in a Radix

What qualifies as a *digit* depends on the *radix* (an *integer* between `2` and `36`, inclusive). The potential *digits* are:

```
0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Their respective weights are `0`, `1`, `2`, ... `35`. In any given radix n, only the first n potential *digits* are considered to be *digits*. For example, the digits in radix `2` are `0` and `1`, the digits in radix `10` are `0` through `9`, and the digits in radix `16` are `0` through `F`.

Case is not significant in *digits*; for example, in radix `16`, both `F` and `f` are *digits* with weight `15`.

13.1.5 Identity of Characters

Two *characters* that are **eql**, **char=**, or **char-equal** are not necessarily **eq**.

13.1.6 Ordering of Characters

The total ordering on *characters* is guaranteed to have the following properties:

- If two *characters* have the same *implementation-defined attributes*, then their ordering by **char<** is consistent with the numerical ordering by the predicate `<` on their code *attributes*.

- If two *characters* differ in any *attribute*, then they are not **char=**.

- The total ordering is not necessarily the same as the total ordering on the *integers* produced by applying **char-int** to the *characters*.

- While *alphabetic*$_1$ *standard characters* of a given *case* must obey a partial ordering, they need not be contiguous; it is permissible for *uppercase* and *lowercase characters* to be interleaved. Thus `(char<= #\a x #\z)` is not a valid way of determining whether or not `x` is a *lowercase character*.

Of the *standard characters*, those which are *alphanumeric* obey the following partial ordering:

```
A<B<C<D<E<F<G<H<I<J<K<L<M<N<O<P<Q<R<S<T<U<V<W<X<Y<Z
a<b<c<d<e<f<g<h<i<j<k<l<m<n<o<p<q<r<s<t<u<v<w<x<y<z
0<1<2<3<4<5<6<7<8<9
either 9<A or Z<0
either 9<a or z<0
```

This implies that, for *standard characters*, *alphabetic*$_1$ ordering holds within each *case* (*uppercase* and *lowercase*), and that the *numeric characters* as a group are not interleaved with *alphabetic characters*. However, the ordering or possible interleaving of *uppercase characters* and *lowercase characters* is *implementation-defined*.

13.1.7 Character Names

The following *character names* must be present in all *conforming implementations*:

`Newline`

The character that represents the division between lines. An implementation must translate between `#\Newline`, a single-character representation, and whatever external representation(s) may be used.

`Space`

The space or blank character.

The following names are *semi-standard*; if an *implementation* supports them, they should be used for the described *characters* and no others.

`Rubout`

The rubout or delete character.

`Page`

The form-feed or page-separator character.

`Tab`

The tabulate character.

`Backspace`

The backspace character.

`Return`

The carriage return character.

`Linefeed`

The line-feed character.

In some *implementations*, one or more of these *character names* might denote a *standard character*; for example, `#\Linefeed` and `#\Newline` might be the *same character* in some *implementations*.

13.1.8 Treatment of Newline during Input and Output

When the character `#\Newline` is written to an output file, the implementation must take the appropriate action to produce a line division. This might involve writing out a record or translating `#\Newline` to a CR/LF sequence. When reading, a corresponding reverse transformation must take place.

13.1.9 Character Encodings

A *character* is sometimes represented merely by its *code*, and sometimes by another *integer* value which is composed from the *code* and all *implementation-defined attributes* (in an *implementation-defined* way that might vary between *Lisp images* even in the same *implementation*). This *integer*, returned by the function **char-int**, is called the character's "encoding." There is no corresponding function from a character's encoding back to the *character*, since its primary intended uses include things like hashing where an inverse operation is not really called for.

13.1.10 Documentation of Implementation-Defined Scripts

An *implementation* must document the *character scripts* it supports. For each *character script* supported, the documentation must describe at least the following:

- Character labels, glyphs, and descriptions. Character labels must be uniquely named using only Latin capital letters A–Z, hyphen (-), and digits 0–9.
- Reader canonicalization. Any mechanisms by which **read** treats *different* characters as equivalent must be documented.
- The impact on **char-upcase**, **char-downcase**, and the case-sensitive *format directives*. In particular, for each *character* with *case*, whether it is *uppercase* or *lowercase*, and which *character* is its equivalent in the opposite case.
- The behavior of the case-insensitive *functions* **char-equal**, **char-not-equal**, **char-lessp**, **char-greaterp**, **char-not-greaterp**, and **char-not-lessp**.
- The behavior of any *character predicates*; in particular, the effects of **alpha-char-p**, **lower-case-p**, **upper-case-p**, **both-case-p**, **graphic-char-p**, and **alphanumericp**.
- The interaction with file I/O, in particular, the supported coded character sets (for example, ISO8859/1-1987) and external encoding schemes supported are documented.

character *System Class*

Class Precedence List:

character, **t**

Description:

A *character* is an *object* that represents a unitary token in an aggregate quantity of text; see Section 13.1 (Character Concepts).

The *types* **base-char** and **extended-char** form an *exhaustive partition* of the *type* **character**.

See Also:

Section 13.1 (Character Concepts), Section 2.4.8.1 (Sharpsign Backslash), Section 22.1.3.2 (Printing Characters)

base-char *Type*

Supertypes:

base-char, **character**, **t**

Description:

The *type* **base-char** is defined as the *upgraded array element type* of **standard-char**. An *implementation* can support additional *subtypes* of *type* **character** (besides the ones listed in this standard) that might or might not be *supertypes* of *type* **base-char**. In addition, an *implementation* can define **base-char** to be the *same type* as **character**.

Base characters are distinguished in the following respects:

1. The *type* **standard-char** is a *subrepertoire* of the *type* **base-char**.
2. The selection of *base characters* that are not *standard characters* is implementation defined.
3. Only *objects* of the *type* **base-char** can be *elements* of a *base string*.
4. No upper bound is specified for the number of characters in the **base-char** *repertoire*; the size of that *repertoire* is *implementation-defined*. The lower bound is 96, the number of *standard characters*.

Whether a character is a *base character* depends on the way that an *implementation* represents *strings*, and not any other properties of the *implementation* or the host operating system. For example, one implementation might encode all *strings* as characters having 16-bit encodings, and another might have two kinds of *strings*: those with characters having 8-bit encodings and those with characters having 16-bit encodings. In the first *implementation*, the *type* **base-char** is equivalent to the *type* **character**: there is only one kind of *string*. In the second *implementation*, the *base characters* might be those *characters* that could be stored in a *string* of *characters* having 8-bit encodings. In such an implementation, the *type* **base-char** is a *proper subtype* of the *type* **character**.

The *type* **standard-char** is a *subtype* of *type* **base-char**.

standard-char *Type*

Supertypes:

standard-char, **base-char**, **character**, **t**

Description:

A fixed set of 96 *characters* required to be present in all *conforming implementations*. *Standard characters* are defined in Section 2.1.3 (Standard Characters).

Any *character* that is not *simple* is not a *standard character*.

See Also:

Section 2.1.3 (Standard Characters)

extended-char *Type*

Supertypes:

extended-char, **character**, **t**

Description:

The *type* **extended-char** is equivalent to the *type* `(and character (not base-char))`.

Notes:

The *type* **extended-char** might have no *elements*$_4$ in *implementations* in which all *characters* are of *type* **base-char**.

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

Syntax:

char= &rest *characters*$^+$ → *generalized-boolean*
char/ = &rest *characters*$^+$ → *generalized-boolean*
char< &rest *characters*$^+$ → *generalized-boolean*
char> &rest *characters*$^+$ → *generalized-boolean*
char<= &rest *characters*$^+$ → *generalized-boolean*
char>= &rest *characters*$^+$ → *generalized-boolean*

char-equal &rest *characters*$^+$ → *generalized-boolean*
char-not-equal &rest *characters*$^+$ → *generalized-boolean*
char-lessp &rest *characters*$^+$ → *generalized-boolean*
char-greaterp &rest *characters*$^+$ → *generalized-boolean*
char-not-greaterp &rest *characters*$^+$ → *generalized-boolean*
char-not-lessp &rest *characters*$^+$ → *generalized-boolean*

Arguments and Values:

character—a *character*.

generalized-boolean—a *generalized boolean*.

Description:

These predicates compare *characters*.

char= returns *true* if all *characters* are the *same*; otherwise, it returns *false*. If two *characters* differ in any *implementation-defined attributes*, then they are not **char=**.

char/= returns *true* if all *characters* are different; otherwise, it returns *false*.

char< returns *true* if the *characters* are monotonically increasing; otherwise, it returns *false*. If two *characters* have *identical implementation-defined attributes*, then their ordering by **char<** is consistent with the numerical ordering by the predicate **<** on their *codes*.

char> returns *true* if the *characters* are monotonically decreasing; otherwise, it returns *false*. If two *characters* have *identical implementation-defined attributes*, then their ordering by **char>** is consistent with the numerical ordering by the predicate **>** on their *codes*.

char<= returns *true* if the *characters* are monotonically nondecreasing; otherwise, it returns *false*. If two *characters* have *identical implementation-defined attributes*, then their ordering by **char<=** is consistent with the numerical ordering by the predicate **<=** on their *codes*.

char>= returns *true* if the *characters* are monotonically nonincreasing; otherwise, it returns *false*.

If two *characters* have *identical implementation-defined attributes*, then their ordering by **char>=** is consistent with the numerical ordering by the predicate `>=` on their *codes*.

char-equal, **char-not-equal**, **char-lessp**, **char-greaterp**, **char-not-greaterp**, and **char-not-lessp** are similar to **char=**, **char/=**, **char<**, **char>**, **char<=**, **char>=**, respectively, except that they ignore differences in *case* and might have an *implementation-defined* behavior for *non-simple characters*. For example, an *implementation* might define that **char-equal**, *etc.* ignore certain *implementation-defined attributes*. The effect, if any, of each *implementation-defined attribute* upon these functions must be specified as part of the definition of that *attribute*.

Examples:

```
(char= #\d #\d) → true
(char= #\A #\a) → false
(char= #\d #\x) → false
(char= #\d #\D) → false
(char/= #\d #\d) → false
(char/= #\d #\x) → true
(char/= #\d #\D) → true
(char= #\d #\d #\d #\d) → true
(char/= #\d #\d #\d #\d) → false
(char= #\d #\d #\x #\d) → false
(char/= #\d #\d #\x #\d) → false
(char= #\d #\y #\x #\c) → false
(char/= #\d #\y #\x #\c) → true
(char= #\d #\c #\d) → false
(char/= #\d #\c #\d) → false
(char< #\d #\x) → true
(char<= #\d #\x) → true
(char< #\d #\d) → false
(char<= #\d #\d) → true
(char< #\a #\e #\y #\z) → true
(char<= #\a #\e #\y #\z) → true
(char< #\a #\e #\e #\y) → false
(char<= #\a #\e #\e #\y) → true
(char> #\e #\d) → true
(char>= #\e #\d) → true
(char> #\d #\c #\b #\a) → true
(char>= #\d #\c #\b #\a) → true
(char> #\d #\d #\c #\a) → false
(char>= #\d #\d #\c #\a) → true
(char> #\e #\d #\b #\c #\a) → false
(char>= #\e #\d #\b #\c #\a) → false
(char> #\z #\A) → implementation-dependent
(char> #\Z #\a) → implementation-dependent
(char-equal #\A #\a) → true
```

```
 (stable-sort (list #\b #\A #\B #\a #\c #\C) #'char-lessp)
→ (#\A #\a #\b #\B #\c #\C)
 (stable-sort (list #\b #\A #\B #\a #\c #\C) #'char<)
→ (#\A #\B #\C #\a #\b #\c) ;Implementation A
→ (#\a #\b #\c #\A #\B #\C) ;Implementation B
→ (#\a #\A #\b #\B #\c #\C) ;Implementation C
→ (#\A #\a #\B #\b #\C #\c) ;Implementation D
→ (#\A #\B #\a #\b #\C #\c) ;Implementation E
```

Exceptional Situations:

Should signal an error of *type* **program-error** if at least one *character* is not supplied.

See Also:

Section 2.1 (Character Syntax), Section 13.1.10 (Documentation of Implementation-Defined Scripts)

Notes:

If characters differ in their *code attribute* or any *implementation-defined attribute*, they are considered to be different by **char=**.

There is no requirement that `(eq c1 c2)` be true merely because `(char= c1 c2)` is *true*. While **eq** can distinguish two *characters* that **char=** does not, it is distinguishing them not as *characters*, but in some sense on the basis of a lower level implementation characteristic. If `(eq c1 c2)` is *true*, then `(char= c1 c2)` is also true. **eql** and **equal** compare *characters* in the same way that **char=** does.

The manner in which *case* is used by **char-equal**, **char-not-equal**, **char-lessp**, **char-greaterp**, **char-not-greaterp**, and **char-not-lessp** implies an ordering for *standard characters* such that `A=a`, `B=b`, and so on, up to `Z=z`, and furthermore either `9<A` or `Z<0`.

character *Function*

Syntax:

character *character* → *denoted-character*

Arguments and Values:

character—a *character designator*.

denoted-character—a *character*.

Description:

Returns the *character* denoted by the *character designator*.

Examples:

```
(character #\a) → #\a
(character "a") → #\a
(character 'a) → #\A
(character '\a) → #\a
(character 65.) is an error.
(character 'apple) is an error.
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *object* is not a *character designator*.

See Also:

coerce

Notes:

```
(character object) ≡ (coerce object 'character)
```

characterp *Function*

Syntax:

characterp *object* → *generalized-boolean*

Arguments and Values:

object—an *object*.

generalized-boolean—a *generalized boolean*.

Description:

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

Examples:

```
(characterp #\a) → true
(characterp 'a) → false
(characterp "a") → false
(characterp 65.) → false
(characterp #\Newline) → true
;; This next example presupposes an implementation
;; in which #\Rubout is an implementation-defined character.
(characterp #\Rubout) → true
```

See Also:

character (*type* and *function*), **typep**

Notes:

```
(characterp object) ≡ (typep object 'character)
```

alpha-char-p *Function*

Syntax:

alpha-char-p *character* → *generalized-boolean*

Arguments and Values:

character—a *character*.

generalized-boolean—a *generalized boolean*.

Description:

Returns *true* if *character* is an *alphabetic*$_1$ *character*; otherwise, returns *false*.

Examples:

```
(alpha-char-p #\a) → true
(alpha-char-p #\5) → false
(alpha-char-p #\Newline) → false
;; This next example presupposes an implementation
;; in which #\α is a defined character.
(alpha-char-p #\α) → implementation-dependent
```

Affected By:

None. (In particular, the results of this predicate are independent of any special syntax which might have been enabled in the *current readtable*.)

Exceptional Situations:

Should signal an error of *type* **type-error** if *character* is not a *character*.

See Also:

alphanumericp, Section 13.1.10 (Documentation of Implementation-Defined Scripts)

alphanumericp *Function*

Syntax:

alphanumericp *character* → *generalized-boolean*

Arguments and Values:

character—a *character*.

generalized-boolean—a *generalized boolean*.

Description:

Returns *true* if *character* is an *alphabetic*$_1$ *character* or a *numeric character*; otherwise, returns *false*.

Examples:

```
(alphanumericp #\Z) → true
(alphanumericp #\9) → true
(alphanumericp #\Newline) → false
(alphanumericp #\#) → false
```

Affected By:

None. (In particular, the results of this predicate are independent of any special syntax which might have been enabled in the *current readtable*.)

Exceptional Situations:

Should signal an error of *type* **type-error** if *character* is not a *character*.

See Also:

alpha-char-p, **graphic-char-p**, **digit-char-p**

Notes:

Alphanumeric characters are graphic as defined by **graphic-char-p**. The alphanumeric characters are a subset of the graphic characters. The standard characters `A` through `Z`, `a` through `z`, and `0` through `9` are alphanumeric characters.

```
(alphanumericp x)
  ≡ (or (alpha-char-p x) (not (null (digit-char-p x))))
```

digit-char *Function*

Syntax:

digit-char *weight* `&optional` *radix* → *char*

Arguments and Values:

weight—a non-negative *integer*.

radix—a *radix*. The default is `10`.

char—a *character* or *false*.

Description:

If *weight* is less than *radix*, **digit-char** returns a *character* which has that *weight* when considered as a digit in the specified radix. If the resulting *character* is to be an *alphabetic*$_1$ *character*, it will be an uppercase *character*.

If *weight* is greater than or equal to *radix*, **digit-char** returns *false*.

Examples:

```
(digit-char 0) → #\0
(digit-char 10 11) → #\A
(digit-char 10 10) → false
(digit-char 7) → #\7
(digit-char 12) → false
(digit-char 12 16) → #\C  ;not #\c
(digit-char 6 2) → false
(digit-char 1 2) → #\1
```

See Also:

digit-char-p, **graphic-char-p**, Section 2.1 (Character Syntax)

Notes:

digit-char-p *Function*

Syntax:

digit-char-p *char* `&optional` *radix* → *weight*

Arguments and Values:

char—a *character*.

radix—a *radix*. The default is 10.

weight—either a non-negative *integer* less than *radix*, or *false*.

Description:

Tests whether *char* is a digit in the specified *radix* (*i.e.*, with a weight less than *radix*). If it is a digit in that *radix*, its weight is returned as an *integer*; otherwise **nil** is returned.

Examples:

```
(digit-char-p #\5)    → 5
(digit-char-p #\5 2)  → false
(digit-char-p #\A)    → false
(digit-char-p #\a)    → false
(digit-char-p #\A 11) → 10
(digit-char-p #\a 11) → 10
(mapcar #'(lambda (radix)
            (map 'list #'(lambda (x) (digit-char-p x radix))
                 "059AaFGZ"))
        '(2 8 10 16 36))
→ ((0 NIL NIL NIL NIL NIL NIL NIL)
    (0 5 NIL NIL NIL NIL NIL NIL)
    (0 5 9 NIL NIL NIL NIL NIL)
    (0 5 9 10 10 15 NIL NIL)
    (0 5 9 10 10 15 16 35))
```

Affected By:

None. (In particular, the results of this predicate are independent of any special syntax which might have been enabled in the *current readtable*.)

See Also:

alphanumericp

Notes:

Digits are *graphic characters*.

---

graphic-char-p *Function*

Syntax:

graphic-char-p *char* → *generalized-boolean*

Arguments and Values:

char—a *character*.

generalized-boolean—a *generalized boolean*.

Description:

Returns *true* if *character* is a *graphic character*; otherwise, returns *false*.

Examples:

```
(graphic-char-p #\G) → true
(graphic-char-p #\#) → true
(graphic-char-p #\Space) → true
(graphic-char-p #\Newline) → false
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *character* is not a *character*.

See Also:

read, Section 2.1 (Character Syntax), Section 13.1.10 (Documentation of Implementation-Defined Scripts)

standard-char-p *Function*

Syntax:

standard-char-p *character* → *generalized-boolean*

Arguments and Values:

character—a *character*.

generalized-boolean—a *generalized boolean*.

Description:

Returns *true* if *character* is of *type* **standard-char**; otherwise, returns *false*.

Examples:

```
(standard-char-p #\Space) → true
(standard-char-p #\~) → true
;; This next example presupposes an implementation
;; in which #\Bell is a defined character.
(standard-char-p #\Bell) → false
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *character* is not a *character*.

char-upcase, char-downcase *Function*

Syntax:

char-upcase *character* → *corresponding-character*
char-downcase *character* → *corresponding-character*

Arguments and Values:

character, *corresponding-character*—a *character*.

Description:

If *character* is a *lowercase character*, **char-upcase** returns the corresponding *uppercase character*. Otherwise, **char-upcase** just returns the given *character*.

If *character* is an *uppercase character*, **char-downcase** returns the corresponding *lowercase character*. Otherwise, **char-downcase** just returns the given *character*.

The result only ever differs from *character* in its *code attribute*; all *implementation-defined attributes* are preserved.

Examples:

```
(char-upcase #\a) → #\A
(char-upcase #\A) → #\A
(char-downcase #\a) → #\a
(char-downcase #\A) → #\a
(char-upcase #\9) → #\9
(char-downcase #\9) → #\9
(char-upcase #\@) → #\@
(char-downcase #\@) → #\@
;; Note that this next example might run for a very long time in
;; some implementations if CHAR-CODE-LIMIT happens to be very large
;; for that implementation.
(dotimes (code char-code-limit)
  (let ((char (code-char code)))
    (when char
      (unless (cond ((upper-case-p char) (char= (char-upcase (char-downcase char)) char))
                    ((lower-case-p char) (char= (char-downcase (char-upcase char)) char))
                    (t (and (char= (char-upcase (char-downcase char)) char)
                            (char= (char-downcase (char-upcase char)) char))))
        (return char)))))
→ NIL
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *character* is not a *character*.

See Also:

upper-case-p, **alpha-char-p**, Section 13.1.4.3 (Characters With Case), Section 13.1.10 (Documentation of Implementation-Defined Scripts)

Notes:

If the *corresponding-char* is *different* than *character*, then both the *character* and the *corresponding-char* have *case*.

Since **char-equal** ignores the *case* of the *characters* it compares, the *corresponding-character* is always the *same* as *character* under **char-equal**.

upper-case-p, lower-case-p, both-case-p *Function*

Syntax:

upper-case-p *character* → *generalized-boolean*
lower-case-p *character* → *generalized-boolean*
both-case-p *character* → *generalized-boolean*

Arguments and Values:

character—a *character*.

generalized-boolean—a *generalized boolean*.

Description:

These functions test the case of a given *character*.

upper-case-p returns *true* if *character* is an *uppercase character*; otherwise, returns *false*.

lower-case-p returns *true* if *character* is a *lowercase character*; otherwise, returns *false*.

both-case-p returns *true* if *character* is a *character* with *case*; otherwise, returns *false*.

Examples:

```
(upper-case-p #\A) → true
(upper-case-p #\a) → false
(both-case-p #\a) → true
(both-case-p #\5) → false
(lower-case-p #\5) → false
(upper-case-p #\5) → false
;; This next example presupposes an implementation
;; in which #\Bell is an implementation-defined character.
(lower-case-p #\Bell) → false
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *character* is not a *character*.

See Also:

char-upcase, **char-downcase**, Section 13.1.4.3 (Characters With Case), Section 13.1.10 (Documentation of Implementation-Defined Scripts)

char-code *Function*

Syntax:

char-code *character* → *code*

Arguments and Values:

character—a *character*.

code—a *character code*.

Description:

char-code returns the *code attribute* of *character*.

Examples:

```
;; An implementation using ASCII character encoding
;; might return these values:
(char-code #\$) → 36
(char-code #\a) → 97
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *character* is not a *character*.

See Also:

char-code-limit

char-int *Function*

Syntax:

char-int *character* → *integer*

Arguments and Values:

character—a *character*.

integer—a non-negative *integer*.

Description:

Returns a non-negative *integer* encoding the *character* object. The manner in which the *integer* is computed is *implementation-dependent*. In contrast to **sxhash**, the result is not guaranteed to be independent of the particular *Lisp image*.

If *character* has no *implementation-defined attributes*, the results of **char-int** and **char-code** are the same.

```
(char= c1 c2) ≡ (= (char-int c1) (char-int c2))
```

for characters *c1* and *c2*.

Examples:

```
(char-int #\A) → 65       ; implementation A
(char-int #\A) → 577      ; implementation B
(char-int #\A) → 262145   ; implementation C
```

See Also:

char-code

code-char *Function*

Syntax:

code-char *code* → *char-p*

Arguments and Values:

code—a *character code*.

char-p—a *character* or **nil**.

Description:

Returns a *character* with the *code attribute* given by *code*. If no such *character* exists and one cannot be created, **nil** is returned.

Examples:

```
(code-char 65.) → #\A  ;in an implementation using ASCII codes
(code-char (char-code #\Space)) → #\Space  ;in any implementation
```

Affected By:

The *implementation*'s character encoding.

See Also:

char-code

Notes:

char-code-limit *Constant Variable*

Constant Value:

A non-negative *integer*, the exact magnitude of which is *implementation-dependent*, but which is not less than `96` (the number of *standard characters*).

Description:

The upper exclusive bound on the *value* returned by the *function* **char-code**.

See Also:

char-code

Notes:

The *value* of **char-code-limit** might be larger than the actual number of *characters* supported by the *implementation*.

char-name

char-name *Function*

Syntax:

char-name *character* → *name*

Arguments and Values:

character—a *character*.

name—a *string* or **nil**.

Description:

Returns a *string* that is the *name* of the *character*, or **nil** if the *character* has no *name*.

All *non-graphic* characters are required to have *names* unless they have some *implementation-defined attribute* which is not *null*. Whether or not other *characters* have *names* is *implementation-dependent*.

The *standard characters* ⟨*Newline*⟩ and ⟨*Space*⟩ have the respective names `"Newline"` and `"Space"`. The *semi-standard characters* ⟨*Tab*⟩, ⟨*Page*⟩, ⟨*Rubout*⟩, ⟨*Linefeed*⟩, ⟨*Return*⟩, and ⟨*Backspace*⟩ (if they are supported by the *implementation*) have the respective names `"Tab"`, `"Page"`, `"Rubout"`, `"Linefeed"`, `"Return"`, and `"Backspace"` (in the indicated case, even though name lookup by "`#\`" and by the *function* **name-char** is not case sensitive).

Examples:

```
 (char-name #\ ) → "Space"
 (char-name #\Space) → "Space"
 (char-name #\Page) → "Page"

 (char-name #\a)
→ NIL
→(or) "LOWERCASE-a"
→(or) "Small-A"
→(or) "LA01"

 (char-name #\A)
→ NIL
→(or) "UPPERCASE-A"
→(or) "Capital-A"
→(or) "LA02"

 ;; Even though its CHAR-NAME can vary, #\A prints as #\A
 (prin1-to-string (read-from-string (format nil "#\\~A" (or (char-name #\A) "A"))))
→ "#\\A"
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *character* is not a *character*.

See Also:

name-char, Section 22.1.3.2 (Printing Characters)

Notes:

Non-graphic characters having *names* are written by the *Lisp printer* as "`#\`" followed by the their *name*; see Section 22.1.3.2 (Printing Characters).

name-char *Function*

Syntax:

name-char *name* → *char-p*

Arguments and Values:

name—a *string designator*.

char-p—a *character* or **nil**.

Description:

Returns the *character object* whose *name* is *name* (as determined by **string-equal**—*i.e.*, lookup is not case sensitive). If such a *character* does not exist, **nil** is returned.

Examples:

```
(name-char 'space) → #\Space
(name-char "space") → #\Space
(name-char "Space") → #\Space
(let ((x (char-name #\a)))
  (or (not x) (eql (name-char x) #\a))) → true
```

Exceptional Situations:

Should signal an error of *type* **type-error** if *name* is not a *string designator*.

See Also:

char-name