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*.

The next figure 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

The next figure 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.