Extensions to Common LISP to Support International Character Sets

Michael Beckerle\textsuperscript{1} Paul Beiser\textsuperscript{2} Jerry Duggan\textsuperscript{3}
Robert Kerns\textsuperscript{4} Kevin Layer\textsuperscript{5} Thom Linden\textsuperscript{6}
Larry Masinter\textsuperscript{7} David Unietis\textsuperscript{8}

June 10, 1989

\textsuperscript{1}Gold Hill Computers
\textsuperscript{2}Hewlett-Packard
\textsuperscript{3}Hewlett-Packard
\textsuperscript{4}Independent consultant
\textsuperscript{5}Franz, Inc.
\textsuperscript{6}IBM Research, Subcommittee Chair
\textsuperscript{7}Xerox Research
\textsuperscript{8}Lucid, Inc.
# Contents

1 Introduction .................................. 2
   1.1 Objectives .................................. 2

2 Overview .................................... 4
   2.1 Character Identity .......................... 5
   2.2 Standard and Semi-Standard Characters ....... 7
   2.3 Hierarchy of Types ......................... 10
      2.3.1 Character Type ....................... 11
      2.3.2 String Type ......................... 12
   2.4 Character Naming .......................... 13
   2.5 Streams and System I/O ..................... 17
   2.6 Miscellaneous ............................ 20
Chapter 1

Introduction

This is a proposal to the X3 J13 committee for both extending and modifying the Common LISP language definition to provide a standard basis for Common LISP support of the variety of characters used to represent the languages of the international community.

This proposal was created by the Character Subcommittee of X3 J13. We would like to acknowledge discussions with T. Yuasa and other members of the JIS Technical Working Group, comments from members of X3 J13, and the proposals [Ida87], [Linden87], [Kerns87], and [Kurokawa88] for providing the motivation and direction for these extensions. As all these documents and discussions were created expressly for LISP standardization usage, we have borrowed freely from their ideas as well as the texts themselves.

1.1 Objectives

The major objectives of this proposal are:

- To provide a consistent, well-defined scheme allowing support of both very large character sets and multiple character sets. ¹

Many software applications are intended for international use, or have requirements for incorporation of language elements of multiple languages within a single application. Also, many applications require specialized languages including, for example, scientific and typesetting symbols. In order to ensure some portability of these applications, data expressed in a mixture of these languages must be treated uniformly by the software language.

¹The distinction between the terms character repertoire and coded character set is made later. The usage of the term character set, avoided after this introduction, encompasses both terms.
All character and string manipulations should operate uniformly, regardless of the character set(s) of the character objects. This applies to array indexing, readtable definitions, read symbol construction and I/O operations.

• To ensure efficient performance of string and character operations.

Many languages, such as Japanese and Chinese, use character sets which contain more characters than the Latin alphabet. Supporting larger sized character sets frequently means employing larger data fields to uniquely encode each character. Common LISP implementations using larger sized character sets can incur performance penalties in terms of space, time, or both.

The use of large and/or multiple character sets by an implementation implies the need for a more complex character type representation. Given a more complex character representation, the efficiency of language operations on characters (e.g. string operations) could be affected.

• To assure forward compatibility of the proposed model and definition with existing Common LISP implementations.

Developers should not be required to re-write large amounts of either LISP code or data representations in order to apply the proposed changes to existing implementations. The proposed changes should provide an easy portability path for existing code to many possible implementations.

There are a number of issues, some under the general rubric of internationalization, which this proposal does not cover. Among these issues are:

• Time and date formats
• Monetary formats
• Numeric punctuation
• Fonts
• Lexicographic orderings
• Right-to-left and bidirectional languages
Chapter 2

Overview

We use several terms within this document which are new in the context of Common LISP. Definitions for the following prominent terms are provided for the reader’s convenience.

A character repertoire defines a collection of characters independent of their specific rendered image or font. This corresponds to the mathematical notion of a set. Character repertoires are specified independent of coding and their characters are only identified with a unique character label, a graphic symbol, and a character description.

A coded character set is a character repertoire plus an encoding providing a unique mapping between each character and a number which serves as the character representation. There are numerous internationally standardized coded character sets; for example, [ISO 8859/1] and [ISO 646].

A character may be included in one or more character repertoires. Similarly, a character may be included in one or more coded character sets. For example, the Latin letter ”A” is contained in the coded character set standards: ISO 8859/1, ISO 8859/2, ISO 6937/2, and others.

To universally identify each character, we utilize a universal registry of characters which incorporates a collection of repertoires called character scripts as a partitioning of all characters. That is, each character is included in one and only one character script.

In Common LISP a character data object is identified by its character code, a unique numerical code. Each character code is composed from a character script and a character label.

Character data objects which are classified as graphic, or displayable, are each associated with a glyph. The glyph is the visual representation of the character.

1We avoid the term character set as it has been (over)used in the context of character repertoire as well as in the context of coded character set.

2The practical realization of this registry is the Draft ISO 10646 Coded Character Set Standard. [ISO DP 10646]
CHAPTER 2. OVERVIEW

character. All other character data objects are classified as non-graphic (or control).

The primary purpose of introducing these terms is to provide a consistent naming to Common LISP concepts which are related to those found in ISO standardization of coded character sets. They also serve as a demarcation between these standardization activities. For example, while Common LISP is free to define unique manipulation facilities for characters, character scripts and coded character sets, it should not define standard coded character sets nor standard character scripts.

A secondary purpose is to detach the language specification from underlying hardware representation. From a language specification viewpoint it is inconsequential whether characters occupy one or more (8-bit) bytes or whether a Common LISP implementation’s internal representation for characters is distinct from or identical to any of the numerous external representations (for example, the text interchange representation [ISO 6937/2]). We specifically do not propose any standard coded character sets.

A final purpose is to serve as a basis for terminology within the standard language specification.

Proposal 2.0.1 (Passed 03/89) The terminology introduced in this proposal will be included in the language specification at the discretion of the editor.

2.1 Character Identity

Characters are uniquely distinguished by their codes, which are drawn from the set of non-negative integers. That is, within Common LISP a unique numerical code is assigned to each semantically different character.

It is important to separate the notion of glyph from the notion of character data object when defining a scheme under which issues of identity can be rigorously decided by a computer language. Glyphs are the visual aspects of characters, writable on surfaces, and sometimes called ‘graphics’. A language specification valid for more than a narrow range of systems can only make assumptions about the existence of abstract glyphs (for example, the Latin letter A) and not about glyph variants (for example, the italicized Latin letter A) or characteristics of display devices.

The notion of attributes of character objects within Common LISP has proven to be either not used or not portable. The essential aspect of the following proposals is to what extent attributes continue to be supported by the language specifications.

Proposal 2.1.1 (Alternative A) (Passed as Modified 03/89) Remove all discussion of attributes from the language specification. Add the following discussion:

---

3The bibliography includes several relevant ISO coded character set standards.
Earlier versions of Common LISP incorporated font and bits as attributes of character objects. These and other supported attributes are considered implementation-defined attributes and if supported by an implementation effect the action of selected functions.

All types, constants and functions dealing with the bits and font attributes are either removed or modified as follows:

- **Modify char-=:** If two characters differ in any implementation-defined attributes, then they are not char=.

- **Modify char-<:** If two characters have identical implementation-defined attributes, then their ordering by char< is consistent with the numerical ordering by the predicate < on their code. (Similarly for char>, char>= and char<=.)

- **Modify char-equal:** The effect, if any, on char-equal of each implementation-defined attribute has to be specified as part of the definition of that attribute (and similarly for char-not-equal, char-lessp, char-greaterp, char-not-greaterp, char-not-lessp).

- **Modify char-upcase and char-downcase:** The effect of char-upcase and char-downcase is to preserve implementation-defined attributes.

- **Modify read:** It is implementation dependent which attributes are removed from symbol names. It is implementation dependent which attributes are removed from characters within double quotes.

- **Modify intern:** It is implementation dependent which implementation-defined attributes are removed.

- **Modify digit-char:** remove the optional font argument.

- **Modify code-char:** remove the optional font and bits arguments.

- **Remove char-font-limit**

- **Remove char-bits-limit**

- **Remove int-char**

- **Remove char-int**

- **Remove char-bits**

- **Remove char-font**

- **Remove make-char**

- **Remove char-control-bit**
• Remove char-meta-bit
• Remove char-super-bit
• Remove char-hyper-bit
• Remove char-bit
• Remove set-char-bit
• Remove string-char and string-char-p

• Modify readable: If implementation-defined attributes are supported, an implementation need not (but may) allow for such characters to have syntax descriptions in the readable. Otherwise, all characters are representable in the readable.

Proposal 2.1.2 (Alternative B) (Passed as Modified 03/89) This is identical to all of Alternative A (above) except that the function char-int is retained. char-int 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 independent of the particular “incarnation” or ”core image”.

With the elimination of font and bits from the specification the usefulness of char-code and code-char is diminished. They are no longer needed for constructing characters. The portable mechanisms for hashing are provided by char-int and sxhash.

In addition, using char-code-limit to iterate over characters is extremely inefficient in implementations that support large or user-defined repertoires.

Proposal 2.1.3 (Alternative C) (Failed 03/89) This an amendment to Alternative B (above).

• Remove char-code-limit
• Remove char-code
• Remove code-char

2.2 Standard and Semi-Standard Characters

The standard characters are the 96 characters used in the Common LISP definition or their equivalents.

This was the Common LISP [Steele84] definition, but equivalents is a vague term.
The standard characters are not defined by their glyphs, but by their roles within the language. There are two aspects to the roles of the standard characters: one is their role in reader and format control string syntax; the second is their role as components of the names of all Common LISP functions, macros, constants, and global variables. As long as an implementation chooses 96 glyphs and treats those 96 in a manner consistent with the language’s specification for the standard characters (e.g., the naming of functions), it doesn’t matter what glyphs the I/O hardware uses to represent those characters: they are the standard characters. Any program or data text written wholly in those characters is portable through simple code conversion.\footnote{For example, the currency glyph, $, might be replaced uniformly by the currency glyph available on a particular display.}

Additional mechanisms, such as in [Kurokawa88], which support establishment of equivalency between otherwise distinct characters are not excluded by this proposal.\footnote{We believe this is an important issue but it requires additional implementation experience. We also encourage new proposals from JIS and ISO LISP Working Groups on this issue.}

**Proposal 2.2.1** (Passed 03/89) The discussion of standard characters is replaced by the following:

*Common LISP requires all implementations to support a standard character subrepertoire. The Common LISP standard character subrepertoire consists of a newline \#\texttt{Newline}, the graphic space character \#\texttt{Space}, and the following additional ninety-four graphic characters or their equivalents.*\footnote{\#\texttt{Space} and \#\texttt{Newline} are omitted. Graphic labels and descriptions are from ISO 6937/2. The first letter of the graphic Id categorizes the character as follows: L - Latin, N - Numeric, S - Special.}
<table>
<thead>
<tr>
<th>Id</th>
<th>Glyph</th>
<th>Name or description</th>
<th>Id</th>
<th>Glyph</th>
<th>Name or description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA01</td>
<td>a</td>
<td>small a</td>
<td>ND01</td>
<td>1</td>
<td>digit 1</td>
</tr>
<tr>
<td>LA02</td>
<td>A</td>
<td>capital A</td>
<td>ND02</td>
<td>2</td>
<td>digit 2</td>
</tr>
<tr>
<td>LB01</td>
<td>b</td>
<td>small b</td>
<td>ND03</td>
<td>3</td>
<td>digit 3</td>
</tr>
<tr>
<td>LB02</td>
<td>B</td>
<td>capital B</td>
<td>ND04</td>
<td>4</td>
<td>digit 4</td>
</tr>
<tr>
<td>LC01</td>
<td>c</td>
<td>small c</td>
<td>ND05</td>
<td>5</td>
<td>digit 5</td>
</tr>
<tr>
<td>LC02</td>
<td>C</td>
<td>capital C</td>
<td>ND06</td>
<td>6</td>
<td>digit 6</td>
</tr>
<tr>
<td>LD01</td>
<td>d</td>
<td>small d</td>
<td>ND07</td>
<td>7</td>
<td>digit 7</td>
</tr>
<tr>
<td>LD02</td>
<td>D</td>
<td>capital D</td>
<td>ND08</td>
<td>8</td>
<td>digit 8</td>
</tr>
<tr>
<td>LE01</td>
<td>e</td>
<td>small e</td>
<td>ND09</td>
<td>9</td>
<td>digit 9</td>
</tr>
<tr>
<td>LE02</td>
<td>E</td>
<td>capital E</td>
<td>ND10</td>
<td>0</td>
<td>digit 0</td>
</tr>
<tr>
<td>LF01</td>
<td>f</td>
<td>small f</td>
<td>SC03</td>
<td>$</td>
<td>dollar sign</td>
</tr>
<tr>
<td>LF02</td>
<td>F</td>
<td>capital F</td>
<td>SP02</td>
<td>!</td>
<td>exclamation mark</td>
</tr>
<tr>
<td>LG01</td>
<td>g</td>
<td>small g</td>
<td>SP04</td>
<td>&quot;</td>
<td>quotation mark</td>
</tr>
<tr>
<td>LG02</td>
<td>G</td>
<td>capital G</td>
<td>SP05</td>
<td>’</td>
<td>apostrophe</td>
</tr>
<tr>
<td>LH01</td>
<td>h</td>
<td>small h</td>
<td>SP06</td>
<td>(</td>
<td>left parenthesis</td>
</tr>
<tr>
<td>LH02</td>
<td>H</td>
<td>capital H</td>
<td>SP07</td>
<td>)</td>
<td>right parenthesis</td>
</tr>
<tr>
<td>LI01</td>
<td>i</td>
<td>small i</td>
<td>SP08</td>
<td>,</td>
<td>comma</td>
</tr>
<tr>
<td>LI02</td>
<td>I</td>
<td>capital I</td>
<td>SP09</td>
<td>-</td>
<td>low line</td>
</tr>
<tr>
<td>LJ01</td>
<td>j</td>
<td>small j</td>
<td>SP10</td>
<td>-</td>
<td>hyphen or minus sign</td>
</tr>
<tr>
<td>LJ02</td>
<td>J</td>
<td>capital J</td>
<td>SP11</td>
<td>.</td>
<td>full stop, period</td>
</tr>
<tr>
<td>LK01</td>
<td>k</td>
<td>small k</td>
<td>SP12</td>
<td>/</td>
<td>solidus</td>
</tr>
<tr>
<td>LK02</td>
<td>K</td>
<td>capital K</td>
<td>SP13</td>
<td>:</td>
<td>colon</td>
</tr>
<tr>
<td>LL01</td>
<td>l</td>
<td>small l</td>
<td>SP14</td>
<td>;</td>
<td>semicolon</td>
</tr>
<tr>
<td>LL02</td>
<td>L</td>
<td>capital L</td>
<td>SP15</td>
<td>?</td>
<td>question mark</td>
</tr>
<tr>
<td>LM01</td>
<td>m</td>
<td>small m</td>
<td>SA01</td>
<td>+</td>
<td>plus sign</td>
</tr>
<tr>
<td>LM02</td>
<td>M</td>
<td>capital M</td>
<td>SA03</td>
<td>&lt;</td>
<td>less-than sign</td>
</tr>
<tr>
<td>LN01</td>
<td>n</td>
<td>small n</td>
<td>SA04</td>
<td>=</td>
<td>equals sign</td>
</tr>
<tr>
<td>LN02</td>
<td>N</td>
<td>capital N</td>
<td>SA05</td>
<td>&gt;</td>
<td>greater-than sign</td>
</tr>
<tr>
<td>LO01</td>
<td>o</td>
<td>small o</td>
<td>SM01</td>
<td>#</td>
<td>number sign</td>
</tr>
<tr>
<td>LO02</td>
<td>O</td>
<td>capital O</td>
<td>SM02</td>
<td>%</td>
<td>percent sign</td>
</tr>
<tr>
<td>LP01</td>
<td>p</td>
<td>small p</td>
<td>SM03</td>
<td>&amp;</td>
<td>ampersand</td>
</tr>
<tr>
<td>LP02</td>
<td>P</td>
<td>capital P</td>
<td>SM04</td>
<td>*</td>
<td>asterisk</td>
</tr>
<tr>
<td>LQ01</td>
<td>q</td>
<td>small q</td>
<td>SM05</td>
<td>@</td>
<td>commercial at</td>
</tr>
<tr>
<td>LQ02</td>
<td>Q</td>
<td>capital Q</td>
<td>SM06</td>
<td>[</td>
<td>left square bracket</td>
</tr>
<tr>
<td>LR01</td>
<td>r</td>
<td>small r</td>
<td>SM07</td>
<td>\</td>
<td>reverse solidus</td>
</tr>
<tr>
<td>LR02</td>
<td>R</td>
<td>capital R</td>
<td>SM08</td>
<td>]</td>
<td>right square bracket</td>
</tr>
<tr>
<td>LS01</td>
<td>s</td>
<td>small s</td>
<td>SM11</td>
<td>{</td>
<td>left curly bracket</td>
</tr>
<tr>
<td>LS02</td>
<td>S</td>
<td>capital S</td>
<td>SM13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LT01</td>
<td>t</td>
<td>small t</td>
<td>SM14</td>
<td>}</td>
<td>right curly bracket</td>
</tr>
<tr>
<td>LT02</td>
<td>T</td>
<td>capital T</td>
<td>SD13</td>
<td>*</td>
<td>grave accent</td>
</tr>
<tr>
<td>LU01</td>
<td>u</td>
<td>small u</td>
<td>SD15</td>
<td>-</td>
<td>circumflex accent</td>
</tr>
<tr>
<td>LU02</td>
<td>U</td>
<td>capital U</td>
<td>SD19</td>
<td>~</td>
<td>tilde</td>
</tr>
<tr>
<td>LV01</td>
<td>v</td>
<td>small v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LV02</td>
<td>V</td>
<td>capital V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LW01</td>
<td>w</td>
<td>small w</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LW02</td>
<td>W</td>
<td>capital W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LX01</td>
<td>x</td>
<td>small x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LX02</td>
<td>X</td>
<td>capital X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LY01</td>
<td>y</td>
<td>small y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LY02</td>
<td>Y</td>
<td>capital Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LZ01</td>
<td>z</td>
<td>small z</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LZ02</td>
<td>Z</td>
<td>capital Z</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The definition of semi-standard characters has been of minimum practical use since implementations may or may not support any of these characters. The essential feature is that, when supported, they have a predictable treatment by the reader.

**Proposal 2.2.2** (Failed 03/89) Remove all discussion of semi-standard characters. Add that in implementations supporting non-graphic characters other than `#\Newline`, the `read` function is required to treat those as whitespace characters.

### 2.3 Hierarchy of Types

Providing support for extensive character repertoires may impact Common LISP implementation performance in terms of space, time, or both.  \[^{7}\] In particular, many existing implementations support variants of the ISO 8859/1 standard. Supporting large repertoires argues for a multi-byte internal representation for each character, even if an application primarily (or exclusively) uses the ISO 8859/1 characters.

This proposal extends the definition of the character and string type hierarchy to allow specialized subtypes of character and string. An implementation is free to associate compact internal representation tailored to each subtype. The `string` type specifier, when used for object creation, for example in `make-sequence`, is defined to mean the most general string subtype supported by the implementation (similarly for the `simple-string` type specifier). This definition emphasizes portability of existing Common LISP applications to international character environments over performance. Applications emphasizing efficiency of text processing in non-international environments will require some modification to utilize subtypes with compact internal representations.

It has been suggested that either a single type is sufficient to support international characters, or that a hierarchy of types could be used, in a manner transparent to the user. A desire to provide flexibility which encourages implementations to support international characters without compromising application efficiency led us to accept the need for more than one type. We believe that these choices reflect a minimal modification of this aspect of the type system, and that exposing the types for string and character construction while requiring uniform treatment for characters otherwise is the most reasonable approach.

[^7]: This does not apply to all implementations. Unique hardware support and user community requirements need to be taken into consideration.
2.3.1 Character Type

Proposal 2.3.1 *(Passed as Modified 03/89)* Define base-character as (upgraded-array-element-type 'standard-char) and extended-character as type (and character (not base-character)).

Characters of type base-character are referred to as base characters. Characters of type extended-character are referred to as extended characters.

This establishes the relationship between the string encoding and array upgrading strategies of the implementation and the important character types.

An implementation may support additional subtypes of character which may or may not be supertypes of base-character. In addition, an implementation may define base-character as equivalent to character.

The base characters are distinguished in the following respects:

- The standard characters are a subrepertoire of the base characters.
- The selection of base characters which are not standard characters is implementation defined.
- Only members of the base character repertoire can be elements of a base string.
- No upper bound is specified for the number of glyphs in the base character repertoire—that is implementation dependent. The lower bound is 96, the number of standard characters defined for Common LISP.  

The distinction of base characters is largely a pragmatic choice. It permits efficient handling of common situations, may be privileged for host system I/O, and can serve as an intermediate basis for portability, less general than the standard characters, but possibly more useful across a narrower range of implementations.

Many computers have some "base" character representation which is a function of hardware instructions for dealing with characters, as well as the organization of the file system. The base character representation is likely to be the smallest transaction unit permitted for text file and terminal I/O operations. On a system with a record based I/O paradigm, the base character representation is likely to be the smallest record quantum. On many computer systems, this representation is a byte.

However, the proposal emphasizes that whether a character is "base" to Common LISP depends on the way that an implementation represents strings, and not any other properties of the implementation or the host operating system. Imagine two implementations, one of which encodes all strings as 16-bit characters, and another which has two kinds of strings: 8-bit strings and 16-bit strings. In the first implementation, the base-character is character: there's only one kind of string. In the second implementation, the base-character

---

8Or, in contrast, the base repertoire may include all implementation supported characters.
would be those that could be stored in an 8-bit string, and it would be a proper sub-type of character.

2.3.2 String Type

Proposal 2.3.2 (Passed 03/89) The string type is defined as a union type. More precisely, a string is a specialized vector whose elements are of type character or a subtype of character. string used as a type specifier for object creation means (vector character).

Proposal 2.3.3 (Passed as Modified 03/89) The following string subtypes are distinguished with standardized names.

- base-string is equivalent to (vector base-character). Strings of type base-string are referred to as base strings.
- base-string is valid as a type specifier that abbreviates.

Proposal 2.3.4 (Passed as Modified 03/89) Define simple-string as a union type. A simple string is a specialized simple one dimensional array whose elements are of type character or a subtype of character. simple-string used as a type specifier for object creation means (simple-array character (size)).

Proposal 2.3.5 (Passed as Modified 03/89) The following simple string subtypes are distinguished with standardized names:

- simple-base-string is equivalent to (simple-array base-character (*)). simple-base-string is a subtype of base-string.
- simple-base-string is valid as a type specifier that abbreviates.

A base string is the most efficient string which can hold the standard characters.

All Common LISP functions defined to operate on strings treat all strings uniformly with the following caveat: for any function which inserts a character into a string, it is an error to insert an extended character into a base string. An implementation may support string subtypes in addition to base-string.

For example, a hypothetical implementation supporting Arabic and Cyrillic characters might provide as extended characters:

- string – may contain Arabic, Cyrillic or base characters in any mixture.
- region-specialized-string – may contain installation selected repertoire (Arabic/Cyrillic) or base characters in any mixture.

An implementation may, optionally, provide automatic coercion to an extended string.

9 An implementation may, optionally, provide automatic coercion to an extended string.
• base-string – may contain base characters

Though, clearly, portability of applications using region-specialized-string is limited, a performance advantage might argue for its use.  

Alternatively, an implementation supporting a large base character repertoire including, say, Japanese Kanji may define base-character as equivalent to character.

We expect that applications sensitive to the performance of character handling in some host environments will utilize the string subtypes to provide performance improvement. Applications with emphasis on international portability will likely utilize only string.

The base string type allows for more compact representation of strings of base characters, which are likely to predominate in any system. Note that in any particular implementation the base characters need not be the most compactly representable, since others might have a smaller repertoire. However, in most implementations base strings are likely to be more space efficient than extended strings.

Proposal 2.3.6 (Passed 03/89) Extend the make-string function to allow an element-type keyword argument:

• make-string size &key :initial-element :element-type [Function]

This returns a simple string of length size, each of whose characters has been initialized to the :initial-element argument. If an :initial-element argument is not specified, then the string will be initialized in an implementation-dependent way. The :element-type argument names the type of the elements of the string; a string is constructed of the most specialized type that can accommodate elements of the given type. If :element-type is omitted, the type character is the default.

2.4 Character Naming

A Common LISP program should be able to name, compose and decompose characters in a uniform, portable manner, independent of any underlying representation. One possible composition is by the pair < coded character set standard, decimal representation >. Thus, for example, one might compose the Latin 'A' with the pair < ISO8859/2-1987, 65 >, < ISO8859/6-1987, 65 >, or < ISO646-1983, 65 >, etc. The difficulty here is two-fold. First, there are several ways to compose the same character and second, there may be multiple answers to the question: To what coded character set does character object

\textit{region-specialized-string} is used here for illustration only; it is not being proposed as a standardized string subtype.

\textit{This syntax is for illustration only and is not being proposed.}
The identical problems occur if the pair < character repertoire standard, decimal representation > is used.

The concept of character registry is introduced by this proposal to resolve the problem of character naming, composition and decomposition. Each character is universally defined by the pair < character script, character label >. For this to be a portable definition, it must have a standard meaning. Thus we propose the formation of an ISO Working Group to define an international Character Registry Standard. At this writing there is no existing Character Registry Standard nor ISO Working Group organized to define such a standard.

Proposal 2.4.1 (Passed 03/89) Common LISP character codes are composed from a character script and a character label. The convention by which a character label and character script compose a character code is implementation dependent.

The naming and content of the standard character scripts is left unspecified by this proposal. Below are some candidate character script names:

- latin
- extended-latin
- international-african-alphabet
- extended-symbols
- diacritics
- cyrillic-for-major-languages
- cyrillic-for-minor-languages
- greek
- arabic
- armenian
- georgian

---

12 Even worse, the answer might change yearly.

13 Existing ISO repertoires seem to be defined exclusively in the context of coded character sets and not as standards in their own right.

14 It is the intention of X3 J13 to promote and adopt an eventual ANSI or ISO Character Registry Standard. In particular, we acknowledge that X3 J13 is not the appropriate forum to define the standard. We believe it is a required component of all programming languages providing support for international characters.

15 The only constraint is that character scripts and labels be named using only the Latin capital letters A-Z, hyphen and digits 0-9.
• hebrew
• hiragana-symbols
• katakana
• control (meaning the collection of standard text communication control codes)

The list above is provided as a starting point for discussion and is not intended to be representative nor exhaustive. The Common LISP language definition does not depend on these names nor any specific content (for example: Where should the plus sign appear?). It is application programs which require a reliable definition of the script names and their constituents. The Common LISP language definition imposes the framework for constructing and manipulating character objects.

Proposal 2.4.2 Standardized Character Scripts are fixed; an implementation may not extend a standard script’s constituent set of characters beyond the standard definition.

An implementation may provide support for all or part of any character script and may provide new character scripts which include characters having unique semantics (i.e. not defined in any standard character script). Implementation scripts must be uniquely named using only Latin capital letters A-Z, hyphen and digits 0-9.

An implementation must document the scripts it supports. For each script supported the documentation must include 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. ¹⁷

• Effect of character predicates. In particular,

  – alpha-char-p
  – lower-case-p
  – upper-case-p
  – both-case-p
  – graphic-char-p
  – alphanumeri-p

¹⁶In fact, they are simply 15 of the scripts represented within [ISO DP 10646]
¹⁷Any mechanisms by which the read function treats distinct characters as equivalent.
• Interaction with File I/O. In particular, the coded character sets and external encoding schemes supported are documented.

We introduce new functions to compose and decompose character objects. We also extend the characterp predicate to support testing membership of a character in a given character repertoire. A global variable *all-character-script-names* is added to allow application determination of implementation supported character scripts.

Proposal 2.4.3 Add the type specifier and (modified) type predicate:

• (character repertoire)
  This denotes a character type specialized to members of the specified repertoire. Repertoire may be :base or :standard or any supported character repertoire name (a symbol), or a list of names.

  (character :base) is equivalent to base-character and (character :standard) is equivalent to standard-character.

• (characterp object &optional repertoire)
  If repertoire is omitted, characterp is true if object is a character object, and otherwise is false. If a repertoire argument is specified, characterp is true if object is a character object and a member of the specified repertoire, and otherwise is false. Repertoire may be any supported character repertoire name (a symbol) or the names :base or :standard. (characterp x :standard) is equivalent to (standard-char-p x). (characterp x :base) is true if x is a member of the base character repertoire.

Proposal 2.4.4 Add the following variable and functions:

• *all-character-script-names* [Variable]
  The value of *all-character-script-names* is a list of all character repertoire names (symbols) supported by the implementation.

• char-label char [Function]
  char-label returns a string representing the character label of char. It is an error if the argument is not a character object.

• char-script-name char [Function]
  char-script-name returns a string representing the character script to which char belongs. It is an error if the argument is not a character object.

---

18 For example, ISO8859/1-1987.
19 For example, testing membership in the Japanese Katakana character repertoire.
• find-char script label [Function]

find-char returns a character object. The arguments script and label are names (symbols) of a character script and label. label uniquely identifies a character within the character script named script. If the implementation does not support the specified character, nil is returned.

Proposal 2.4.5 Character names accepted and constructed by char-name, name-char, and \#\ are extended to include character script names of the form script:label.

2.5 Streams and System I/O

A lot of the work of ensuring that a Common LISP implementation operates correctly in a multiple coded character set environment must be performed by the I/O interface. The system I/O interface, abstracted in Common LISP as streams, is responsible for ensuring that text input from outside LISP is properly mapped into character objects internally, and that the inverse mapping is performed on output. It is beyond the scope of a language definition to specify the details of this operation, but options are specified which allow runtime indication from the user as to what coded character sets a stream uses, and how the mappings should be done. It is expected that implementations will provide reasonable defaults and invocation options to accommodate desired use at an installation.

There are often multiple coded character sets supportable on a computer, through the use of special display and entry hardware, which are varying interpretations of the basic system character representation. For example, ISO 8859/1 and ISO 6937/2 are two different interpretations of the same 1-byte code representations. Many countries have their own glyph-to-code mappings for 1-byte character codes addressing the special requirements of national languages. Differentiating between these, without reference to display hardware, is a matter of convention, since they all use the same set of code representations. When a single byte is not enough, two or more bytes are sometimes used for character encoding. This makes character handling even more difficult on machines where the natural representation size is a byte, since not only is the semantic value of a character code a matter of convention, which may vary within the same computing system, but so is the identification of a set of bits as a complete character code.

Given that multiple coded character sets exist, it is useful to provide portable mechanisms based on their definitions.

Proposal 2.5.1 Add the following functions:

• char-external-code char name [Function]

char-external-code returns the non-negative integer representing the encoding of the character char in the coded character set named by name, a
symbol. If the implementation does not support the specified coded character set, \texttt{nil} is returned. If the named coded character set does not contain the character, \texttt{nil} is returned.

- \texttt{find-external-char name index [Function]}

\texttt{find-external-char} returns a character object. The argument index is a non-negative integer representing the encoding of a character in the coded character set named by name, a symbol. If the implementation does not support the specified coded character set, \texttt{nil} is returned. If the named coded character set does not contain the character, \texttt{nil} is returned.

An implementation supporting multiple coded character sets must allow for the external representation of characters to be separately (and perhaps multiply) specified to \texttt{open}, since there can be circumstances under which more than one external representation for characters is in use, or more than one coded character set is mixed together in an external representation convention.

Which coded character sets and encoding schemes are supported by the overall computing system and the details of the mapping of glyphs to characters to character codes are left unspecified by Common LISP.

**Proposal 2.5.2** Add the additional keyword argument to \texttt{open}:

- :\texttt{external-format} which specifies a name, or list of names (keyword symbols) indicating an implementation recognized scheme for representing 1 or more coded character sets with non-homogeneous codes.

  The default value is :\texttt{default} and is implementation defined but must include the base characters.

  As many coded character set names must be provided as the implementation requires for that external coding convention.

  Coded character set names must include the full reference number and approval year. For example, :ISO8859P1V1987 and :ISO6937P2V1983. All implementation recognized schemes are formed from the Latin uppercase A-Z, hyphen, and digit 0-9 characters.

  This argument is provided for input, output, and bidirectional streams. It is an error to try to write a character other than a member of the specified coded character sets to a stream. (This excludes the \#\texttt{Newline} character. Implementations must provide appropriate line division behavior for all character streams.)

The existing default for the :\texttt{element-type} argument of \texttt{open} is \texttt{string-char}. This is no longer appropriate given the elimination of \texttt{string-char} within the standard specification.
Proposal 2.5.3 (Withdrawn 03/89) Modify the :element-type argument to open as follows:

- Add base-character as a valid type.
- Remove string-char as a valid type.

The following alternative is consistent with the general premise that portability is emphasized over efficiency.

Proposal 2.5.4 (Alternative A) The default for the :element-type argument of open is character.

The following alternative (B), allows implementations to match the behavior of open to the expected behavior of their file systems.

Proposal 2.5.5 (Alternative B) The default for the :element-type argument of open is implementation defined as a super-type of base-character and a sub-type of character.

Proposal 2.5.6 Modify the following functions:

- with-output-to-string if no string argument is provided, produces a stream that accepts all characters and returns a string of the most specialized type that accommodates the characters that were actually output.

- make-string-output-stream produces a stream that accepts all characters and returns (via get-output-stream-string) a string of the most specialized type that accommodates the characters that were actually output.

In addition to supporting conversion at the system interface, the language must allow user programs to determine how much space data objects will require when output in whichever external representations are available.

This function is necessary to determine if strings can be written to fixed length fields in databases. Note that this function does not address the problem of calculating screen width of strings printed in proportional fonts.

Proposal 2.5.7 Add the following function:

- file-string-length file-stream object [Function]

  file-string-length returns a non-negative integer which represents the difference between what (file-position file-stream) would be after writing the object and its current value, or nil if this cannot be determined. object must be a string or character.

  This integer corresponds to the current state of the stream and may change if there has been intervening output.
CHAPTER 2. OVERVIEW

2.6 Miscellaneous

In the process of creating this document, some comments were found within CLtL which seem appropriate to modify independently of the other proposals mentioned previously. For each, we identify the existing statement of CLtL and the recommended change.

Proposal 2.6.1 (Passed 03/89)
⇒‡(p12) Chapter 2 Data Types
replace provides for a rich character set, including ways to represent characters of various type styles.
with provides support for international language characters as well as characters used in specialized arenas, eg. mathematics.

Proposal 2.6.2 (Passed as Modified 03/89)
⇒‡(p25) Chapter 2 Symbols
clarify A symbol may have any character in its print name.

Proposal 2.6.3 (Passed 03/89)
⇒‡(p163) Chapter 10 Symbols
replace It is ordinarily not permitted to alter a symbol’s print name.
with It is an error to alter a symbol’s print name.

Proposal 2.6.4 (Passed 03/89)
⇒‡(p168) Chapter 10 The Print Name
replace It is an extremely bad idea to modify a string being used as the print name of a symbol.
with It is an error to modify a string being used as the print name of a symbol.

Proposal 2.6.5 (Passed 03/89)
⇒‡(p249,make-sequence) Chapter 14 Simple Sequence Functions
append If type string is specified, the result is equivalent to make-string.
Bibliography


