

## Characters and Strings

- Characters
  - Characters in Scheme
  - Collating Sequences
  - Handling Case
  - More Character Predicates
- Strings
  - String Procedures
- Appendix: Representing Characters
  - ASCII
  - Unicode

A *character* is a small, repeatable unit within some system of writing -- a letter or a punctuation mark, if the system is alphabetic, or an ideogram in a writing system like Han (Chinese). A string is a sequence of characters. Unlike symbols, which are *atomic*, strings can be separated into constituent parts.

## Characters

### Characters in Scheme

As you might expect, Scheme needs a way to distinguish between many different but similar things, including: characters (these units of writing), strings (formed by combining characters), symbols (which are treated as atomic and also cannot be combined), and variables. Similarly, Scheme needs to distinguish between numbers (which you can compute with) and characters (which you can put in strings).

In Scheme, a name for any of the text characters can be formed by writing `#\` before that character. For instance, the expression `#\A` denotes the capital A, the expression `#\3` denotes the character 3 (to be distinguished from the number 3) and the expression `#\?` denotes the question mark. (Control characters, however, usually cannot be named in this way.) In addition, the expression `#\space` denotes the space character, and `#\newline` denotes the newline character (the one that is used to terminate lines of text files stored on Unix systems).

### Collating Sequences

In any implementation of Scheme, it is assumed that the available characters can be arranged in a linear order (the “collating sequence” for the character set), and each character is associated with an integer that specifies its position in that order. In ASCII, the numbers that are associated with characters run from 0 to 127; in Unicode, they lie within the range from 0 to 65535. (Fortunately, Unicode includes all of the ASCII characters and associates with each one the same collating-sequence number that ASCII uses.) Applying the built-in `char->integer` procedure to a character gives you the collating-sequence number for that character; applying the converse procedure, `integer->char`, to an integer in the appropriate range gives you the character that has that collating-sequence number.

The importance of the collating-sequence numbers is that they extend the notion of alphabetical order to all the characters. Scheme provides five built-in predicates for comparing characters (`char<?`, `char<=?`, `char=?`, `char>=?`, and `char>?`). They all work by determining which of the two characters comes first in the collating sequence (that is, which one has the lower collating-sequence number).

Scheme requires that if you compare two capital letters or two lower-case letters, you'll get standard alphabetical order: (`char<? #\A #\Z`) must be true, for instance. If you compare a capital letter with a lower-case letter, though, the result depends on the design of the character set.<sup>1</sup> Similarly, if you compare two digit characters, Scheme guarantees that the results will be consistent with numerical order: `#\0` precedes `#\1`, which precedes `#\2`, and so on. But if you compare a digit with a letter, or anything with a punctuation mark, the results depend on the character set.

## Handling Case

Because there are many applications in which it is helpful to ignore the distinction between a capital letter and its lower-case equivalent in comparisons, Scheme also provides *case-insensitive* versions of the comparison procedures: `char-ci<?`, `char-ci<=?`, `char-ci=?`, `char-ci>=?`, and `char-ci>?`. These procedures essentially convert all letters to the same case (in DrScheme, upper case) before comparing them.

There are also two procedures for converting case.

- If its argument is a lower-case letter, `char-upcase` returns the corresponding capital letter; otherwise, it returns the argument unchanged.
- If its argument is a capital letter, `char-downcase` returns the corresponding lower-case letter; otherwise, it returns the argument unchanged.

## More Character Predicates

Scheme provides several one-argument predicates that apply to characters:

- `char-alphabetic?` determines whether its argument is a letter (`#\a` through `#\z` or `#\A` through `#\Z`).
- `char-numeric?` determines whether its argument is a digit character (`#\0` through `#\9`).
- `char-whitespace?` determines whether its argument is a “whitespace character” -- one that is conventionally stored in a text file primarily to position text legibly. In ASCII, the whitespace characters are the space character and four specific control characters: `<Control/I>` (tab), `<Control/J>` (line feed), `<Control/L>` (form feed), and `<Control/M>` (carriage return). On most systems, `#\newline` is a whitespace character. On our Linux systems, `#\newline` is the same as `<Control/J>` and so counts as a whitespace character.
- `char-upper-case?` determines whether its argument is a capital letter.
- `char-lower-case?` determines whether its argument is a lower-case letter.

It may seem that it's easy to implement some of these operations. For example, you might want to implement `char-alphabetic?` as

A character is alphabetic if it is between #\a through #\z or between #\A through #\Z

which translates to

```
(define is-letter?
  (lambda (c)
    (or (char<=? #\a c #\z)
        (char<=? #\A c #\Z))))
> (is-letter? #\a)
#t
> (is-letter? #\?)
#f
> (is-letter? #\Q)
#t
```

However, that implementation is not necessarily correct for all versions of Scheme: Since Scheme does not guarantee that the letters are collated without gaps, it's possible that this implementation of `char-alphabetic?` treats some non-letters as letters. The alternative, comparing to each valid letter in turn, seems inefficient. By making this procedure built-in, the designers of Scheme have encouraged programmers to rely on a correct (and, presumably, efficient) implementation.

Note that all of these predicates assume that their parameter is a character. Hence, if you don't know the type of a parameter, you need to use a more complex test, like

```
(if (and (char? x) (char-whitespace? x)
        ...)
```

## Strings

A string is a sequence of zero or more characters. Most strings can be named by enclosing the characters they contain between plain double quotation marks, to produce a *string literal*: for instance, "hyperbola" is the nine-character string consisting of the characters #\h, #\y, #\p, #\e, #\r, #\b, #\o, #\l, and #\a, in that order, and "" is the zero-character string (the *null string*).

String literals may contain spaces and newline characters; when such characters are between double quotation marks, they are treated like any other characters in the string. There is a slight problem when one wants to put a double quotation mark into a string literal: To indicate that the double quotation mark is part of the string (rather than marking the end of the string), one must place a backslash character immediately in front of it. For instance, "Say \"hi\"" is the eight-character string consisting of the characters #\S, #\a, #\y, #\space, #\", #\h, #\i, and #\", in that order. The backslash before a double quotation mark in a string literal is an *escape* character, present only to indicate that the character immediately following it is part of the string.

This use of the backslash character causes yet another slight problem: What if one wants to put a backslash into a string? The solution is to place another backslash character immediately in front of it. For instance, "a\\b" is the three-character string consisting of the characters #\a, #\\, and #\b, in that order. The first backslash in the string literal is an escape, and the second is the character that it protects, the one that is part of the string.

## String Procedures

Scheme provides several basic procedures for working with strings:

The `string?` predicate determines whether its argument is or is not a string.

The `make-string` procedure constructs and returns a string that consists of repetitions of a single character. Its first argument indicates how long the string should be, and the second argument specifies which character it should be made of. For instance,

```
> (make-string 5 #\a)
"aaaaa"
```

constructs and returns the string "aaaaa".

The `string` procedure takes any number of characters as arguments and constructs and returns a string consisting of exactly those characters. For instance, `(string #\H #\i #\!)` constructs and returns the string "Hi!".

The `string->list` and `list->string` procedures do just what you'd expect.

The `string-length` procedure takes any string as argument and returns the number of characters in that string. For instance, the value of `(string-length "parabola")` is 8 and the value of `(string-length "a\\b")` is 3.

The `string-ref` procedure is used to select the character at a specified position within a string. Like `list-ref`, `string-ref` presupposes *zero-based indexing*; the position is specified by the number of characters that precede it in the string. (So the first character in the string is at position 0, the second at position 1, and so on.) For instance, the value of `(string-ref "ellipse" 4)` is `#\p` -- the character that follows four other characters and so is at position 4 in zero-based indexing.

Strings can be compared for "lexicographic order," the extension of alphabetical order that is derived from the collating sequence of the local character set. Once more, Scheme provides both case-sensitive and case-insensitive versions of these predicates: `string<?`, `string<=?`, `string=?`, `string>=?`, and `string>?` are the case-sensitive versions, and `string-ci<?`, `string-ci<=?`, `string-ci=?`, `string-ci>=?`, and `string-ci>?` the case-insensitive ones.

The `substring` procedure takes three arguments. The first is a string and the second and third are non-negative integers not exceeding the length of that string. `substring` returns the part of its first argument that starts after the number of characters specified by the second argument and ends after the number of characters specified by the third argument. For instance: `(substring "hypocycloid" 3 8)` returns the substring "ocycl" -- the substring that starts after the initial "hyp" and ends after the eighth character, the l.

The `string-append` procedure takes any number of strings as arguments and returns a string formed by concatenating those arguments. For instance, the value of `(string-append "al" "fal" "fa")` is "alfalfa".

## Appendix: Representing Characters

When a character is stored in a computer, it must be represented as a sequence of *bits* -- “binary digits,” that is, zeroes and ones. However, the choice of a particular bit sequence to represent a particular character is more or less arbitrary. In the early days of computing, each equipment manufacturer developed one or more “character codes” of its own, so that, for example, the capital letter A was represented by the sequence 110001 on an IBM 1401 computer, by 000001 on a Control Data 6600, by 11000001 on an IBM 360, and so on. This made it troublesome to transfer character data from one computer to another, since it was necessary to convert each character from the source machine’s encoding to the target machine’s encoding. The difficulty was compounded by the fact that different manufacturers supported different characters; all provided the twenty-six capital letters used in writing English and the ten digits used in writing Arabic numerals, but there was much variation in the selection of mathematical symbols, punctuation marks, etc.

### ASCII

In 1963, a number of manufacturers agreed to use the American Standard Code for Information Interchange (ASCII), which is currently the most common and widely used character code. It includes representations for ninety-four characters selected from American and Western European text, commercial, and technical scripts: the twenty-six English letters in both upper and lower case, the ten digits, and a miscellaneous selection of punctuation marks, mathematical symbols, commercial symbols, and diacritical marks. (These ninety-four characters are the ones that can be generated by using the forty-seven lighter-colored keys in the typewriter-like part of a MathLAN workstation’s keyboard, with or without the simultaneous use of the <Shift> key.) ASCII also reserves a bit sequence for a “space” character, and thirty-three bit sequences for so-called *control characters*, which have various implementation-dependent effects on printing and display devices -- the “newline” character that drops the cursor or printing head to the next line, the “bell” or “alert” character that causes the workstation to beep briefly, and such like.

In ASCII, each character or control character is represented by a sequence of exactly seven bits, and every sequence of seven bits represents a different character or control character. There are therefore  $2^7$  (that is, 128) ASCII characters altogether.

### Unicode

Over the last quarter-century, non-English-speaking computer users have grown increasingly impatient with the fact that ASCII does not provide many of the characters that are essential in writing other languages. A more recently devised character code, the Unicode Worldwide Character Standard, currently defines bit sequences for 49194 characters for the Arabic, Armenian, Bengali, Bopomofo, Canadian Syllabics, Cherokee, Cyrillic, Devanagari, Ethiopic, Georgian, Greek, Gujarati, Gurmukhi, Han, Hangul, Hebrew, Hiragana, Kannada, Katakana, Khmer, Latin, Lao, Malayalam, Mongolian, Myanmar, Ogham, Oriya, Runic, Sinhala, Tamil, Telugu, Thaana, Thai, Tibetan, and Yi writing systems, as well as a large number of miscellaneous numerical, mathematical, musical, astronomical, religious, technical, and printers’ symbols, components of diagrams, and geometric shapes.

Unicode uses a sequence of sixteen bits for each character, allowing for  $2^{16}$  (that is, 65536) codes altogether. Many bit sequences are still unassigned and may, in future versions of Unicode, be allocated for some of the numerous writing systems that are not yet supported. The designers have completed work on the Deseret, Etruscan, and Gothic writing systems, although they have not yet been added to the Unicode standard. Characters for the Shavian, Linear B, Cypriot, Tagalog, Hanunóo, Buhid, Tagbanwa, Cham, Tai, Glagolitic, Coptic, Buginese, Old Hungarian Runic, Phoenician, Avestan, Tifinagh, Javanese, Rong, Egyptian Hieroglyphic, Meroitic, Old Persian Cuneiform, Ugaritic Cuneiform, Tengwar, Cirth, tlhIngan Hol (i.e., ‘‘Klingon<sup>2</sup>’’), Brahmi, Old Permic, Sinaitic, South Arabian, Pollard, Blissymbolics, and Soyombo writing systems are under consideration or in preparation.

Although our local Scheme implementations use and presuppose the ASCII character set, the Scheme language does not require this, and Scheme programmers should try to write their programs in such a way that they could easily be adapted for use with other character sets (particularly Unicode).

---

### Endnotes

<sup>1</sup> In ASCII, every capital letter -- even #\Z -- precedes every lower-case letter -- even #\a.

<sup>2</sup> Can you tell that CS folks are geeks?

---

### Endnotes

<sup>1</sup> In ASCII, every capital letter -- even #\Z -- precedes every lower-case letter -- even #\a.

<sup>2</sup> Can you tell that CS folks are geeks?