

NAME

perlunicode - Unicode support in Perl

DESCRIPTION

Important Caveats

Unicode support is an extensive requirement. While Perl does not implement the Unicode standard or the accompanying technical reports from cover to cover, Perl does support many Unicode features.

People who want to learn to use Unicode in Perl, should probably read *the Perl Unicode tutorial*, *perlunitut*, before reading this reference document.

Input and Output Layers

Perl knows when a filehandle uses Perl's internal Unicode encodings (UTF-8, or UTF-EBCDIC if in EBCDIC) if the filehandle is opened with the `":utf8"` layer. Other encodings can be converted to Perl's encoding on input or from Perl's encoding on output by use of the `":encoding(...)"` layer. See *open*.

To indicate that Perl source itself is in UTF-8, use `use utf8;`.

Regular Expressions

The regular expression compiler produces polymorphic opcodes. That is, the pattern adapts to the data and automatically switches to the Unicode character scheme when presented with data that is internally encoded in UTF-8, or instead uses a traditional byte scheme when presented with byte data.

`use utf8` still needed to enable UTF-8/UTF-EBCDIC in scripts

As a compatibility measure, the `use utf8` pragma must be explicitly included to enable recognition of UTF-8 in the Perl scripts themselves (in string or regular expression literals, or in identifier names) on ASCII-based machines or to recognize UTF-EBCDIC on EBCDIC-based machines. **These are the only times when an explicit `use utf8` is needed.** See *utf8*.

BOM-marked scripts and UTF-16 scripts autodetected

If a Perl script begins marked with the Unicode BOM (UTF-16LE, UTF16-BE, or UTF-8), or if the script looks like non-BOM-marked UTF-16 of either endianness, Perl will correctly read in the script as Unicode. (BOMless UTF-8 cannot be effectively recognized or differentiated from ISO 8859-1 or other eight-bit encodings.)

`use encoding` needed to upgrade non-Latin-1 byte strings

By default, there is a fundamental asymmetry in Perl's Unicode model: implicit upgrading from byte strings to Unicode strings assumes that they were encoded in *ISO 8859-1 (Latin-1)*, but Unicode strings are downgraded with UTF-8 encoding. This happens because the first 256 codepoints in Unicode happens to agree with Latin-1.

See *Byte and Character Semantics* for more details.

Byte and Character Semantics

Beginning with version 5.6, Perl uses logically-wide characters to represent strings internally.

In future, Perl-level operations will be expected to work with characters rather than bytes.

However, as an interim compatibility measure, Perl aims to provide a safe migration path from byte semantics to character semantics for programs. For operations where Perl can unambiguously decide that the input data are characters, Perl switches to character semantics. For operations where this determination cannot be made without additional information from the user, Perl decides in favor of compatibility and chooses to use byte semantics.

Under byte semantics, when `use locale` is in effect, Perl uses the semantics associated with the

current locale. Absent a `use locale`, and absent a `use feature 'unicode_strings'` pragma, Perl currently uses US-ASCII (or Basic Latin in Unicode terminology) byte semantics, meaning that characters whose ordinal numbers are in the range 128 - 255 are undefined except for their ordinal numbers. This means that none have case (upper and lower), nor are any a member of character classes, like `[:alpha:]` or `\w`. (But all do belong to the `\w` class or the Perl regular expression extension `[:^alpha:]`.)

This behavior preserves compatibility with earlier versions of Perl, which allowed byte semantics in Perl operations only if none of the program's inputs were marked as being a source of Unicode character data. Such data may come from filehandles, from calls to external programs, from information provided by the system (such as `%ENV`), or from literals and constants in the source text.

The `bytes` pragma will always, regardless of platform, force byte semantics in a particular lexical scope. See *bytes*.

The `use feature 'unicode_strings'` pragma is intended to always, regardless of platform, force Unicode semantics in a particular lexical scope. In release 5.12, it is partially implemented, applying only to case changes. See *The "Unicode Bug"* below.

The `utf8` pragma is primarily a compatibility device that enables recognition of UTF-(8|EBCDIC) in literals encountered by the parser. Note that this pragma is only required while Perl defaults to byte semantics; when character semantics become the default, this pragma may become a no-op. See *utf8*.

Unless explicitly stated, Perl operators use character semantics for Unicode data and byte semantics for non-Unicode data. The decision to use character semantics is made transparently. If input data comes from a Unicode source--for example, if a character encoding layer is added to a filehandle or a literal Unicode string constant appears in a program--character semantics apply. Otherwise, byte semantics are in effect. The `bytes` pragma should be used to force byte semantics on Unicode data, and the `use feature 'unicode_strings'` pragma to force Unicode semantics on byte data (though in 5.12 it isn't fully implemented).

If strings operating under byte semantics and strings with Unicode character data are concatenated, the new string will have character semantics. This can cause surprises: See *BUGS*, below. You can choose to be warned when this happens. See *encoding::warnings*.

Under character semantics, many operations that formerly operated on bytes now operate on characters. A character in Perl is logically just a number ranging from 0 to 2^{31} or so. Larger characters may encode into longer sequences of bytes internally, but this internal detail is mostly hidden for Perl code. See *perluniintro* for more.

Effects of Character Semantics

Character semantics have the following effects:

- Strings--including hash keys--and regular expression patterns may contain characters that have an ordinal value larger than 255.

If you use a Unicode editor to edit your program, Unicode characters may occur directly within the literal strings in UTF-8 encoding, or UTF-16. (The former requires a BOM or `use utf8`, the latter requires a BOM.)

Unicode characters can also be added to a string by using the `\N{U+...}` notation. The Unicode code for the desired character, in hexadecimal, should be placed in the braces, after the `U`. For instance, a smiley face is `\N{U+263A}`.

Alternatively, you can use the `\x{...}` notation for characters 0x100 and above. For characters below 0x100 you may get byte semantics instead of character semantics; see *The "Unicode Bug"*. On EBCDIC machines there is the additional problem that the value for such characters gives the EBCDIC character rather than the Unicode one.

Additionally, if you

```
use charnames ':full';
```

you can use the `\N{...}` notation and put the official Unicode character name within the braces, such as `\N{WHITE SMILING FACE}`. See *charnames*.

- If an appropriate *encoding* is specified, identifiers within the Perl script may contain Unicode alphanumeric characters, including ideographs. Perl does not currently attempt to canonicalize variable names.
- Regular expressions match characters instead of bytes. `"."` matches a character instead of a byte.
- Character classes in regular expressions match characters instead of bytes and match against the character properties specified in the Unicode properties database. `\w` can be used to match a Japanese ideograph, for instance.
- Named Unicode properties, scripts, and block ranges may be used like character classes via the `\p{ }` "matches property" construct and the `\P{ }` negation, "doesn't match property". See *Unicode Character Properties* for more details.

You can define your own character properties and use them in the regular expression with the `\p{ }` or `\P{ }` construct. See *User-Defined Character Properties* for more details.

- The special pattern `\X` matches a logical character, an "extended grapheme cluster" in Standardese. In Unicode what appears to the user to be a single character, for example an accented `G`, may in fact be composed of a sequence of characters, in this case a `G` followed by an accent character. `\X` will match the entire sequence.
- The `tr///` operator translates characters instead of bytes. Note that the `tr///CU` functionality has been removed. For similar functionality see `pack('U0', ...)` and `pack('C0', ...)`.
- Case translation operators use the Unicode case translation tables when character input is provided. Note that `uc()`, or `\U` in interpolated strings, translates to uppercase, while `ucfirst`, or `\u` in interpolated strings, translates to titlecase in languages that make the distinction (which is equivalent to uppercase in languages without the distinction).
- Most operators that deal with positions or lengths in a string will automatically switch to using character positions, including `chop()`, `chomp()`, `substr()`, `pos()`, `index()`, `rindex()`, `sprintf()`, `write()`, and `length()`. An operator that specifically does not switch is `vec()`. Operators that really don't care include operators that treat strings as a bucket of bits such as `sort()`, and operators dealing with filenames.
- The `pack()/unpack()` letter `C` does *not* change, since it is often used for byte-oriented formats. Again, think `char` in the C language.
There is a new `U` specifier that converts between Unicode characters and code points. There is also a `w` specifier that is the equivalent of `chr/ord` and properly handles character values even if they are above 255.
- The `chr()` and `ord()` functions work on characters, similar to `pack("W")` and `unpack("W")`, *not* `pack("C")` and `unpack("C")`. `pack("C")` and `unpack("C")` are methods for emulating byte-oriented `chr()` and `ord()` on Unicode strings. While these methods reveal the internal encoding of Unicode strings, that is not something one normally needs to care about at all.
- The bit string operators, `&` `|` `^` `~`, can operate on character data. However, for backward compatibility, such as when using bit string operations when characters are all less than 256 in ordinal value, one should not use `~` (the bit complement) with characters of both values less than 256 and values greater than 256. Most importantly, DeMorgan's laws (`~($x|$y) eq ~$x&~$y` and `~($x&$y) eq ~$x|~$y`) will not hold. The reason for this mathematical *faux pas* is that the complement cannot return **both** the 8-bit (byte-wide) bit complement **and** the

full character-wide bit complement.

- You can define your own mappings to be used in `lc()`, `lcfirst()`, `uc()`, and `ucfirst()` (or their string-inlined versions). See *User-Defined Case Mappings* for more details.
- And finally, `scalar reverse()` reverses by character rather than by byte.

Unicode Character Properties

Most Unicode character properties are accessible by using regular expressions. They are used like character classes via the `\p{ }` "matches property" construct and the `\P{ }` negation, "doesn't match property".

For instance, `\p{Uppercase}` matches any character with the Unicode "Uppercase" property, while `\p{L}` matches any character with a `General_Category` of "L" (letter) property. Brackets are not required for single letter properties, so `\p{L}` is equivalent to `\pL`.

More formally, `\p{Uppercase}` matches any character whose Unicode Uppercase property value is True, and `\P{Uppercase}` matches any character whose Uppercase property value is False, and they could have been written as `\p{Uppercase=True}` and `\p{Uppercase=False}`, respectively

This formality is needed when properties are not binary, that is if they can take on more values than just True and False. For example, the `Bidi_Class` (see *Bidirectional Character Types* below), can take on a number of different values, such as Left, Right, Whitespace, and others. To match these, one needs to specify the property name (`Bidi_Class`), and the value being matched against (Left, Right, etc.). This is done, as in the examples above, by having the two components separated by an equal sign (or interchangeably, a colon), like `\p{Bidi_Class: Left}`.

All Unicode-defined character properties may be written in these compound forms of `\p{property=value}` or `\p{property:value}`, but Perl provides some additional properties that are written only in the single form, as well as single-form short-cuts for all binary properties and certain others described below, in which you may omit the property name and the equals or colon separator.

Most Unicode character properties have at least two synonyms (or aliases if you prefer), a short one that is easier to type, and a longer one which is more descriptive and hence it is easier to understand what it means. Thus the "L" and "Letter" above are equivalent and can be used interchangeably. Likewise, "Upper" is a synonym for "Uppercase", and we could have written `\p{Uppercase}` equivalently as `\p{Upper}`. Also, there are typically various synonyms for the values the property can be. For binary properties, "True" has 3 synonyms: "T", "Yes", and "Y"; and "False" has correspondingly "F", "No", and "N". But be careful. A short form of a value for one property may not mean the same thing as the same short form for another. Thus, for the `General_Category` property, "L" means "Letter", but for the `Bidi_Class` property, "L" means "Left". A complete list of properties and synonyms is in *perluniprops*.

Upper/lower case differences in the property names and values are irrelevant, thus `\p{Upper}` means the same thing as `\p{upper}` or even `\p{UPPeR}`. Similarly, you can add or subtract underscores anywhere in the middle of a word, so that these are also equivalent to `\p{U_p_p_e_r}`. And white space is irrelevant adjacent to non-word characters, such as the braces and the equals or colon separators so `\p{ Upper }` and `\p{ Upper_case : Y }` are equivalent to these as well. In fact, in most cases, white space and even hyphens can be added or deleted anywhere. So even `\p{ Up-per case = Yes }` is equivalent. All this is called "loose-matching" by Unicode. The few places where stricter matching is employed is in the middle of numbers, and the Perl extension properties that begin or end with an underscore. Stricter matching cares about white space (except adjacent to the non-word characters) and hyphens, and non-interior underscores.

You can also use negation in both `\p{ }` and `\P{ }` by introducing a caret (^) between the first brace and the property name: `\p{^Tamil}` is equal to `\P{Tamil}`.

General_Category

Every Unicode character is assigned a general category, which is the "most usual categorization of a character" (from <http://www.unicode.org/reports/tr44>).

The compound way of writing these is like `\p{General_Category=Number}` (short, `\p{gc:n}`). But Perl furnishes shortcuts in which everything up through the equal or colon separator is omitted. So you can instead just write `\pN`.

Here are the short and long forms of the General Category properties:

Short	Long
L	Letter
LC, L&	Cased_Letter (that is: <code>[\p{Ll}\p{Lu}\p{Lt}]</code>)
Lu	Uppercase_Letter
Ll	Lowercase_Letter
Lt	Titlecase_Letter
Lm	Modifier_Letter
Lo	Other_Letter
M	Mark
Mn	Nonspacing_Mark
Mc	Spacing_Mark
Me	Enclosing_Mark
N	Number
Nd	Decimal_Number (also Digit)
Nl	Letter_Number
No	Other_Number
P	Punctuation (also Punct)
Pc	Connector_Punctuation
Pd	Dash_Punctuation
Ps	Open_Punctuation
Pe	Close_Punctuation
Pi	Initial_Punctuation (may behave like Ps or Pe depending on usage)
Pf	Final_Punctuation (may behave like Ps or Pe depending on usage)
Po	Other_Punctuation
S	Symbol
Sm	Math_Symbol
Sc	Currency_Symbol
Sk	Modifier_Symbol
So	Other_Symbol
Z	Separator
Zs	Space_Separator
Zl	Line_Separator
Zp	Paragraph_Separator
C	Other
Cc	Control (also Cntrl)

Cf	Format
Cs	Surrogate (not usable)
Co	Private_Use
Cn	Unassigned

Single-letter properties match all characters in any of the two-letter sub-properties starting with the same letter. `LC` and `L&` are special cases, which are aliases for the set of `Ll`, `Lu`, and `Lt`.

Because Perl hides the need for the user to understand the internal representation of Unicode characters, there is no need to implement the somewhat messy concept of surrogates. `Cs` is therefore not supported.

Bidirectional Character Types

Because scripts differ in their directionality--Hebrew is written right to left, for example--Unicode supplies these properties in the `Bidi_Class` class:

Property	Meaning
L	Left-to-Right
LRE	Left-to-Right Embedding
LRO	Left-to-Right Override
R	Right-to-Left
AL	Arabic Letter
RLE	Right-to-Left Embedding
RLO	Right-to-Left Override
PDF	Pop Directional Format
EN	European Number
ES	European Separator
ET	European Terminator
AN	Arabic Number
CS	Common Separator
NSM	Non-Spacing Mark
BN	Boundary Neutral
B	Paragraph Separator
S	Segment Separator
WS	Whitespace
ON	Other Neutrals

This property is always written in the compound form. For example, `\p{Bidi_Class:R}` matches characters that are normally written right to left.

Scripts

The world's languages are written in a number of scripts. This sentence (unless you're reading it in translation) is written in Latin, while Russian is written in Cyrillic, and Greek is written in, well, Greek; Japanese mainly in Hiragana or Katakana. There are many more.

The Unicode Script property gives what script a given character is in, and can be matched with the compound form like `\p{Script=Hebrew}` (short: `\p{sc=hebr}`). Perl furnishes shortcuts for all script names. You can omit everything up through the equals (or colon), and simply write `\p{Latin}` or `\P{Cyrillic}`.

A complete list of scripts and their shortcuts is in *perluniprops*.

Use of "Is" Prefix

For backward compatibility (with Perl 5.6), all properties mentioned so far may have `Is` or `Is_` prepended to their name, so `\P{Is_Lu}`, for example, is equal to `\P{Lu}`, and

`\p{IsScript:Arabic}` is equal to `\p{Arabic}`.

Blocks

In addition to **scripts**, Unicode also defines **blocks** of characters. The difference between scripts and blocks is that the concept of scripts is closer to natural languages, while the concept of blocks is more of an artificial grouping based on groups of Unicode characters with consecutive ordinal values. For example, the "Basic Latin" block is all characters whose ordinals are between 0 and 127, inclusive, in other words, the ASCII characters. The "Latin" script contains some letters from this block as well as several more, like "Latin-1 Supplement", "Latin Extended-A", *etc.*, but it does not contain all the characters from those blocks. It does not, for example, contain digits, because digits are shared across many scripts. Digits and similar groups, like punctuation, are in the script called `Common`. There is also a script called `Inherited` for characters that modify other characters, and inherit the script value of the controlling character.

For more about scripts versus blocks, see UAX#24 "Unicode Script Property":
<http://www.unicode.org/reports/tr24>

The Script property is likely to be the one you want to use when processing natural language; the Block property may be useful in working with the nuts and bolts of Unicode.

Block names are matched in the compound form, like `\p{Block: Arrows}` or `\p{Blk=Hebrew}`. Unlike most other properties only a few block names have a Unicode-defined short name. But Perl does provide a (slight) shortcut: You can say, for example `\p{In_Arrows}` or `\p{In_Hebrew}`. For backwards compatibility, the `In` prefix may be omitted if there is no naming conflict with a script or any other property, and you can even use an `Is` prefix instead in those cases. But it is not a good idea to do this, for a couple reasons:

- 1 It is confusing. There are many naming conflicts, and you may forget some. For example, `\p{Hebrew}` means the *script* Hebrew, and NOT the *block* Hebrew. But would you remember that 6 months from now?
- 2 It is unstable. A new version of Unicode may pre-empt the current meaning by creating a property with the same name. There was a time in very early Unicode releases when `\p{Hebrew}` would have matched the *block* Hebrew; now it doesn't.

Some people just prefer to always use `\p{Block: foo}` and `\p{Script: bar}` instead of the shortcuts, for clarity, and because they can't remember the difference between 'In' and 'Is' anyway (or aren't confident that those who eventually will read their code will know).

A complete list of blocks and their shortcuts is in *perluniprops*.

Other Properties

There are many more properties than the very basic ones described here. A complete list is in *perluniprops*.

Unicode defines all its properties in the compound form, so all single-form properties are Perl extensions. A number of these are just synonyms for the Unicode ones, but some are genuine extensions, including a couple that are in the compound form. And quite a few of these are actually recommended by Unicode (in <http://www.unicode.org/reports/tr18>).

This section gives some details on all the extensions that aren't synonyms for compound-form Unicode properties (for those, you'll have to refer to the *Unicode Standard*).

`\p{All}`

This matches any of the 1_114_112 Unicode code points. It is a synonym for `\p{Any}`.

`\p{Alnum}`

This matches any `\p{Alphabetic}` or `\p{Decimal_Number}` character.

\p{Any}

This matches any of the 1_114_112 Unicode code points. It is a synonym for `\p{All}`.

\p{Assigned}

This matches any assigned code point; that is, any code point whose general category is not Unassigned (or equivalently, not Cn).

\p{Blank}

This is the same as `\h` and `\p{HorizSpace}`: A character that changes the spacing horizontally.

\p{Decomposition_Type: Non_Canonical} (Short: `\p{Dt=NonCanon}`)

Matches a character that has a non-canonical decomposition.

To understand the use of this rarely used property=value combination, it is necessary to know some basics about decomposition. Consider a character, say H. It could appear with various marks around it, such as an acute accent, or a circumflex, or various hooks, circles, arrows, *etc.*, above, below, to one side and/or the other, *etc.* There are many possibilities among the world's languages. The number of combinations is astronomical, and if there were a character for each combination, it would soon exhaust Unicode's more than a million possible characters. So Unicode took a different approach: there is a character for the base H, and a character for each of the possible marks, and they can be combined variously to get a final logical character. So a logical character--what appears to be a single character--can be a sequence of more than one individual characters. This is called an "extended grapheme cluster". (Perl furnishes the `\x` construct to match such sequences.)

But Unicode's intent is to unify the existing character set standards and practices, and a number of pre-existing standards have single characters that mean the same thing as some of these combinations. An example is ISO-8859-1, which has quite a few of these in the Latin-1 range, an example being "LATIN CAPITAL LETTER E WITH ACUTE". Because this character was in this pre-existing standard, Unicode added it to its repertoire. But this character is considered by Unicode to be equivalent to the sequence consisting of first the character "LATIN CAPITAL LETTER E", then the character "COMBINING ACUTE ACCENT".

"LATIN CAPITAL LETTER E WITH ACUTE" is called a "pre-composed" character, and the equivalence with the sequence is called canonical equivalence. All pre-composed characters are said to have a decomposition (into the equivalent sequence) and the decomposition type is also called canonical.

However, many more characters have a different type of decomposition, a "compatible" or "non-canonical" decomposition. The sequences that form these decompositions are not considered canonically equivalent to the pre-composed character. An example, again in the Latin-1 range, is the "SUPERSCRIPT ONE". It is kind of like a regular digit 1, but not exactly; its decomposition into the digit 1 is called a "compatible" decomposition, specifically a "super" decomposition. There are several such compatibility decompositions (see <http://www.unicode.org/reports/tr44>), including one called "compat" which means some miscellaneous type of decomposition that doesn't fit into the decomposition categories that Unicode has chosen.

Note that most Unicode characters don't have a decomposition, so their decomposition type is "None".

Perl has added the `Non_Canonical` type, for your convenience, to mean any of the compatibility decompositions.

\p{Graph}

Matches any character that is graphic. Theoretically, this means a character that on a printer would cause ink to be used.

\p{HorizSpace}

This is the same as `\h` and `\p{Blank}`: A character that changes the spacing horizontally.

`\p{In=*}`

This is a synonym for `\p{Present_In=*}`

`\p{PerlSpace}`

This is the same as `\s`, restricted to ASCII, namely `[\f\n\r\t]`.

Mnemonic: Perl's (original) space

`\p{PerlWord}`

This is the same as `\w`, restricted to ASCII, namely `[A-Za-z0-9_]`

Mnemonic: Perl's (original) word.

`\p{PosixAlnum}`

This matches any alphanumeric character in the ASCII range, namely `[A-Za-z0-9]`.

`\p{PosixAlpha}`

This matches any alphabetic character in the ASCII range, namely `[A-Za-z]`.

`\p{PosixBlank}`

This matches any blank character in the ASCII range, namely `[\t]`.

`\p{PosixCntrl}`

This matches any control character in the ASCII range, namely `[\x00-\x1F\x7F]`

`\p{PosixDigit}`

This matches any digit character in the ASCII range, namely `[0-9]`.

`\p{PosixGraph}`

This matches any graphical character in the ASCII range, namely `[\x21-\x7E]`.

`\p{PosixLower}`

This matches any lowercase character in the ASCII range, namely `[a-z]`.

`\p{PosixPrint}`

This matches any printable character in the ASCII range, namely `[\x20-\x7E]`. These are the graphical characters plus SPACE.

`\p{PosixPunct}`

This matches any punctuation character in the ASCII range, namely `[\x21-\x2F\x3A-\x40\x5B-\x60\x7B-\x7E]`. These are the graphical characters that aren't word characters. Note that the Posix standard includes in its definition of punctuation, those characters that Unicode calls "symbols."

`\p{PosixSpace}`

This matches any space character in the ASCII range, namely `[\f\n\r\t\x0B]` (the last being a vertical tab).

`\p{PosixUpper}`

This matches any uppercase character in the ASCII range, namely `[A-Z]`.

`\p{Present_In: *} (Short: \p{In=*)}`

This property is used when you need to know in what Unicode version(s) a character is.

The `"**"` above stands for some two digit Unicode version number, such as 1.1 or 4.0; or the `"**"` can also be `Unassigned`. This property will match the code points whose final disposition

has been settled as of the Unicode release given by the version number; `\p{Present_In: Unassigned}` will match those code points whose meaning has yet to be assigned.

For example, U+0041 "LATIN CAPITAL LETTER A" was present in the very first Unicode release available, which is 1.1, so this property is true for all valid "*" versions. On the other hand, U+1EFF was not assigned until version 5.1 when it became "LATIN SMALL LETTER Y WITH LOOP", so the only "*" that would match it are 5.1, 5.2, and later.

Unicode furnishes the `Age` property from which this is derived. The problem with `Age` is that a strict interpretation of it (which Perl takes) has it matching the precise release a code point's meaning is introduced in. Thus U+0041 would match only 1.1; and U+1EFF only 5.1. This is not usually what you want.

Some non-Perl implementations of the `Age` property may change its meaning to be the same as the Perl `Present_In` property; just be aware of that.

Another confusion with both these properties is that the definition is not that the code point has been assigned, but that the meaning of the code point has been determined. This is because 66 code points will always be unassigned, and, so the `Age` for them is the Unicode version the decision to make them so was made in. For example, U+FDD0 is to be permanently unassigned to a character, and the decision to do that was made in version 3.1, so `\p{Age=3.1}` matches this character and `\p{Present_In: 3.1}` and `up` matches as well.

`\p{Print}`

This matches any character that is graphical or blank, except controls.

`\p{SpacePerl}`

This is the same as `\s`, including beyond ASCII.

Mnemonic: Space, as modified by Perl. (It doesn't include the vertical tab which both the Posix standard and Unicode consider to be space.)

`\p{VertSpace}`

This is the same as `\v`: A character that changes the spacing vertically.

`\p{Word}`

This is the same as `\w`, including beyond ASCII.

User-Defined Character Properties

You can define your own binary character properties by defining subroutines whose names begin with "In" or "Is". The subroutines can be defined in any package. The user-defined properties can be used in the regular expression `\p` and `\P` constructs; if you are using a user-defined property from a package other than the one you are in, you must specify its package in the `\p` or `\P` construct.

```
# assuming property Is_Foreign defined in Lang::
package main; # property package name required
if ($txt =~ /\p{Lang::IsForeign}+/) { ... }

package Lang; # property package name not required
if ($txt =~ /\p{IsForeign}+/) { ... }
```

Note that the effect is compile-time and immutable once defined.

The subroutines must return a specially-formatted string, with one or more newline-separated lines. Each line must be one of the following:

- A single hexadecimal number denoting a Unicode code point to include.
- Two hexadecimal numbers separated by horizontal whitespace (space or tabular characters) denoting a range of Unicode code points to include.

- Something to include, prefixed by "+": a built-in character property (prefixed by "utf8::") or a user-defined character property, to represent all the characters in that property; two hexadecimal code points for a range; or a single hexadecimal code point.
- Something to exclude, prefixed by "-": an existing character property (prefixed by "utf8::") or a user-defined character property, to represent all the characters in that property; two hexadecimal code points for a range; or a single hexadecimal code point.
- Something to negate, prefixed by "!": an existing character property (prefixed by "utf8::") or a user-defined character property, to represent all the characters in that property; two hexadecimal code points for a range; or a single hexadecimal code point.
- Something to intersect with, prefixed by "&": an existing character property (prefixed by "utf8::") or a user-defined character property, for all the characters except the characters in the property; two hexadecimal code points for a range; or a single hexadecimal code point.

For example, to define a property that covers both the Japanese syllabaries (hiragana and katakana), you can define

```
sub InKana {
return <<END;
3040\t309F
30A0\t30FF
END
}
```

Imagine that the here-doc end marker is at the beginning of the line. Now you can use `\p{InKana}` and `\P{InKana}`.

You could also have used the existing block property names:

```
sub InKana {
return <<'END';
+utf8::InHiragana
+utf8::InKatakana
END
}
```

Suppose you wanted to match only the allocated characters, not the raw block ranges: in other words, you want to remove the non-characters:

```
sub InKana {
return <<'END';
+utf8::InHiragana
+utf8::InKatakana
-utf8::IsCn
END
}
```

The negation is useful for defining (surprise!) negated classes.

```
sub InNotKana {
return <<'END';
!utf8::InHiragana
-utf8::InKatakana
+utf8::IsCn
END
}
```

Intersection is useful for getting the common characters matched by two (or more) classes.

```
sub InFooAndBar {
    return <<'END';
+main::Foo
&main::Bar
END
}
```

It's important to remember not to use "&" for the first set; that would be intersecting with nothing (resulting in an empty set).

User-Defined Case Mappings

You can also define your own mappings to be used in the `lc()`, `lcfirst()`, `uc()`, and `ucfirst()` (or their string-inlined versions). The principle is similar to that of user-defined character properties: to define subroutines with names like `ToLower` (for `lc()` and `lcfirst()`), `ToTitle` (for the first character in `ucfirst()`), and `ToUpper` (for `uc()`, and the rest of the characters in `ucfirst()`).

The string returned by the subroutines needs to be two hexadecimal numbers separated by two tabulators: the two numbers being, respectively, the source code point and the destination code point. For example:

```
sub ToUpper {
return <<END;
0061\t\t0041
END
}
```

defines an `uc()` mapping that causes only the character "a" to be mapped to "A"; all other characters will remain unchanged.

(For serious hackers only) The above means you have to furnish a complete mapping; you can't just override a couple of characters and leave the rest unchanged. You can find all the mappings in the directory `$Config{privlib}/unicore/To/`. The mapping data is returned as the here-document, and the `utf8::ToSpecFoo` are special exception mappings derived from `<$Config{privlib}>/unicore/SpecialCasing.txt`. The "Digit" and "Fold" mappings that one can see in the directory are not directly user-accessible, one can use either the `Unicode::UCD` module, or just match case-insensitively (that's when the "Fold" mapping is used).

The mappings will only take effect on scalars that have been marked as having Unicode characters, for example by using `utf8::upgrade()`. Old byte-style strings are not affected.

The mappings are in effect for the package they are defined in.

Character Encodings for Input and Output

See *Encode*.

Unicode Regular Expression Support Level

The following list of Unicode support for regular expressions describes all the features currently supported. The references to "Level N" and the section numbers refer to the Unicode Technical Standard #18, "Unicode Regular Expressions", version 11, in May 2005.

- Level 1 - Basic Unicode Support

[1]	RL1.1	Hex Notation	- done
	RL1.2	Properties	- done
[2][3]			

- [4] RL1.2a Compatibility Properties - done
 - [5] RL1.3 Subtraction and Intersection - MISSING
 - [6] RL1.4 Simple Word Boundaries - done
 - [7] RL1.5 Simple Loose Matches - done
 - [8] RL1.6 Line Boundaries - MISSING
 - [9] RL1.7 Supplementary Code Points - done
- [1] `\x{...}`
- [2] `\p{...}` `\P{...}`
- [3] supports not only minimal list, but all Unicode character properties (see L</Unicode Character Properties>)
- [4] `\d \D \s \S \w \W \X [[:prop:]] [[:^prop:]]`
- [5] can use regular expression look-ahead [a] or user-defined character properties [b] to emulate set operations
- [6] `\b \B`
- [7] note that Perl does Full case-folding in matching (but with bugs),
not Simple: for example U+1F88 is equivalent to U+1F00 U+03B9,
not with 1F80. This difference matters mainly for certain Greek capital letters with certain modifiers: the Full case-folding decomposes the letter, while the Simple case-folding would map it to a single character.
- [8] should do `^` and `$` also on U+000B (`\v` in C), FF (`\f`), CR (`\r`), CRLF (`\r\n`), NEL (U+0085), LS (U+2028), and PS (U+2029); should also affect `<>`, `$.`, and script line numbers; should not split lines within CRLF [c] (i.e. there is no empty line between `\r` and `\n`)
- [9] UTF-8/UTF-EBDDIC used in perl allows not only U+10000 to U+10FFFF but also beyond U+10FFFF [d]

[a] You can mimic class subtraction using lookahead. For example, what UTS#18 might write as

```
[ {Greek} - [ {UNASSIGNED} ] ]
```

in Perl can be written as:

```
(?!\p{Unassigned})\p{InGreekAndCoptic}
(?:\p{Assigned})\p{InGreekAndCoptic}
```

But in this particular example, you probably really want

```
\p{GreekAndCoptic}
```

which will match assigned characters known to be part of the Greek script.

Also see the `Unicode::Regex::Set` module, it does implement the full UTS#18 grouping, intersection, union, and removal (subtraction) syntax.

[b] '+' for union, '-' for removal (set-difference), '&' for intersection (see *User-Defined Character Properties*)

[c] Try the `:crlf` layer (see *PerlIO*).

[d] U+FFFF will currently generate a warning message if 'utf8' warnings are enabled

- Level 2 - Extended Unicode Support

[10][11]	RL2.1	Canonical Equivalents	- MISSING	
	RL2.2	Default Grapheme Clusters	- MISSING	[12]
	RL2.3	Default Word Boundaries	- MISSING	[14]
	RL2.4	Default Loose Matches	- MISSING	[15]
	RL2.5	Name Properties	- MISSING	[16]
	RL2.6	Wildcard Properties	- MISSING	
[10] see UAX#15 "Unicode Normalization Forms"				
[11] have <code>Unicode::Normalize</code> but not integrated to regexes				
[12] have <code>\X</code> but we don't have a "Grapheme Cluster Mode"				
[14] see UAX#29, Word Boundaries				
[15] see UAX#21 "Case Mappings"				
[16] have <code>\N{...}</code> but neither compute names of CJK Ideographs and Hangul Syllables nor use a loose match [e]				

[e] `\N{...}` allows namespaces (see *charnings*).

- Level 3 - Tailored Support

[17][18]	RL3.1	Tailored Punctuation	- MISSING	
	RL3.2	Tailored Grapheme Clusters	- MISSING	
	RL3.3	Tailored Word Boundaries	- MISSING	
	RL3.4	Tailored Loose Matches	- MISSING	
	RL3.5	Tailored Ranges	- MISSING	
	RL3.6	Context Matching	- MISSING	[19]
(RL3.7	Incremental Matches	- MISSING	
	RL3.8	Unicode Set Sharing)		
	RL3.9	Possible Match Sets	- MISSING	
	RL3.10	Folded Matching	- MISSING	[20]
	RL3.11	Submatchers	- MISSING	
[17] see UAX#10 "Unicode Collation Algorithms"				
[18] have <code>Unicode::Collate</code> but not integrated to regexes				
[19] have <code>(?<=x)</code> and <code>(?=x)</code> , but look-aheads or look-behinds should see				
outside of the target substring				
[20] need insensitive matching for linguistic features other than case;				
for example, hiragana to katakana, wide and narrow, simplified Han				
to traditional Han (see UTR#30 "Character Foldings")				

Unicode Encodings

Unicode characters are assigned to *code points*, which are abstract numbers. To use these numbers, various encodings are needed.

- UTF-8

UTF-8 is a variable-length (1 to 6 bytes, current character allocations require 4 bytes), byte-order independent encoding. For ASCII (and we really do mean 7-bit ASCII, not another 8-bit encoding), UTF-8 is transparent.

The following table is from Unicode 3.2.

Code Points	1st Byte	2nd Byte	3rd Byte	4th Byte
U+0000..U+007F	00..7F			
U+0080..U+07FF		* C2..DF	80..BF	
U+0800..U+0FFF	E0	* A0..BF	80..BF	
U+1000..U+CFFF		E1..EC	80..BF	80..BF
U+D000..U+D7FF		ED	80..9F	80..BF
U+D800..U+DFFF		++++++	utf16 surrogates, not legal utf8	
++++++				
U+E000..U+FFFF		EE..EF	80..BF	80..BF
U+10000..U+3FFFF	F0	* 90..BF	80..BF	80..BF
U+40000..U+FFFFF	F1..F3	80..BF	80..BF	80..BF
U+100000..U+10FFFF	F4	80..8F	80..BF	80..BF

Note the gaps before several of the byte entries above marked by '*'. These are caused by legal UTF-8 avoiding non-shortest encodings: it is technically possible to UTF-8-encode a single code point in different ways, but that is explicitly forbidden, and the shortest possible encoding should always be used (and that is what Perl does).

Another way to look at it is via bits:

Code Points	1st Byte	2nd Byte	3rd Byte	4th Byte
0aaaaaaa	0aaaaaaa			
00000bbbbbaaaaaa	110bbbb	10aaaaaa		
ccccbbbbbbaaaaaa	1110cccc	10bbbbbb	10aaaaaa	
00000ddccccccbbbbbbaaaaaa	11110ddd	10cccccc	10bbbbbb	
10aaaaaa				

As you can see, the continuation bytes all begin with "10", and the leading bits of the start byte tell how many bytes there are in the encoded character.

- UTF-EBCDIC

Like UTF-8 but EBCDIC-safe, in the way that UTF-8 is ASCII-safe.

- UTF-16, UTF-16BE, UTF-16LE, Surrogates, and BOMs (Byte Order Marks)

The followings items are mostly for reference and general Unicode knowledge, Perl doesn't use these constructs internally.

UTF-16 is a 2 or 4 byte encoding. The Unicode code points U+0000..U+FFFF are stored in a single 16-bit unit, and the code points U+10000..U+10FFFF in two 16-bit units. The latter case is using *surrogates*, the first 16-bit unit being the *high surrogate*, and the second being the *low surrogate*.

Surrogates are code points set aside to encode the U+10000..U+10FFFF range of Unicode code points in pairs of 16-bit units. The *high surrogates* are the range U+D800..U+DBFF and the *low surrogates* are the range U+DC00..U+DFFF. The surrogate encoding is

```
$hi = ($uni - 0x10000) / 0x400 + 0xD800;
$lo = ($uni - 0x10000) % 0x400 + 0xDC00;
```

and the decoding is

```
$uni = 0x10000 + ($hi - 0xD800) * 0x400 + ($lo - 0xDC00);
```

If you try to generate surrogates (for example by using `chr()`), you will get a warning, if warnings are turned on, because those code points are not valid for a Unicode character.

Because of the 16-bitness, UTF-16 is byte-order dependent. UTF-16 itself can be used for in-memory computations, but if storage or transfer is required either UTF-16BE (big-endian) or UTF-16LE (little-endian) encodings must be chosen.

This introduces another problem: what if you just know that your data is UTF-16, but you don't know which endianness? Byte Order Marks, or BOMs, are a solution to this. A special character has been reserved in Unicode to function as a byte order marker: the character with the code point `U+FEFF` is the BOM.

The trick is that if you read a BOM, you will know the byte order, since if it was written on a big-endian platform, you will read the bytes `0xFE 0xFF`, but if it was written on a little-endian platform, you will read the bytes `0xFF 0xFE`. (And if the originating platform was writing in UTF-8, you will read the bytes `0xEF 0xBB 0xBF`.)

The way this trick works is that the character with the code point `U+FFFE` is guaranteed not to be a valid Unicode character, so the sequence of bytes `0xFF 0xFE` is unambiguously "BOM, represented in little-endian format" and cannot be `U+FFFE`, represented in big-endian format". (Actually, `U+FFFE` is legal for use by your program, even for input/output, but better not use it if you need a BOM. But it is "illegal for interchange", so that an unsuspecting program won't get confused.)

- UTF-32, UTF-32BE, UTF-32LE

The UTF-32 family is pretty much like the UTF-16 family, expect that the units are 32-bit, and therefore the surrogate scheme is not needed. The BOM signatures will be `0x00 0x00 0xFF 0xFF` for BE and `0xFF 0xFE 0x00 0x00` for LE.

- UCS-2, UCS-4

Encodings defined by the ISO 10646 standard. UCS-2 is a 16-bit encoding. Unlike UTF-16, UCS-2 is not extensible beyond `U+FFFF`, because it does not use surrogates. UCS-4 is a 32-bit encoding, functionally identical to UTF-32.

- UTF-7

A seven-bit safe (non-eight-bit) encoding, which is useful if the transport or storage is not eight-bit safe. Defined by RFC 2152.

Security Implications of Unicode

Read *Unicode Security Considerations*. Also, note the following:

- Malformed UTF-8

Unfortunately, the specification of UTF-8 leaves some room for interpretation of how many bytes of encoded output one should generate from one input Unicode character. Strictly speaking, the shortest possible sequence of UTF-8 bytes should be generated, because otherwise there is potential for an input buffer overflow at the receiving end of a UTF-8 connection. Perl always generates the shortest length UTF-8, and with warnings on, Perl will warn about non-shortest length UTF-8 along with other malformations, such as the surrogates, which are not real Unicode code points.

- Regular expressions behave slightly differently between byte data and character (Unicode) data. For example, the "word character" character class `\w` will work differently depending on if data is eight-bit bytes or Unicode.

In the first case, the set of `\w` characters is either small--the default set of alphabetic characters, digits, and the `"_"`--or, if you are using a locale (see *perllocale*), the `\w` might contain a few more letters according to your language and country.

In the second case, the `\w` set of characters is much, much larger. Most importantly, even in the set of the first 256 characters, it will probably match different characters: unlike most locales, which are specific to a language and country pair, Unicode classifies all the characters that are letters *somewhere* as `\w`. For example, your locale might not think that LATIN SMALL LETTER ETH is a letter (unless you happen to speak Icelandic), but Unicode does.

As discussed elsewhere, Perl has one foot (two hooves?) planted in each of two worlds: the old world of bytes and the new world of characters, upgrading from bytes to characters when necessary. If your legacy code does not explicitly use Unicode, no automatic switch-over to characters should happen. Characters shouldn't get downgraded to bytes, either. It is possible to accidentally mix bytes and characters, however (see *perluniintro*), in which case `\w` in regular expressions might start behaving differently. Review your code. Use warnings and the `strict` pragma.

Unicode in Perl on EBCDIC

The way Unicode is handled on EBCDIC platforms is still experimental. On such platforms, references to UTF-8 encoding in this document and elsewhere should be read as meaning the UTF-EBCDIC specified in Unicode Technical Report 16, unless ASCII vs. EBCDIC issues are specifically discussed. There is no `utfebcdic` pragma or `":utfebcdic"` layer; rather, `"utf8"` and `":utf8"` are reused to mean the platform's "natural" 8-bit encoding of Unicode. See *perlebcdic* for more discussion of the issues.

Locales

Usually locale settings and Unicode do not affect each other, but there are a couple of exceptions:

- You can enable automatic UTF-8-ification of your standard file handles, default `open()` layer, and `@ARGV` by using either the `-C` command line switch or the `PERL_UNICODE` environment variable, see *perlrun* for the documentation of the `-C` switch.
- Perl tries really hard to work both with Unicode and the old byte-oriented world. Most often this is nice, but sometimes Perl's straddling of the proverbial fence causes problems.

When Unicode Does Not Happen

While Perl does have extensive ways to input and output in Unicode, and few other 'entry points' like the `@ARGV` which can be interpreted as Unicode (UTF-8), there still are many places where Unicode (in some encoding or another) could be given as arguments or received as results, or both, but it is not.

The following are such interfaces. Also, see *The "Unicode Bug"*. For all of these interfaces Perl currently (as of 5.8.3) simply assumes byte strings both as arguments and results, or UTF-8 strings if the `encoding` pragma has been used.

One reason why Perl does not attempt to resolve the role of Unicode in these cases is that the answers are highly dependent on the operating system and the file system(s). For example, whether filenames can be in Unicode, and in exactly what kind of encoding, is not exactly a portable concept. Similarly for the `qx` and `system`: how well will the 'command line interface' (and which of them?) handle Unicode?

- `chdir`, `chmod`, `chown`, `chroot`, `exec`, `link`, `lstat`, `mkdir`, `rename`, `rmdir`, `stat`, `symlink`, `truncate`, `unlink`, `utime`, `-X`
- `%ENV`
- `glob` (aka the `<*>`)
- `open`, `opendir`, `sysopen`
- `qx` (aka the backtick operator), `system`
- `readdir`, `readlink`

The "Unicode Bug"

The term, the "Unicode bug" has been applied to an inconsistency with the Unicode characters whose ordinals are in the Latin-1 Supplement block, that is, between 128 and 255. Without a locale specified, unlike all other characters or code points, these characters have very different semantics in byte semantics versus character semantics.

In character semantics they are interpreted as Unicode code points, which means they have the same semantics as Latin-1 (ISO-8859-1).

In byte semantics, they are considered to be unassigned characters, meaning that the only semantics they have is their ordinal numbers, and that they are not members of various character classes. None are considered to match `\w` for example, but all match `\w`. (On EBCDIC platforms, the behavior may be different from this, depending on the underlying C language library functions.)

The behavior is known to have effects on these areas:

- Changing the case of a scalar, that is, using `uc()`, `ucfirst()`, `lc()`, and `lcfirst()`, or `\L`, `\U`, `\u` and `\l` in regular expression substitutions.
- Using caseless (`/i`) regular expression matching
- Matching a number of properties in regular expressions, such as `\w`
- User-defined case change mappings. You can create a `ToUpper()` function, for example, which overrides Perl's built-in case mappings. The scalar must be encoded in utf8 for your function to actually be invoked.

This behavior can lead to unexpected results in which a string's semantics suddenly change if a code point above 255 is appended to or removed from it, which changes the string's semantics from byte to character or vice versa. As an example, consider the following program and its output:

```
$ perl -le'  
    $s1 = "\xC2";  
    $s2 = "\x{2660}";  
    for ($s1, $s2, $s1.$s2) {  
        print /\w/ || 0;  
    }  
,  
0  
0  
1
```

If there's no `\w` in `s1` or in `s2`, why does their concatenation have one?

This anomaly stems from Perl's attempt to not disturb older programs that didn't use Unicode, and hence had no semantics for characters outside of the ASCII range (except in a locale), along with Perl's desire to add Unicode support seamlessly. The result wasn't seamless: these characters were orphaned.

Work is being done to correct this, but only some of it was complete in time for the 5.12 release. What has been finished is the important part of the case changing component. Due to concerns, and some evidence, that older code might have come to rely on the existing behavior, the new behavior must be explicitly enabled by the feature `unicode_strings` in the *feature* pragma, even though no new syntax is involved.

See *"lc" in perlfunc* for details on how this pragma works in combination with various others for casing. Even though the pragma only affects casing operations in the 5.12 release, it is planned to have it affect all the problematic behaviors in later releases: you can't have one without them all.

In the meantime, a workaround is to always call `utf8::upgrade($string)`, or to use the standard module

Encode. Also, a scalar that has any characters whose ordinal is above 0x100, or which were specified using either of the `\N{ . . . }` notations will automatically have character semantics.

Forcing Unicode in Perl (Or Unforcing Unicode in Perl)

Sometimes (see *When Unicode Does Not Happen* or *The "Unicode Bug"*) there are situations where you simply need to force a byte string into UTF-8, or vice versa. The low-level calls `utf8::upgrade($bytestring)` and `utf8::downgrade($utf8string[, FAIL_OK])` are the answers.

Note that `utf8::downgrade()` can fail if the string contains characters that don't fit into a byte.

Calling either function on a string that already is in the desired state is a no-op.

Using Unicode in XS

If you want to handle Perl Unicode in XS extensions, you may find the following C APIs useful. See also *"Unicode Support" in perlguides* for an explanation about Unicode at the XS level, and *perlapi* for the API details.

- `DO_UTF8(sv)` returns true if the UTF8 flag is on and the bytes pragma is not in effect. `SvUTF8(sv)` returns true if the UTF8 flag is on; the bytes pragma is ignored. The UTF8 flag being on does **not** mean that there are any characters of code points greater than 255 (or 127) in the scalar or that there are even any characters in the scalar. What the UTF8 flag means is that the sequence of octets in the representation of the scalar is the sequence of UTF-8 encoded code points of the characters of a string. The UTF8 flag being off means that each octet in this representation encodes a single character with code point 0..255 within the string. Perl's Unicode model is not to use UTF-8 until it is absolutely necessary.
- `uvchr_to_utf8(buf, chr)` writes a Unicode character code point into a buffer encoding the code point as UTF-8, and returns a pointer pointing after the UTF-8 bytes. It works appropriately on EBCDIC machines.
- `utf8_to_uvchr(buf, lenp)` reads UTF-8 encoded bytes from a buffer and returns the Unicode character code point and, optionally, the length of the UTF-8 byte sequence. It works appropriately on EBCDIC machines.
- `utf8_length(start, end)` returns the length of the UTF-8 encoded buffer in characters. `sv_len_utf8(sv)` returns the length of the UTF-8 encoded scalar.
- `sv_utf8_upgrade(sv)` converts the string of the scalar to its UTF-8 encoded form. `sv_utf8_downgrade(sv)` does the opposite, if possible. `sv_utf8_encode(sv)` is like `sv_utf8_upgrade` except that it does not set the UTF8 flag. `sv_utf8_decode()` does the opposite of `sv_utf8_encode()`. Note that none of these are to be used as general-purpose encoding or decoding interfaces: use *Encode* for that. `sv_utf8_upgrade()` is affected by the encoding pragma but `sv_utf8_downgrade()` is not (since the encoding pragma is designed to be a one-way street).
- `is_utf8_char(s)` returns true if the pointer points to a valid UTF-8 character.
- `is_utf8_string(buf, len)` returns true if `len` bytes of the buffer are valid UTF-8.
- `UTF8SKIP(buf)` will return the number of bytes in the UTF-8 encoded character in the buffer. `UNISKIP(chr)` will return the number of bytes required to UTF-8-encode the Unicode character code point. `UTF8SKIP()` is useful for example for iterating over the characters of a UTF-8 encoded buffer; `UNISKIP()` is useful, for example, in computing the size required for a UTF-8 encoded buffer.
- `utf8_distance(a, b)` will tell the distance in characters between the two pointers pointing to the same UTF-8 encoded buffer.
- `utf8_hop(s, off)` will return a pointer to a UTF-8 encoded buffer that is `off` (positive or negative) Unicode characters displaced from the UTF-8 buffer `s`. Be careful not to overstep

the buffer: `utf8_hop()` will merrily run off the end or the beginning of the buffer if told to do so.

- `pv_uni_display(dsv, spv, len, pvlm, flags)` and `sv_uni_display(dsv, ssv, pvlm, flags)` are useful for debugging the output of Unicode strings and scalars. By default they are useful only for debugging--they display **all** characters as hexadecimal code points--but with the flags `UNI_DISPLAY_ISPRINT`, `UNI_DISPLAY_BACKSLASH`, and `UNI_DISPLAY_QQ` you can make the output more readable.
- `ibcmp_utf8(s1, pe1, l1, u1, s2, pe2, l2, u2)` can be used to compare two strings case-insensitively in Unicode. For case-sensitive comparisons you can just use `memcmp()` and `memcmpNE()` as usual.

For more information, see *perlapi*, and *utf8.c* and *utf8.h* in the Perl source code distribution.

Hacking Perl to work on earlier Unicode versions (for very serious hackers only)

Perl by default comes with the latest supported Unicode version built in, but you can change to use any earlier one.

Download the files in the version of Unicode that you want from the Unicode web site <http://www.unicode.org>). These should replace the existing files in `\$Config{privlib}/unicore.` (`\$Config` is available from the `Config` module.) Follow the instructions in *README.perl* in that directory to change some of their names, and then run *make*.

It is even possible to download them to a different directory, and then change *utf8_heavy.pl* in the directory `\$Config{privlib}` to point to the new directory, or maybe make a copy of that directory before making the change, and using `@INC` or the `-I` run-time flag to switch between versions at will (but because of caching, not in the middle of a process), but all this is beyond the scope of these instructions.

BUGS

Interaction with Locales

Use of locales with Unicode data may lead to odd results. Currently, Perl attempts to attach 8-bit locale info to characters in the range 0..255, but this technique is demonstrably incorrect for locales that use characters above that range when mapped into Unicode. Perl's Unicode support will also tend to run slower. Use of locales with Unicode is discouraged.

Problems with characters in the Latin-1 Supplement range

See *The "Unicode Bug"*

Problems with case-insensitive regular expression matching

There are problems with case-insensitive matches, including those involving character classes (enclosed in [square brackets]), characters whose fold is to multiple characters (such as the single character LATIN SMALL LIGATURE FFL matches case-insensitively with the 3-character string `ffl`), and characters in the Latin-1 Supplement.

Interaction with Extensions

When Perl exchanges data with an extension, the extension should be able to understand the UTF8 flag and act accordingly. If the extension doesn't know about the flag, it's likely that the extension will return incorrectly-flagged data.

So if you're working with Unicode data, consult the documentation of every module you're using if there are any issues with Unicode data exchange. If the documentation does not talk about Unicode at all, suspect the worst and probably look at the source to learn how the module is implemented. Modules written completely in Perl shouldn't cause problems. Modules that directly or indirectly access code written in other programming languages are at risk.

For affected functions, the simple strategy to avoid data corruption is to always make the encoding of

the exchanged data explicit. Choose an encoding that you know the extension can handle. Convert arguments passed to the extensions to that encoding and convert results back from that encoding. Write wrapper functions that do the conversions for you, so you can later change the functions when the extension catches up.

To provide an example, let's say the popular `Foo::Bar::escape_html` function doesn't deal with Unicode data yet. The wrapper function would convert the argument to raw UTF-8 and convert the result back to Perl's internal representation like so:

```
sub my_escape_html ($) {
    my($what) = shift;
    return unless defined $what;

    Encode::decode_utf8(Foo::Bar::escape_html(Encode::encode_utf8($what)));
}
```

Sometimes, when the extension does not convert data but just stores and retrieves them, you will be in a position to use the otherwise dangerous `Encode::_utf8_on()` function. Let's say the popular `Foo::Bar` extension, written in C, provides a `param` method that lets you store and retrieve data according to these prototypes:

```
$self->param($name, $value);           # set a scalar
$value = $self->param($name);          # retrieve a scalar
```

If it does not yet provide support for any encoding, one could write a derived class with such a `param` method:

```
sub param {
    my($self,$name,$value) = @_;
    utf8::upgrade($name);           # make sure it is UTF-8 encoded
    if (defined $value) {
        utf8::upgrade($value);      # make sure it is UTF-8 encoded
        return $self->SUPER::param($name,$value);
    } else {
        my $ret = $self->SUPER::param($name);
        Encode::_utf8_on($ret);      # we know, it is UTF-8 encoded
        return $ret;
    }
}
```

Some extensions provide filters on data entry/exit points, such as `DB_File::filter_store_key` and family. Look out for such filters in the documentation of your extensions, they can make the transition to Unicode data much easier.

Speed

Some functions are slower when working on UTF-8 encoded strings than on byte encoded strings. All functions that need to hop over characters such as `length()`, `substr()` or `index()`, or matching regular expressions can work **much** faster when the underlying data are byte-encoded.

In Perl 5.8.0 the slowness was often quite spectacular; in Perl 5.8.1 a caching scheme was introduced which will hopefully make the slowness somewhat less spectacular, at least for some operations. In general, operations with UTF-8 encoded strings are still slower. As an example, the Unicode properties (character classes) like `\p{Nd}` are known to be quite a bit slower (5-20 times) than their simpler counterparts like `\d` (then again, there 268 Unicode characters matching `Nd` compared with the 10 ASCII characters matching `d`).

Problems on EBCDIC platforms

There are a number of known problems with Perl on EBCDIC platforms. If you want to use Perl there, send email to perlbug@perl.org.

In earlier versions, when byte and character data were concatenated, the new string was sometimes created by decoding the byte strings as *ISO 8859-1 (Latin-1)*, even if the old Unicode string used EBCDIC.

If you find any of these, please report them as bugs.

Porting code from perl-5.6.X

Perl 5.8 has a different Unicode model from 5.6. In 5.6 the programmer was required to use the `utf8` pragma to declare that a given scope expected to deal with Unicode data and had to make sure that only Unicode data were reaching that scope. If you have code that is working with 5.6, you will need some of the following adjustments to your code. The examples are written such that the code will continue to work under 5.6, so you should be safe to try them out.

- A filehandle that should read or write UTF-8

```
if ($] > 5.007) {  
    binmode $fh, ":encoding(utf8)";  
}
```

- A scalar that is going to be passed to some extension

Be it `Compress::Zlib`, `Apache::Request` or any extension that has no mention of Unicode in the manpage, you need to make sure that the UTF8 flag is stripped off. Note that at the time of this writing (October 2002) the mentioned modules are not UTF-8-aware. Please check the documentation to verify if this is still true.

```
if ($] > 5.007) {  
    require Encode;  
    $val = Encode::encode_utf8($val); # make octets  
}
```

- A scalar we got back from an extension

If you believe the scalar comes back as UTF-8, you will most likely want the UTF8 flag restored:

```
if ($] > 5.007) {  
    require Encode;  
    $val = Encode::decode_utf8($val);  
}
```

- Same thing, if you are really sure it is UTF-8

```
if ($] > 5.007) {  
    require Encode;  
    Encode::_utf8_on($val);  
}
```

- A wrapper for `fetchrow_array` and `fetchrow_hashref`

When the database contains only UTF-8, a wrapper function or method is a convenient way to replace all your `fetchrow_array` and `fetchrow_hashref` calls. A wrapper function will also make it easier to adapt to future enhancements in your database driver. Note that at the time of this writing (October 2002), the DBI has no standardized way to deal with UTF-8 data. Please check the documentation to verify if that is still true.

```
sub fetchrow {
```

```
my($self, $sth, $what) = @_; # $what is one of
fetchrow_{array,hashref}
if ($] < 5.007) {
    return $sth->$what;
} else {
    require Encode;
    if (wantarray) {
        my @arr = $sth->$what;
        for (@arr) {
            defined && /[^\000-\177]/ && Encode::_utf8_on($_);
        }
        return @arr;
    } else {
        my $ret = $sth->$what;
        if (ref $ret) {
            for my $k (keys %$ret) {
                defined && /[^\000-\177]/ && Encode::_utf8_on($_) for
$ret->{$k};
            }
            return $ret;
        } else {
            defined && /[^\000-\177]/ && Encode::_utf8_on($_) for $ret;
            return $ret;
        }
    }
}
```

- A large scalar that you know can only contain ASCII

Scalars that contain only ASCII and are marked as UTF-8 are sometimes a drag to your program. If you recognize such a situation, just remove the UTF8 flag:

```
utf8::downgrade($val) if $] > 5.007;
```

SEE ALSO

perlunitut, *perluniintro*, *perluniprops*, *Encode*, *open*, *utf8*, *bytes*, *perlretut*, "*UTF-8*" in *perlvar* <http://www.unicode.org/reports/tr44>).