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.

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 using a particular encoding, see *encoding*.

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 Unicode data--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*.

You can also use the `encoding` pragma to change the default encoding of the data in your script; see *encoding*.

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.

If you wish to interpret byte strings as UTF-8 instead, use the `encoding` pragma:

```
    use encoding 'utf8';
```

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.

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 as 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 `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.

If strings operating under byte semantics and strings with Unicode character data are concatenated, the new string will be created by decoding the byte strings as *ISO 8859-1 (Latin-1)*, even if the old Unicode string used EBCDIC. This translation is done without regard to the system's native 8-bit encoding. To change this for systems with non-Latin-1 and non-EBCDIC native encodings, use the `encoding` pragma. See *encoding*.

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 one of the various Unicode encodings (UTF-8, UTF-EBCDIC, UCS-2, etc.), but will be recognized as such and converted to Perl's internal representation only if the appropriate *encoding* is specified.

  Unicode characters can also be added to a string by using the `\x{...}` notation. The Unicode code for the desired character, in hexadecimal, should be placed in the braces. For instance, a smiley face is `\x{263A}`. This encoding scheme only works for characters with a code of 0x100 or above.

  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}`.

- 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. The `\C` pattern is provided to force a match a single byte--a `char` in C, hence `\C`.

- 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.

  (However, and as a limitation of the current implementation, using `\w` or `\W` *inside* a `[...]` character class will still match with byte semantics.)

- 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".

  For instance, `\p{Lu}` matches any character with the Unicode "Lu" (Letter, uppercase) property, while `\p{M}` matches any character with an "M" (mark--accents and such) property. Brackets are not required for single letter properties, so `\p{M}` is equivalent to `\pM`. Many predefined properties are available, such as `\p{Mirrored}` and `\p{Tibetan}`.

  The official Unicode script and block names have spaces and dashes as separators, but for convenience you can use dashes, spaces, or underbars, and case is unimportant. It is recommended, however, that for consistency you use the following naming: the official Unicode script, property, or block name (see below for the additional rules that apply to block names) with whitespace and dashes removed, and the words "uppercase-first-lowercase-rest". `Latin-1 Supplement` thus becomes `Latin1Supplement`.

  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}`.

  NOTE: the properties, scripts, and blocks listed here are as of Unicode 3.2.0, March 2002, or Perl 5.8.0, July 2002. Unicode 4.0.0 came out in April 2003, and Perl 5.8.1 in September 2003.

  Here are the basic Unicode General Category properties, followed by their long form. You can use either; `\p{Lu}` and `\p{UppercaseLetter}`, for instance, are identical.

```
    Short       Long

    L           Letter
    LC          CasedLetter
    Lu          UppercaseLetter
    Ll          LowercaseLetter
    Lt          TitlecaseLetter
    Lm          ModifierLetter
    Lo          OtherLetter

    M           Mark
    Mn          NonspacingMark
    Mc          SpacingMark
    Me          EnclosingMark

    N           Number
    Nd          DecimalNumber
    Nl          LetterNumber
    No          OtherNumber

    P           Punctuation
    Pc          ConnectorPunctuation
    Pd          DashPunctuation
    Ps          OpenPunctuation
    Pe          ClosePunctuation
    Pi          InitialPunctuation
                (may behave like Ps or Pe depending on usage)
    Pf          FinalPunctuation
```

	(may behave like Ps or Pe depending on usage)
Po	OtherPunctuation
S	Symbol
Sm	MathSymbol
Sc	CurrencySymbol
Sk	ModifierSymbol
So	OtherSymbol
Z	Separator
Zs	SpaceSeparator
Zl	LineSeparator
Zp	ParagraphSeparator
C	Other
Cc	Control
Cf	Format
Cs	Surrogate (not usable)
Co	PrivateUse
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.

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

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

For example, `\p{BidiClass:R}` matches characters that are normally written right to left.

Scripts

The script names which can be used by `\p{...}` and `\P{...}`, such as in `\p{Latin}` or `\p{Cyrillic}`, are as follows:

```
    Arabic
    Armenian
    Bengali
    Bopomofo
    Buhid
    CanadianAboriginal
    Cherokee
    Cyrillic
    Deseret
    Devanagari
    Ethiopic
    Georgian
    Gothic
    Greek
    Gujarati
    Gurmukhi
    Han
    Hangul
    Hanunoo
    Hebrew
    Hiragana
    Inherited
    Kannada
    Katakana
    Khmer
    Lao
    Latin
    Malayalam
    Mongolian
    Myanmar
    Ogham
    OldItalic
    Oriya
    Runic
    Sinhala
    Syriac
    Tagalog
    Tagbanwa
    Tamil
    Telugu
    Thaana
    Thai
    Tibetan
    Yi
```

Extended property classes can supplement the basic properties, defined by the *PropList* Unicode database:

```
    ASCIIHexDigit
    BidiControl
    Dash
    Deprecated
    Diacritic
    Extender
    GraphemeLink
    HexDigit
    Hyphen
    Ideographic
    IDSBinaryOperator
    IDSTrinaryOperator
    JoinControl
    LogicalOrderException
    NoncharacterCodePoint
    OtherAlphabetic
    OtherDefaultIgnorableCodePoint
    OtherGraphemeExtend
    OtherLowercase
    OtherMath
    OtherUppercase
    QuotationMark
    Radical
    SoftDotted
    TerminalPunctuation
    UnifiedIdeograph
    WhiteSpace
```

and there are further derived properties:

```
    Alphabetic      Lu + Ll + Lt + Lm + Lo + OtherAlphabetic
    Lowercase       Ll + OtherLowercase
    Uppercase       Lu + OtherUppercase
    Math            Sm + OtherMath

    ID_Start        Lu + Ll + Lt + Lm + Lo + Nl
    ID_Continue     ID_Start + Mn + Mc + Nd + Pc

    Any             Any character
    Assigned        Any non-Cn character (i.e. synonym for \P{Cn})
    Unassigned      Synonym for \p{Cn}
    Common          Any character (or unassigned code point)
                    not explicitly assigned to a script
```

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

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 256 Unicode characters. For example, the `Latin` script contains letters from many blocks but 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 a category called `Common`.

For more about scripts, see the UTR #24:

```
   http://www.unicode.org/unicode/reports/tr24/
```

For more about blocks, see:

```
   http://www.unicode.org/Public/UNIDATA/Blocks.txt
```

Block names are given with the `In` prefix. For example, the Katakana block is referenced via `\p{InKatakana}`. The `In` prefix may be omitted if there is no naming conflict with a script or any other property, but it is recommended that `In` always be used for block tests to avoid confusion.

These block names are supported:

```
    InAlphabeticPresentationForms
    InArabic
    InArabicPresentationFormsA
    InArabicPresentationFormsB
    InArmenian
    InArrows
    InBasicLatin
    InBengali
    InBlockElements
    InBopomofo
    InBopomofoExtended
    InBoxDrawing
    InBraillePatterns
    InBuhid
    InByzantineMusicalSymbols
    InCJKCompatibility
    InCJKCompatibilityForms
    InCJKCompatibilityIdeographs
    InCJKCompatibilityIdeographsSupplement
    InCJKRadicalsSupplement
    InCJKSymbolsAndPunctuation
    InCJKUnifiedIdeographs
    InCJKUnifiedIdeographsExtensionA
    InCJKUnifiedIdeographsExtensionB
    InCherokee
    InCombiningDiacriticalMarks
    InCombiningDiacriticalMarksforSymbols
    InCombiningHalfMarks
    InControlPictures
    InCurrencySymbols
    InCyrillic
    InCyrillicSupplementary
    InDeseret
    InDevanagari
    InDingbats
    InEnclosedAlphanumerics
    InEnclosedCJKLettersAndMonths
    InEthiopic
    InGeneralPunctuation
    InGeometricShapes
    InGeorgian
    InGothic
    InGreekExtended
    InGreekAndCoptic
    InGujarati
    InGurmukhi
    InHalfwidthAndFullwidthForms
    InHangulCompatibilityJamo
    InHangulJamo
    InHangulSyllables
    InHanunoo
    InHebrew
    InHighPrivateUseSurrogates
    InHighSurrogates
    InHiragana
    InIPAExtensions
    InIdeographicDescriptionCharacters
    InKanbun
    InKangxiRadicals
    InKannada
    InKatakana
    InKatakanaPhoneticExtensions
    InKhmer
    InLao
    InLatin1Supplement
    InLatinExtendedA
    InLatinExtendedAdditional
    InLatinExtendedB
    InLetterlikeSymbols
    InLowSurrogates
    InMalayalam
    InMathematicalAlphanumericSymbols
    InMathematicalOperators
    InMiscellaneousMathematicalSymbolsA
    InMiscellaneousMathematicalSymbolsB
    InMiscellaneousSymbols
    InMiscellaneousTechnical
    InMongolian
    InMusicalSymbols
    InMyanmar
    InNumberForms
    InOgham
    InOldItalic
    InOpticalCharacterRecognition
    InOriya
    InPrivateUseArea
    InRunic
    InSinhala
    InSmallFormVariants
    InSpacingModifierLetters
    InSpecials
    InSuperscriptsAndSubscripts
    InSupplementalArrowsA
    InSupplementalArrowsB
    InSupplementalMathematicalOperators
    InSupplementaryPrivateUseAreaA
    InSupplementaryPrivateUseAreaB
    InSyriac
    InTagalog
    InTagbanwa
    InTags
    InTamil
    InTelugu
    InThaana
    InThai
    InTibetan
    InUnifiedCanadianAboriginalSyllabics
```

```
InVariationSelectors
InYiRadicals
InYiSyllables
```

- The special pattern `\X` matches any extended Unicode sequence--"a combining character sequence" in Standardese--where the first character is a base character and subsequent characters are mark characters that apply to the base character. `\X` is equivalent to `(?:\PM\pM*)`.
- 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.
- 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()`. Operators that specifically do not switch include `vec()`, `pack()`, and `unpack()`. Operators that really don't care include operators that treats strings as a bucket of bits such as `sort()`, and operators dealing with filenames.
- The `pack()`/`unpack()` letters `c` and `C` do *not* change, since they are 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.
- The `chr()` and `ord()` functions work on characters, similar to `pack("U")` and `unpack("U")`, *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.
- lc(), uc(), lcfirst(), and ucfirst() work for the following cases:
  - the case mapping is from a single Unicode character to another single Unicode character, or
  - the case mapping is from a single Unicode character to more than one Unicode character.

  Things to do with locales (Lithuanian, Turkish, Azeri) do **not** work since Perl does not understand the concept of Unicode locales.

  See the Unicode Technical Report #21, Case Mappings, for more details.
- And finally, `scalar reverse()` reverses by character rather than by byte.

User-Defined Character Properties

You can define your own 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 IsForeign 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:

- 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 "!": 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).

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 the same: define subroutines in the `main` package 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 now to be three hexadecimal numbers separated by tabulators: start of the source range, end of the source range, and start of the destination range. For example:

```
   sub ToUpper {
return <<END;
   0061\t0063\t0041
   END
   }
```

defines an uc() mapping that causes only the characters "a", "b", and "c" to be mapped to "A", "B", "C", all other characters will remain unchanged.

If there is no source range to speak of, that is, the mapping is from a single character to another single character, leave the end of the source range empty, but the two tabulator characters are still needed. For example:

```
   sub ToLower {
return <<END;
   0041\t\t0061
   END
   }
```

defines a lc() mapping that causes only "A" to be mapped to "a", all other characters will remain unchanged.

(For serious hackers only) If you want to introspect the default mappings, you can find the data 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).

A final note on the user-defined property tests and mappings: they will be used only if the scalar has been marked as having Unicode characters. Old byte-style strings will not be affected.

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 Report 18, "Unicode Regular Expression Guidelines", version 6 (Unicode 3.2.0, Perl 5.8.0).

- Level 1 - Basic Unicode Support

```
        2.1 Hex Notation                        - done          [1]
            Named Notation                      - done          [2]
        2.2 Categories                          - done
[3][4]
        2.3 Subtraction                         - MISSING
[5][6]
        2.4 Simple Word Boundaries              - done          [7]
        2.5 Simple Loose Matches                - done          [8]
        2.6 End of Line                         - MISSING
[9][10]

        [ 1] \x{...}
        [ 2] \N{...}
        [ 3] . \p{...} \P{...}
        [ 4] support for scripts (see UTR#24 Script Names), blocks,
             binary properties, enumerated non-binary properties, and
             numeric properties (as listed in UTR#18 Other
Properties)
        [ 5] have negation
        [ 6] can use regular expression look-ahead [a]
             or user-defined character properties [b] to emulate
subtraction
        [ 7] include Letters in word characters
        [ 8] note that Perl does Full case-folding in matching, not
Simple:
             for example U+1F88 is equivalent with U+1F00 U+03B9,
             not with 1F80.  This difference matters 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.
        [ 9] see UTR #13 Unicode Newline Guidelines
        [10] should do ^ and $ also on \x{85}, \x{2028} and \x{2029}
```

```
             (should also affect <>, $., and script line numbers)
             (the \x{85}, \x{2028} and \x{2029} do match \s)
```

[a] You can mimic class subtraction using lookahead. For example, what UTR #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 UTR #18 grouping, intersection, union, and removal (subtraction) syntax.

[b] See *User-Defined Character Properties*.

- Level 2 - Extended Unicode Support

```
        3.1 Surrogates                          - MISSING [11]
        3.2 Canonical Equivalents               - MISSING
[12][13]
        3.3 Locale-Independent Graphemes        - MISSING       [14]
        3.4 Locale-Independent Words            - MISSING       [15]
        3.5 Locale-Independent Loose Matches    - MISSING       [16]

        [11] Surrogates are solely a UTF-16 concept and Perl’s
internal
             representation is UTF-8.  The Encode module does UTF-16,
though.
        [12] see UTR#15 Unicode Normalization
        [13] have Unicode::Normalize but not integrated to regexes
        [14] have \X but at this level . should equal that
        [15] need three classes, not just \w and \W
        [16] see UTR#21 Case Mappings
```

- Level 3 - Locale-Sensitive Support

```
        4.1 Locale-Dependent Categories         - MISSING
        4.2 Locale-Dependent Graphemes          - MISSING
[16][17]
        4.3 Locale-Dependent Words              - MISSING
        4.4 Locale-Dependent Loose Matches      - MISSING
        4.5 Locale-Dependent Ranges             - MISSING

        [16] see UTR#10 Unicode Collation Algorithms
        [17] have Unicode::Collate but not integrated to regexes
```

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       ******* ill-formed *******
   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 `A0..BF` in `U+0800..U+0FFF`, the `80..9F` in `U+D000...U+D7FF`, the `90..BF` in `U+10000..U+3FFFF`, and the `80...8F` in `U+100000..U+10FFFF`. The "gaps" 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. So that's 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     110bbbbb  10aaaaaa
            ccccbbbbbbaaaaaa     1110cccc  10bbbbbb  10aaaaaa
  00000dddccccccbbbbbbaaaaaa     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 the 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".

- 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 0xFE 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

- 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. 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 this 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?

- chmod, 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

Forcing Unicode in Perl (Or Unforcing Unicode in Perl)

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

Do not use them without careful thought, though: Perl may easily get very confused, angry, or even crash, if you suddenly change the 'nature' of scalar like that. Especially careful you have to be if you use the utf8::upgrade(): any random byte string is not valid UTF-8.

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 perlguts* 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 is 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.
- `uvuni_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.
- `utf8_to_uvuni(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.
- `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 an 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, pvlim, flags)` and `sv_uni_display(dsv, ssv, pvlim, 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, u1, 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

memEQ() and memNE() as usual.

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

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.

Interaction with Extensions

When Perl exchanges data with an extension, the extension should be able to understand the UTF-8 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).

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, ":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 UTF-8 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 UTF-8 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 UTF-8 flag:

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

SEE ALSO

perluniintro, *encoding*, *Encode*, *open*, *utf8*, *bytes*, *perlretut*, *"${^UNICODE}" in perlvar*