NAME

Unicode::UCD - Unicode character database

SYNOPSIS

```
use Unicode::UCD 'charinfo';
my $charinfo   = charinfo($codepoint);

use Unicode::UCD 'casefold';
my $casefold = casefold(0xFB00);

use Unicode::UCD 'all_casefolds';
my $all_casefolds_ref = all_casefolds();

use Unicode::UCD 'casespec';
my $casespec = casespec(0xFB00);

use Unicode::UCD 'charblock';
my $charblock  = charblock($codepoint);

use Unicode::UCD 'charscript';
my $charscript = charscript($codepoint);

use Unicode::UCD 'charblocks';
my $charblocks = charblocks();

use Unicode::UCD 'charscripts';
my $charscripts = charscripts();

use Unicode::UCD qw(charscript charinrange);
my $range = charscript($script);
print "looks like $script\n" if charinrange($range, $codepoint);

use Unicode::UCD qw(general_categories bidi_types);
my $categories = general_categories();
my $types = bidi_types();

use Unicode::UCD 'prop_aliases';
my @space_names = prop_aliases("space");

use Unicode::UCD 'prop_value_aliases';
my @gc_punct_names = prop_value_aliases("Gc", "Punct");

use Unicode::UCD 'prop_invlist';
my @puncts = prop_invlist("gc=punctuation");

use Unicode::UCD 'prop_invmap';
my ($list_ref, $map_ref, $format, $missing)
                                  = prop_invmap("General Category");

use Unicode::UCD 'compexcl';
my $compexcl = compexcl($codepoint);
```

```
use Unicode::UCD 'namedseq';
my $namedseq = namedseq($named_sequence_name);

my $unicode_version = Unicode::UCD::UnicodeVersion();

my $convert_to_numeric =
          Unicode::UCD::num("\N{RUMI DIGIT ONE}\N{RUMI DIGIT TWO}");
```

DESCRIPTION

The Unicode::UCD module offers a series of functions that provide a simple interface to the Unicode Character Database.

code point argument

Some of the functions are called with a *code point argument*, which is either a decimal or a hexadecimal scalar designating a Unicode code point, or `U+` followed by hexadecimals designating a Unicode code point. In other words, if you want a code point to be interpreted as a hexadecimal number, you must prefix it with either `0x` or `U+`, because a string like e.g. `123` will be interpreted as a decimal code point.

Examples:

```
223     # Decimal 223
0223    # Hexadecimal 223 (= 547 decimal)
0xDF    # Hexadecimal DF (= 223 decimal
U+DF    # Hexadecimal DF
```

Note that the largest code point in Unicode is U+10FFFF.

charinfo()

```
use Unicode::UCD 'charinfo';

my $charinfo = charinfo(0x41);
```

This returns information about the input *code point argument* as a reference to a hash of fields as defined by the Unicode standard. If the *code point argument* is not assigned in the standard (i.e., has the general category `Cn` meaning `Unassigned`) or is a non-character (meaning it is guaranteed to never be assigned in the standard), `undef` is returned.

Fields that aren't applicable to the particular code point argument exist in the returned hash, and are empty.

The keys in the hash with the meanings of their values are:

code

the input *code point argument* expressed in hexadecimal, with leading zeros added if necessary to make it contain at least four hexdigits

name

name of *code*, all IN UPPER CASE. Some control-type code points do not have names. This field will be empty for `Surrogate` and `Private Use` code points, and for the others without a name, it will contain a description enclosed in angle brackets, like `<control>`.

category

The short name of the general category of *code*. This will match one of the keys in the hash returned by *general_categories()*.

The *prop_value_aliases()* function can be used to get all the synonyms of the category name.

combining

the combining class number for *code* used in the Canonical Ordering Algorithm. For Unicode 5.1, this is described in Section 3.11 `Canonical Ordering Behavior` available at *http://www.unicode.org/versions/Unicode5.1.0/*

The *prop_value_aliases()* function can be used to get all the synonyms of the combining class number.

bidi

bidirectional type of *code*. This will match one of the keys in the hash returned by *bidi_types()*.

The *prop_value_aliases()* function can be used to get all the synonyms of the bidi type name.

decomposition

is empty if *code* has no decomposition; or is one or more codes (separated by spaces) that, taken in order, represent a decomposition for *code*. Each has at least four hexdigits. The codes may be preceded by a word enclosed in angle brackets then a space, like `<compat>` , giving the type of decomposition

This decomposition may be an intermediate one whose components are also decomposable. Use *Unicode::Normalize* to get the final decomposition.

decimal

if *code* is a decimal digit this is its integer numeric value

digit

if *code* represents some other digit-like number, this is its integer numeric value

numeric

if *code* represents a whole or rational number, this is its numeric value. Rational values are expressed as a string like `1/4`.

mirrored

`Y` or `N` designating if *code* is mirrored in bidirectional text

unicode10

name of *code* in the Unicode 1.0 standard if one existed for this code point and is different from the current name

comment

As of Unicode 6.0, this is always empty.

upper

is empty if there is no single code point uppercase mapping for *code* (its uppercase mapping is itself); otherwise it is that mapping expressed as at least four hexdigits. (*casespec()* should be used in addition to **charinfo()** for case mappings when the calling program can cope with multiple code point mappings.)

lower

is empty if there is no single code point lowercase mapping for *code* (its lowercase mapping is itself); otherwise it is that mapping expressed as at least four hexdigits. (*casespec()* should be used in addition to **charinfo()** for case mappings when the calling program can cope with multiple code point mappings.)

title

is empty if there is no single code point titlecase mapping for *code* (its titlecase mapping is

itself); otherwise it is that mapping expressed as at least four hexdigits. (*casespec()* should be used in addition to **charinfo()** for case mappings when the calling program can cope with multiple code point mappings.)

block

the block *code* belongs to (used in `\p{Blk=...}`). See *Blocks versus Scripts*.

script

the script *code* belongs to. See *Blocks versus Scripts*.

Note that you cannot do (de)composition and casing based solely on the *decomposition*, *combining*, *lower*, *upper*, and *title* fields; you will need also the *compexcl()*, and *casespec()* functions.

charblock()

```
use Unicode::UCD 'charblock';

my $charblock = charblock(0x41);
my $charblock = charblock(1234);
my $charblock = charblock(0x263a);
my $charblock = charblock("U+263a");

my $range     = charblock('Armenian');
```

With a *code point argument* charblock() returns the *block* the code point belongs to, e.g. `Basic Latin`. The old-style block name is returned (see *Old-style versus new-style block names*). If the code point is unassigned, this returns the block it would belong to if it were assigned. (If the Unicode version being used is so early as to not have blocks, all code points are considered to be in `No_Block`.)

See also *Blocks versus Scripts*.

If supplied with an argument that can't be a code point, charblock() tries to do the opposite and interpret the argument as an old-style block name. The return value is a *range set* with one range: an anonymous list with a single element that consists of another anonymous list whose first element is the first code point in the block, and whose second (and final) element is the final code point in the block. (The extra list consisting of just one element is so that the same program logic can be used to handle both this return, and the return from *charscript()* which can have multiple ranges.) You can test whether a code point is in a range using the *charinrange()* function. If the argument is not a known block, `undef` is returned.

charscript()

```
use Unicode::UCD 'charscript';

my $charscript = charscript(0x41);
my $charscript = charscript(1234);
my $charscript = charscript("U+263a");

my $range      = charscript('Thai');
```

With a *code point argument* charscript() returns the *script* the code point belongs to, e.g. `Latin`, `Greek`, `Han`. If the code point is unassigned or the Unicode version being used is so early that it doesn't have scripts, this function returns `"Unknown"`.

If supplied with an argument that can't be a code point, charscript() tries to do the opposite and interpret the argument as a script name. The return value is a *range set*: an anonymous list of lists

that contain *start-of-range*, *end-of-range* code point pairs. You can test whether a code point is in a range set using the *charinrange()* function. If the argument is not a known script, `undef` is returned.

See also *Blocks versus Scripts*.

charblocks()

```
use Unicode::UCD 'charblocks';

my $charblocks = charblocks();
```

charblocks() returns a reference to a hash with the known block names as the keys, and the code point ranges (see *charblock()*) as the values.

The names are in the old-style (see *Old-style versus new-style block names*).

prop_invmap("block") can be used to get this same data in a different type of data structure.

See also *Blocks versus Scripts*.

charscripts()

```
use Unicode::UCD 'charscripts';

my $charscripts = charscripts();
```

charscripts() returns a reference to a hash with the known script names as the keys, and the code point ranges (see *charscript()*) as the values.

prop_invmap("script") can be used to get this same data in a different type of data structure.

See also *Blocks versus Scripts*.

charinrange()

In addition to using the `\p{Blk=...}` and `\P{Blk=...}` constructs, you can also test whether a code point is in the *range* as returned by *charblock()* and *charscript()* or as the values of the hash returned by *charblocks()* and *charscripts()* by using charinrange():

```
use Unicode::UCD qw(charscript charinrange);

$range = charscript('Hiragana');
print "looks like hiragana\n" if charinrange($range, $codepoint);
```

general_categories()

```
use Unicode::UCD 'general_categories';

my $categories = general_categories();
```

This returns a reference to a hash which has short general category names (such as `Lu`, `Nd`, `Zs`, `S`) as keys and long names (such as `UppercaseLetter`, `DecimalNumber`, `SpaceSeparator`, `Symbol`) as values. The hash is reversible in case you need to go from the long names to the short names. The general category is the one returned from *charinfo()* under the `category` key.

The *prop_value_aliases()* function can be used to get all the synonyms of the category name.

bidi_types()

```
use Unicode::UCD 'bidi_types';
```

```
my $categories = bidi_types();
```

This returns a reference to a hash which has the short bidi (bidirectional) type names (such as `L`, `R`) as keys and long names (such as `Left-to-Right`, `Right-to-Left`) as values. The hash is reversible in case you need to go from the long names to the short names. The bidi type is the one returned from *charinfo()* under the `bidi` key. For the exact meaning of the various bidi classes the Unicode TR9 is recommended reading: *http://www.unicode.org/reports/tr9/* (as of Unicode 5.0.0)

The *prop_value_aliases()* function can be used to get all the synonyms of the bidi type name.

compexcl()

```
use Unicode::UCD 'compexcl';

my $compexcl = compexcl(0x09dc);
```

This routine returns `undef` if the Unicode version being used is so early that it doesn't have this property. It is included for backwards compatibility, but as of Perl 5.12 and more modern Unicode versions, for most purposes it is probably more convenient to use one of the following instead:

```
my $compexcl = chr(0x09dc) =~ /\p{Comp_Ex};
my $compexcl = chr(0x09dc) =~ /\p{Full_Composition_Exclusion};
```

or even

```
my $compexcl = chr(0x09dc) =~ /\p{CE};
my $compexcl = chr(0x09dc) =~ /\p{Composition_Exclusion};
```

The first two forms return **true** if the *code point argument* should not be produced by composition normalization. For the final two forms to return **true**, it is additionally required that this fact not otherwise be determinable from the Unicode data base.

This routine behaves identically to the final two forms. That is, it does not return **true** if the code point has a decomposition consisting of another single code point, nor if its decomposition starts with a code point whose combining class is non-zero. Code points that meet either of these conditions should also not be produced by composition normalization, which is probably why you should use the `Full_Composition_Exclusion` property instead, as shown above.

The routine returns **false** otherwise.

casefold()

```
use Unicode::UCD 'casefold';

my $casefold = casefold(0xDF);
if (defined $casefold) {
    my @full_fold_hex = split / /, $casefold->{'full'};
    my $full_fold_string =
                join "", map {chr(hex($_))} @full_fold_hex;
    my @turkic_fold_hex =
                    split / /, ($casefold->{'turkic'} ne "")
                                    ? $casefold->{'turkic'}
                                    : $casefold->{'full'};
    my $turkic_fold_string =
                    join "", map {chr(hex($_))} @turkic_fold_hex;
}
if (defined $casefold && $casefold->{'simple'} ne "") {
    my $simple_fold_hex = $casefold->{'simple'};
```

```
        my $simple_fold_string = chr(hex($simple_fold_hex));
    }
```

This returns the (almost) locale-independent case folding of the character specified by the *code point argument*. (Starting in Perl v5.16, the core function `fc()` returns the `full` mapping (described below) faster than this does, and for entire strings.)

If there is no case folding for the input code point, `undef` is returned.

If there is a case folding for that code point, a reference to a hash with the following fields is returned:

code

the input *code point argument* expressed in hexadecimal, with leading zeros added if necessary to make it contain at least four hexdigits

full

one or more codes (separated by spaces) that, taken in order, give the code points for the case folding for *code*. Each has at least four hexdigits.

simple

is empty, or is exactly one code with at least four hexdigits which can be used as an alternative case folding when the calling program cannot cope with the fold being a sequence of multiple code points. If *full* is just one code point, then *simple* equals *full*. If there is no single code point folding defined for *code*, then *simple* is the empty string. Otherwise, it is an inferior, but still better-than-nothing alternative folding to *full*.

mapping

is the same as *simple* if *simple* is not empty, and it is the same as *full* otherwise. It can be considered to be the simplest possible folding for *code*. It is defined primarily for backwards compatibility.

status

is `C` (for `common`) if the best possible fold is a single code point (*simple* equals *full* equals *mapping*). It is `S` if there are distinct folds, *simple* and *full* (*mapping* equals *simple*). And it is `F` if there is only a *full* fold (*mapping* equals *full*; *simple* is empty). Note that this describes the contents of *mapping*. It is defined primarily for backwards compatibility.

For Unicode versions between 3.1 and 3.1.1 inclusive, *status* can also be `I` which is the same as `C` but is a special case for dotted uppercase I and dotless lowercase i:

***** If you use this `I` mapping

the result is case-insensitive, but dotless and dotted I's are not distinguished

***** If you exclude this `I` mapping

the result is not fully case-insensitive, but dotless and dotted I's are distinguished

turkic

contains any special folding for Turkic languages. For versions of Unicode starting with 3.2, this field is empty unless *code* has a different folding in Turkic languages, in which case it is one or more codes (separated by spaces) that, taken in order, give the code points for the case folding for *code* in those languages. Each code has at least four hexdigits. Note that this folding does not maintain canonical equivalence without additional processing.

For Unicode versions between 3.1 and 3.1.1 inclusive, this field is empty unless there is a special folding for Turkic languages, in which case *status* is `I`, and *mapping*, *full*, *simple*, and *turkic* are all equal.

Programs that want complete generality and the best folding results should use the folding contained

in the *full* field. But note that the fold for some code points will be a sequence of multiple code points.

Programs that can't cope with the fold mapping being multiple code points can use the folding contained in the *simple* field, with the loss of some generality. In Unicode 5.1, about 7% of the defined foldings have no single code point folding.

The *mapping* and *status* fields are provided for backwards compatibility for existing programs. They contain the same values as in previous versions of this function.

Locale is not completely independent. The *turkic* field contains results to use when the locale is a Turkic language.

For more information about case mappings see *http://www.unicode.org/unicode/reports/tr21*

all_casefolds()

```
use Unicode::UCD 'all_casefolds';

my $all_folds_ref = all_casefolds();
foreach my $char_with_casefold (sort { $a <=> $b }
                                keys %$all_folds_ref)
{
    printf "%04X:", $char_with_casefold;
    my $casefold = $all_folds_ref->{$char_with_casefold};

    # Get folds for $char_with_casefold

    my @full_fold_hex = split / /, $casefold->{'full'};
    my $full_fold_string =
                join "", map {chr(hex($_))} @full_fold_hex;
    print " full=", join " ", @full_fold_hex;
    my @turkic_fold_hex =
                   split / /, ($casefold->{'turkic'} ne "")
                                    ? $casefold->{'turkic'}
                                    : $casefold->{'full'};
    my $turkic_fold_string =
                   join "", map {chr(hex($_))} @turkic_fold_hex;
    print "; turkic=", join " ", @turkic_fold_hex;
    if (defined $casefold && $casefold->{'simple'} ne "") {
        my $simple_fold_hex = $casefold->{'simple'};
        my $simple_fold_string = chr(hex($simple_fold_hex));
        print "; simple=$simple_fold_hex";
    }
    print "\n";
}
```

This returns all the case foldings in the current version of Unicode in the form of a reference to a hash. Each key to the hash is the decimal representation of a Unicode character that has a casefold to other than itself. The casefold of a semi-colon is itself, so it isn't in the hash; likewise for a lowercase "a", but there is an entry for a capital "A". The hash value for each key is another hash, identical to what is returned by *casefold()* if called with that code point as its argument. So the value `all_casefolds()->{ord("A")}` is equivalent to `casefold(ord("A"))`;

casespec()

```
use Unicode::UCD 'casespec';

my $casespec = casespec(0xFB00);
```

This returns the potentially locale-dependent case mappings of the *code point argument*. The mappings may be longer than a single code point (which the basic Unicode case mappings as returned by *charinfo()* never are).

If there are no case mappings for the *code point argument*, or if all three possible mappings (*lower*, *title* and *upper*) result in single code points and are locale independent and unconditional, `undef` is returned (which means that the case mappings, if any, for the code point are those returned by *charinfo()*).

Otherwise, a reference to a hash giving the mappings (or a reference to a hash of such hashes, explained below) is returned with the following keys and their meanings:

The keys in the bottom layer hash with the meanings of their values are:

code

the input *code point argument* expressed in hexadecimal, with leading zeros added if necessary to make it contain at least four hexdigits

lower

one or more codes (separated by spaces) that, taken in order, give the code points for the lower case of *code*. Each has at least four hexdigits.

title

one or more codes (separated by spaces) that, taken in order, give the code points for the title case of *code*. Each has at least four hexdigits.

upper

one or more codes (separated by spaces) that, taken in order, give the code points for the upper case of *code*. Each has at least four hexdigits.

condition

the conditions for the mappings to be valid. If `undef`, the mappings are always valid. When defined, this field is a list of conditions, all of which must be true for the mappings to be valid. The list consists of one or more *locales* (see below) and/or *contexts* (explained in the next paragraph), separated by spaces. (Other than as used to separate elements, spaces are to be ignored.) Case distinctions in the condition list are not significant. Conditions preceded by "NON_" represent the negation of the condition.

A *context* is one of those defined in the Unicode standard. For Unicode 5.1, they are defined in Section 3.13 `Default Case Operations` available at *http://www.unicode.org/versions/Unicode5.1.0/*. These are for context-sensitive casing.

The hash described above is returned for locale-independent casing, where at least one of the mappings has length longer than one. If `undef` is returned, the code point may have mappings, but if so, all are length one, and are returned by *charinfo()*. Note that when this function does return a value, it will be for the complete set of mappings for a code point, even those whose length is one.

If there are additional casing rules that apply only in certain locales, an additional key for each will be defined in the returned hash. Each such key will be its locale name, defined as a 2-letter ISO 3166 country code, possibly followed by a "_" and a 2-letter ISO language code (possibly followed by a "_" and a variant code). You can find the lists of all possible locales, see *Locale::Country* and *Locale::Language*. (In Unicode 6.0, the only locales returned by this function are `lt`, `tr`, and `az`.)

Each locale key is a reference to a hash that has the form above, and gives the casing rules for that particular locale, which take precedence over the locale-independent ones when in that locale.

If the only casing for a code point is locale-dependent, then the returned hash will not have any of the base keys, like `code`, `upper`, etc., but will contain only locale keys.

For more information about case mappings see *http://www.unicode.org/unicode/reports/tr21/*

namedseq()

```
use Unicode::UCD 'namedseq';

my $namedseq = namedseq("KATAKANA LETTER AINU P");
my @namedseq = namedseq("KATAKANA LETTER AINU P");
my %namedseq = namedseq();
```

If used with a single argument in a scalar context, returns the string consisting of the code points of the named sequence, or `undef` if no named sequence by that name exists. If used with a single argument in a list context, it returns the list of the ordinals of the code points. If used with no arguments in a list context, returns a hash with the names of the named sequences as the keys and the named sequences as strings as the values. Otherwise, it returns `undef` or an empty list depending on the context.

This function only operates on officially approved (not provisional) named sequences.

Note that as of Perl 5.14, `\N{KATAKANA LETTER AINU P}` will insert the named sequence into double-quoted strings, and `charnames::string_vianame("KATAKANA LETTER AINU P")` will return the same string this function does, but will also operate on character names that aren't named sequences, without you having to know which are which. See *charnames*.

num()

```
use Unicode::UCD 'num';

my $val = num("123");
my $one_quarter = num("\N{VULGAR FRACTION 1/4}");
```

`num` returns the numeric value of the input Unicode string; or `undef` if it doesn't think the entire string has a completely valid, safe numeric value.

If the string is just one character in length, the Unicode numeric value is returned if it has one, or `undef` otherwise. Note that this need not be a whole number. `num("\N{TIBETAN DIGIT HALF ZERO}")`, for example returns -0.5.

If the string is more than one character, `undef` is returned unless all its characters are decimal digits (that is, they would match `\d+`), from the same script. For example if you have an ASCII '0' and a Bengali '3', mixed together, they aren't considered a valid number, and `undef` is returned. A further restriction is that the digits all have to be of the same form. A half-width digit mixed with a full-width one will return `undef`. The Arabic script has two sets of digits; `num` will return `undef` unless all the digits in the string come from the same set.

`num` errs on the side of safety, and there may be valid strings of decimal digits that it doesn't recognize. Note that Unicode defines a number of "digit" characters that aren't "decimal digit" characters. "Decimal digits" have the property that they have a positional value, i.e., there is a units position, a 10's position, a 100's, etc, AND they are arranged in Unicode in blocks of 10 contiguous code points. The Chinese digits, for example, are not in such a contiguous block, and so Unicode doesn't view them as decimal digits, but merely digits, and so `\d` will not match them. A single-character string containing one of these digits will have its decimal value returned by `num`, but any longer string containing only these digits will return `undef`.

Strings of multiple sub- and superscripts are not recognized as numbers. You can use either of the compatibility decompositions in Unicode::Normalize to change these into digits, and then call `num` on the result.

prop_aliases()

```
use Unicode::UCD 'prop_aliases';

my ($short_name, $full_name, @other_names) = prop_aliases("space");
my $same_full_name = prop_aliases("Space");     # Scalar context
my ($same_short_name) = prop_aliases("Space");  # gets 0th element
print "The full name is $full_name\n";
print "The short name is $short_name\n";
print "The other aliases are: ", join(", ", @other_names), "\n";

prints:
The full name is White_Space
The short name is WSpace
The other aliases are: Space
```

Most Unicode properties have several synonymous names. Typically, there is at least a short name, convenient to type, and a long name that more fully describes the property, and hence is more easily understood.

If you know one name for a Unicode property, you can use `prop_aliases` to find either the long name (when called in scalar context), or a list of all of the names, somewhat ordered so that the short name is in the 0th element, the long name in the next element, and any other synonyms are in the remaining elements, in no particular order.

The long name is returned in a form nicely capitalized, suitable for printing.

The input parameter name is loosely matched, which means that white space, hyphens, and underscores are ignored (except for the trailing underscore in the old_form grandfathered-in `"L_"`, which is better written as `"LC"`, and both of which mean `General_Category=Cased Letter`).

If the name is unknown, `undef` is returned (or an empty list in list context). Note that Perl typically recognizes property names in regular expressions with an optional `"Is_"` (with or without the underscore) prefixed to them, such as `\p{isgc=punct}`. This function does not recognize those in the input, returning `undef`. Nor are they included in the output as possible synonyms.

`prop_aliases` does know about the Perl extensions to Unicode properties, such as `Any` and `XPosixAlpha`, and the single form equivalents to Unicode properties such as `XDigit`, `Greek`, `In_Greek`, and `Is_Greek`. The final example demonstrates that the `"Is_"` prefix is recognized for these extensions; it is needed to resolve ambiguities. For example, `prop_aliases('lc')` returns the list `(lc, Lowercase_Mapping)`, but `prop_aliases('islc')` returns `(Is_LC, Cased_Letter)`. This is because `islc` is a Perl extension which is short for `General_Category=Cased Letter`. The lists returned for the Perl extensions will not include the `"Is_"` prefix (whether or not the input had it) unless needed to resolve ambiguities, as shown in the `"islc"` example, where the returned list had one element containing `"Is_"`, and the other without.

It is also possible for the reverse to happen: `prop_aliases('isc')` returns the list `(isc, ISO_Comment)`; whereas `prop_aliases('c')` returns `(C, Other)` (the latter being a Perl extension meaning `General_Category=Other`. *"Properties accessible through Unicode::UCD" in perluniprops* lists the available forms, including which ones are discouraged from use.

Those discouraged forms are accepted as input to `prop_aliases`, but are not returned in the lists. `prop_aliases('isL&')` and `prop_aliases('isL_')`, which are old synonyms for `"Is_LC"` and should not be used in new code, are examples of this. These both return `(Is_LC, Cased_Letter)`. Thus this function allows you to take a discourarged form, and find its acceptable alternatives. The same goes with single-form Block property equivalences. Only the forms that begin with `"In_"` are not discouraged; if you pass `prop_aliases` a discouraged form, you will get back the equivalent ones that begin with `"In_"`. It will otherwise look like a new-style block name (see.

Old-style versus new-style block names).

`prop_aliases` does not know about any user-defined properties, and will return `undef` if called with one of those. Likewise for Perl internal properties, with the exception of "Perl_Decimal_Digit" which it does know about (and which is documented below in *prop_invmap()*).

prop_value_aliases()

```
use Unicode::UCD 'prop_value_aliases';

my ($short_name, $full_name, @other_names)
                                = prop_value_aliases("Gc", "Punct");
my $same_full_name = prop_value_aliases("Gc", "P");   # Scalar cntxt
my ($same_short_name) = prop_value_aliases("Gc", "P"); # gets 0th
                                                       # element
print "The full name is $full_name\n";
print "The short name is $short_name\n";
print "The other aliases are: ", join(", ", @other_names), "\n";

prints:
The full name is Punctuation
The short name is P
The other aliases are: Punct
```

Some Unicode properties have a restricted set of legal values. For example, all binary properties are restricted to just `true` or `false`; and there are only a few dozen possible General Categories.

For such properties, there are usually several synonyms for each possible value. For example, in binary properties, *truth* can be represented by any of the strings "Y", "Yes", "T", or "True"; and the General Category "Punctuation" by that string, or "Punct", or simply "P".

Like property names, there is typically at least a short name for each such property-value, and a long name. If you know any name of the property-value, you can use `prop_value_aliases`() to get the long name (when called in scalar context), or a list of all the names, with the short name in the 0th element, the long name in the next element, and any other synonyms in the remaining elements, in no particular order, except that any all-numeric synonyms will be last.

The long name is returned in a form nicely capitalized, suitable for printing.

Case, white space, hyphens, and underscores are ignored in the input parameters (except for the trailing underscore in the old-form grandfathered-in general category property value `"L_"`, which is better written as `"LC"`).

If either name is unknown, `undef` is returned. Note that Perl typically recognizes property names in regular expressions with an optional `"Is_"` (with or without the underscore) prefixed to them, such as `\p{isgc=punct}`. This function does not recognize those in the property parameter, returning `undef`.

If called with a property that doesn't have synonyms for its values, it returns the input value, possibly normalized with capitalization and underscores.

For the block property, new-style block names are returned (see *Old-style versus new-style block names*).

To find the synonyms for single-forms, such as `\p{Any}`, use *prop_aliases()* instead.

`prop_value_aliases` does not know about any user-defined properties, and will return `undef` if called with one of those.

prop_invlist()

`prop_invlist` returns an inversion list (described below) that defines all the code points for the binary Unicode property (or "property=value" pair) given by the input parameter string:

```
use feature 'say';
use Unicode::UCD 'prop_invlist';
say join ", ", prop_invlist("Any");

prints:
0, 1114112
```

If the input is unknown `undef` is returned in scalar context; an empty-list in list context. If the input is known, the number of elements in the list is returned if called in scalar context.

perluniprops gives the list of properties that this function accepts, as well as all the possible forms for them (including with the optional "Is_" prefixes). (Except this function doesn't accept any Perl-internal properties, some of which are listed there.) This function uses the same loose or tighter matching rules for resolving the input property's name as is done for regular expressions. These are also specified in *perluniprops*. Examples of using the "property=value" form are:

```
say join ", ", prop_invlist("Script=Shavian");

prints:
66640, 66688

say join ", ", prop_invlist("ASCII_Hex_Digit=No");

prints:
0, 48, 58, 65, 71, 97, 103

say join ", ", prop_invlist("ASCII_Hex_Digit=Yes");

prints:
48, 58, 65, 71, 97, 103
```

Inversion lists are a compact way of specifying Unicode property-value definitions. The 0th item in the list is the lowest code point that has the property-value. The next item (item [1]) is the lowest code point beyond that one that does NOT have the property-value. And the next item beyond that ([2]) is the lowest code point beyond that one that does have the property-value, and so on. Put another way, each element in the list gives the beginning of a range that has the property-value (for even numbered elements), or doesn't have the property-value (for odd numbered elements). The name for this data structure stems from the fact that each element in the list toggles (or inverts) whether the corresponding range is or isn't on the list.

In the final example above, the first ASCII Hex digit is code point 48, the character "0", and all code points from it through 57 (a "9") are ASCII hex digits. Code points 58 through 64 aren't, but 65 (an "A") through 70 (an "F") are, as are 97 ("a") through 102 ("f"). 103 starts a range of code points that aren't ASCII hex digits. That range extends to infinity, which on your computer can be found in the variable `$Unicode::UCD::MAX_CP`. (This variable is as close to infinity as Perl can get on your platform, and may be too high for some operations to work; you may wish to use a smaller number for your purposes.)

Note that the inversion lists returned by this function can possibly include non-Unicode code points, that is anything above 0x10FFFF. This is in contrast to Perl regular expression matches on those code points, in which a non-Unicode code point always fails to match. For example, both of these

have the same result:

```
chr(0x110000) =~ \p{ASCII_Hex_Digit=True}      # Fails.
chr(0x110000) =~ \p{ASCII_Hex_Digit=False}     # Fails!
```

And both raise a warning that a Unicode property is being used on a non-Unicode code point. It is arguable as to which is the correct thing to do here. This function has chosen the way opposite to the Perl regular expression behavior. This allows you to easily flip to to the Perl regular expression way (for you to go in the other direction would be far harder). Simply add 0x110000 at the end of the non-empty returned list if it isn't already that value; and pop that value if it is; like:

```
my @list = prop_invlist("foo");
if (@list) {
    if ($list[-1] == 0x110000) {
        pop @list;  # Defeat the turning on for above Unicode
    }
    else {
        push @list, 0x110000; # Turn off for above Unicode
    }
}
```

It is a simple matter to expand out an inversion list to a full list of all code points that have the property-value:

```
my @invlist = prop_invlist($property_name);
die "empty" unless @invlist;
my @full_list;
for (my $i = 0; $i < @invlist; $i += 2) {
   my $upper = ($i + 1) < @invlist
               ? $invlist[$i+1] - 1      # In range
               : $Unicode::UCD::MAX_CP;  # To infinity.  You may want
                                         # to stop much much earlier;
                                         # going this high may expose
                                         # perl deficiencies with very
                                         # large numbers.
   for my $j ($invlist[$i] .. $upper) {
       push @full_list, $j;
   }
}
```

`prop_invlist` does not know about any user-defined nor Perl internal-only properties, and will return `undef` if called with one of those.

prop_invmap()

```
use Unicode::UCD 'prop_invmap';
my ($list_ref, $map_ref, $format, $missing)
                                    = prop_invmap("General Category");
```

`prop_invmap` is used to get the complete mapping definition for a property, in the form of an inversion map. An inversion map consists of two parallel arrays. One is an ordered list of code points that mark range beginnings, and the other gives the value (or mapping) that all code points in the corresponding range have.

`prop_invmap` is called with the name of the desired property. The name is loosely matched, meaning that differences in case, white-space, hyphens, and underscores are not meaningful (except for the trailing underscore in the old-form grandfathered-in property `"L_"`, which is better written as

"LC", or even better, "Gc=LC").

Many Unicode properties have more than one name (or alias). prop_invmap understands all of these, including Perl extensions to them. Ambiguities are resolved as described above for *prop_aliases()*. The Perl internal property "Perl_Decimal_Digit, described below, is also accepted. undef is returned if the property name is unknown. See *"Properties accessible through Unicode::UCD" in perluniprops* for the properties acceptable as inputs to this function.

It is a fatal error to call this function except in list context.

In addition to the the two arrays that form the inversion map, prop_invmap returns two other values; one is a scalar that gives some details as to the format of the entries of the map array; the other is used for specialized purposes, described at the end of this section.

This means that prop_invmap returns a 4 element list. For example,

```
my ($blocks_ranges_ref, $blocks_maps_ref, $format, $default)
                                                = prop_invmap("Block");
```

In this call, the two arrays will be populated as shown below (for Unicode 6.0):

```
Index  @blocks_ranges  @blocks_maps
  0        0x0000      Basic Latin
  1        0x0080      Latin-1 Supplement
  2        0x0100      Latin Extended-A
  3        0x0180      Latin Extended-B
  4        0x0250      IPA Extensions
  5        0x02B0      Spacing Modifier Letters
  6        0x0300      Combining Diacritical Marks
  7        0x0370      Greek and Coptic
  8        0x0400      Cyrillic
 ...
233        0x2B820     No_Block
234        0x2F800     CJK Compatibility Ideographs Supplement
235        0x2FA20     No_Block
236        0xE0000     Tags
237        0xE0080     No_Block
238        0xE0100     Variation Selectors Supplement
239        0xE01F0     No_Block
240        0xF0000     Supplementary Private Use Area-A
241        0x100000    Supplementary Private Use Area-B
242        0x110000    No_Block
```

The first line (with Index [0]) means that the value for code point 0 is "Basic Latin". The entry "0x0080" in the @blocks_ranges column in the second line means that the value from the first line, "Basic Latin", extends to all code points in the range from 0 up to but not including 0x0080, that is, through 127. In other words, the code points from 0 to 127 are all in the "Basic Latin" block. Similarly, all code points in the range from 0x0080 up to (but not including) 0x0100 are in the block named "Latin-1 Supplement", etc. (Notice that the return is the old-style block names; see *Old-style versus new-style block names*).

The final line (with Index [242]) means that the value for all code points above the legal Unicode maximum code point have the value "No_Block", which is the term Unicode uses for a non-existing block.

The arrays completely specify the mappings for all possible code points. The final element in an inversion map returned by this function will always be for the range that consists of all the code points that aren't legal Unicode, but that are expressible on the platform. (That is, it starts with code point

0x110000, the first code point above the legal Unicode maximum, and extends to infinity.) The value for that range will be the same that any typical unassigned code point has for the specified property. (Certain unassigned code points are not "typical"; for example the non-character code points, or those in blocks that are to be written right-to-left. The above-Unicode range's value is not based on these atypical code points.) It could be argued that, instead of treating these as unassigned Unicode code points, the value for this range should be `undef`. If you wish, you can change the returned arrays accordingly.

The maps are almost always simple scalars that should be interpreted as-is. These values are those given in the Unicode-supplied data files, which may be inconsistent as to capitalization and as to which synonym for a property-value is given. The results may be normalized by using the *prop_value_aliases()* function.

There are exceptions to the simple scalar maps. Some properties have some elements in their map list that are themselves lists of scalars; and some special strings are returned that are not to be interpreted as-is. Element [2] (placed into `$format` in the example above) of the returned four element list tells you if the map has any of these special elements or not, as follows:

`s`

means all the elements of the map array are simple scalars, with no special elements. Almost all properties are like this, like the `block` example above.

`sl`

means that some of the map array elements have the form given by `"s"`, and the rest are lists of scalars. For example, here is a portion of the output of calling `prop_invmap`() with the "Script Extensions" property:

```
 @scripts_ranges  @scripts_maps
      ...
      0x0953      Devanagari
      0x0964      [ Bengali, Devanagari, Gurumukhi, Oriya ]
      0x0966      Devanagari
      0x0970      Common
```

Here, the code points 0x964 and 0x965 are both used in Bengali, Devanagari, Gurmukhi, and Oriya, but no other scripts.

The Name_Alias property is also of this form. But each scalar consists of two components: 1) the name, and 2) the type of alias this is. They are separated by a colon and a space. In Unicode 6.1, there are several alias types:

`correction`

indicates that the name is a corrected form for the original name (which remains valid) for the same code point.

`control`

adds a new name for a control character.

`alternate`

is an alternate name for a character

`figment`

is a name for a character that has been documented but was never in any actual standard.

`abbreviation`

is a common abbreviation for a character

The lists are ordered (roughly) so the most preferred names come before less preferred ones.

For example,

```
 @aliases_ranges        @alias_maps
    ...
    0x009E        [ 'PRIVACY MESSAGE: control', 'PM: abbreviation' ]
    0x009F        [ 'APPLICATION PROGRAM COMMAND: control',
                    'APC: abbreviation'
                  ]
    0x00A0        'NBSP: abbreviation'
    0x00A1        ""
    0x00AD        'SHY: abbreviation'
    0x00AE        ""
    0x01A2        'LATIN CAPITAL LETTER GHA: correction'
    0x01A3        'LATIN SMALL LETTER GHA: correction'
    0x01A4        ""
    ...
```

A map to the empty string means that there is no alias defined for the code point.

a

is like `"s"` in that all the map array elements are scalars, but here they are restricted to all being integers, and some have to be adjusted (hence the name `"a"`) to get the correct result. For example, in:

```
 my ($uppers_ranges_ref, $uppers_maps_ref, $format)
                           = prop_invmap("Simple_Uppercase_Mapping");
```

the returned arrays look like this:

```
 @$uppers_ranges_ref    @$uppers_maps_ref   Note
       0                       0
      97                      65          'a' maps to 'A', b => B ...
     123                       0
     181                     924          MICRO SIGN => Greek Cap MU
     182                       0
     ...
```

Let's start with the second line. It says that the uppercase of code point 97 is 65; or `uc("a") == "A"`. But the line is for the entire range of code points 97 through 122. To get the mapping for any code point in a range, you take the offset it has from the beginning code point of the range, and add that to the mapping for that first code point. So, the mapping for 122 ("z") is derived by taking the offset of 122 from 97 (=25) and adding that to 65, yielding 90 ("z"). Likewise for everything in between.

The first line works the same way. The first map in a range is always the correct value for its code point (because the adjustment is 0). Thus the `uc(chr(0))` is just itself. Also, `uc(chr(1))` is also itself, as the adjustment is 0+1-0 .. `uc(chr(96))` is 96.

Requiring this simple adjustment allows the returned arrays to be significantly smaller than otherwise, up to a factor of 10, speeding up searching through them.

al

means that some of the map array elements have the form given by `"a"`, and the rest are ordered lists of code points. For example, in:

```
 my ($uppers_ranges_ref, $uppers_maps_ref, $format)
                                  = prop_invmap("Uppercase_Mapping");
```

the returned arrays look like this:

```
 @$uppers_ranges_ref    @$uppers_maps_ref
```

```
     0                    0
    97                   65
   123                    0
   181                  924
   182                    0
   ...
0x0149             [ 0x02BC 0x004E ]
0x014A                    0
0x014B                  330
   ...
```

This is the full Uppercase_Mapping property (as opposed to the Simple_Uppercase_Mapping given in the example for format "a"). The only difference between the two in the ranges shown is that the code point at 0x0149 (LATIN SMALL LETTER N PRECEDED BY APOSTROPHE) maps to a string of two characters, 0x02BC (MODIFIER LETTER APOSTROPHE) followed by 0x004E (LATIN CAPITAL LETTER N).

No adjustments are needed to entries that are references to arrays; each such entry will have exactly one element in its range, so the offset is always 0.

ae

This is like "a", but some elements are the empty string, and should not be adjusted. The one internal Perl property accessible by prop_invmap is of this type: "Perl_Decimal_Digit" returns an inversion map which gives the numeric values that are represented by the Unicode decimal digit characters. Characters that don't represent decimal digits map to the empty string, like so:

```
@digits    @values
0x0000       ""
0x0030        0
0x003A:      ""
0x0660:       0
0x066A:      ""
0x06F0:       0
0x06FA:      ""
0x07C0:       0
0x07CA:      ""
0x0966:       0
...
```

This means that the code points from 0 to 0x2F do not represent decimal digits; the code point 0x30 (DIGIT ZERO) represents 0; code point 0x31, (DIGIT ONE), represents 0+1-0 = 1; ... code point 0x39, (DIGIT NINE), represents 0+9-0 = 9; ... code points 0x3A through 0x65F do not represent decimal digits; 0x660 (ARABIC-INDIC DIGIT ZERO), represents 0; ... 0x07C1 (NKO DIGIT ONE), represents 0+1-0 = 1 ...

ale

is a combination of the "al" type and the "ae" type. Some of the map array elements have the forms given by "al", and the rest are the empty string. The property NFKC_Casefold has this form. An example slice is:

```
@$ranges_ref  @$maps_ref         Note
   ...
  0x00AA       97                FEMININE ORDINAL INDICATOR => 'a'
  0x00AB        0
  0x00AD                         SOFT HYPHEN => ""
  0x00AE        0
  0x00AF     [ 0x0020, 0x0304 ]  MACRON => SPACE . COMBINING MACRON
```

```
0x00B0        0
...
```

ar

means that all the elements of the map array are either rational numbers or the string `"NaN"`, meaning "Not a Number". A rational number is either an integer, or two integers separated by a solidus (`"/"`). The second integer represents the denominator of the division implied by the solidus, and is actually always positive, so it is guaranteed not to be 0 and to not be signed. When the element is a plain integer (without the solidus), it may need to be adjusted to get the correct value by adding the offset, just as other `"a"` properties. No adjustment is needed for fractions, as the range is guaranteed to have just a single element, and so the offset is always 0.

If you want to convert the returned map to entirely scalar numbers, you can use something like this:

```
my ($invlist_ref, $invmap_ref, $format) = prop_invmap($property);
if ($format && $format eq "ar") {
    map { $_ = eval $_ if $_ ne 'NaN' } @$map_ref;
}
```

Here's some entries from the output of the property "Nv", which has format `"ar"`.

```
@numerics_ranges  @numerics_maps       Note
       0x00           "NaN"
       0x30             0           DIGIT 0 .. DIGIT 9
       0x3A           "NaN"
       0xB2             2           SUPERSCRIPTs 2 and 3
       0xB4           "NaN"
       0xB9             1           SUPERSCRIPT 1
       0xBA           "NaN"
       0xBC            1/4          VULGAR FRACTION 1/4
       0xBD            1/2          VULGAR FRACTION 1/2
       0xBE            3/4          VULGAR FRACTION 3/4
       0xBF           "NaN"
       0x660            0           ARABIC-INDIC DIGIT ZERO .. NINE
       0x66A          "NaN"
```

n

means the Name property. All the elements of the map array are simple scalars, but some of them contain special strings that require more work to get the actual name.

Entries such as:

```
CJK UNIFIED IDEOGRAPH-<code point>
```

mean that the name for the code point is "CJK UNIFIED IDEOGRAPH-" with the code point (expressed in hexadecimal) appended to it, like "CJK UNIFIED IDEOGRAPH-3403" (similarly for `CJK COMPATIBILITY IDEOGRAPH-<code point>`).

Also, entries like

```
<hangul syllable>
```

means that the name is algorithmically calculated. This is easily done by the function *"charnames::viacode(code)" in charnames*.

Note that for control characters (`Gc=cc`), Unicode's data files have the string `"<control>"`, but the real name of each of these characters is the empty string. This function returns that real name, the empty string. (There are names for these characters, but they are considered

aliases, not the Name property name, and are contained in the `Name_Alias` property.)

ad

means the Decomposition_Mapping property. This property is like `"al"` properties, except that one of the scalar elements is of the form:

```
<hangul syllable>
```

This signifies that this entry should be replaced by the decompositions for all the code points whose decomposition is algorithmically calculated. (All of them are currently in one range and no others outisde the range are likely to ever be added to Unicode; the `"n"` format has this same entry.) These can be generated via the function *Unicode::Normalize::NFD()*.

Note that the mapping is the one that is specified in the Unicode data files, and to get the final decomposition, it may need to be applied recursively.

Note that a format begins with the letter "a" if and only the property it is for requires adjustments by adding the offsets in multi-element ranges. For all these properties, an entry should be adjusted only if the map is a scalar which is an integer. That is, it must match the regular expression:

```
/ ^ -? \d+ $ /xa
```

Further, the first element in a range never needs adjustment, as the adjustment would be just adding 0.

A binary search can be used to quickly find a code point in the inversion list, and hence its corresponding mapping.

The final element (index [3], assigned to `$default` in the "block" example) in the four element list returned by this function may be useful for applications that wish to convert the returned inversion map data structure into some other, such as a hash. It gives the mapping that most code points map to under the property. If you establish the convention that any code point not explicitly listed in your data structure maps to this value, you can potentially make your data structure much smaller. As you construct your data structure from the one returned by this function, simply ignore those ranges that map to this value, generally called the "default" value. For example, to convert to the data structure searchable by *charinrange()*, you can follow this recipe for properties that don't require adjustments:

```
my ($list_ref, $map_ref, $format, $missing) = prop_invmap($property);
my @range_list;

# Look at each element in the list, but the -2 is needed because we
# look at $i+1 in the loop, and the final element is guaranteed to map
# to $missing by prop_invmap(), so we would skip it anyway.
for my $i (0 .. @$list_ref - 2) {
   next if $map_ref->[$i] eq $missing;
   push @range_list, [ $list_ref->[$i],
                       $list_ref->[$i+1],
                       $map_ref->[$i]
                     ];
}

print charinrange(\@range_list, $code_point), "\n";
```

With this, `charinrange()` will return `undef` if its input code point maps to `$missing`. You can avoid this by omitting the `next` statement, and adding a line after the loop to handle the final element of the inversion map.

Similarly, this recipe can be used for properties that do require adjustments:

```
    for my $i (0 .. @$list_ref - 2) {
       next if $map_ref->[$i] eq $missing;

       # prop_invmap() guarantees that if the mapping is to an array, the
       # range has just one element, so no need to worry about adjustments.
       if (ref $map_ref->[$i]) {
           push @range_list,
                      [ $list_ref->[$i], $list_ref->[$i], $map_ref->[$i] ];
       }
       else {  # Otherwise each element is actually mapped to a separate
               # value, so the range has to be split into single code point
               # ranges.

           my $adjustment = 0;

           # For each code point that gets mapped to something...
           for my $j ($list_ref->[$i] .. $list_ref->[$i+1] -1 ) {

               # ... add a range consisting of just it mapping to the
               # original plus the adjustment, which is incremented for the
               # next time through the loop, as the offset increases by 1
               # for each element in the range
               push @range_list,
                                [ $j, $j, $map_ref->[$i] + $adjustment++ ];
           }
       }
    }
```

Note that the inversion maps returned for the `Case_Folding` and `Simple_Case_Folding` properties do not include the Turkic-locale mappings. Use *casefold()* for these.

`prop_invmap` does not know about any user-defined properties, and will return `undef` if called with one of those.

Unicode::UCD::UnicodeVersion

This returns the version of the Unicode Character Database, in other words, the version of the Unicode standard the database implements. The version is a string of numbers delimited by dots (`'.'`).

Blocks versus Scripts

The difference between a block and a script is that scripts are closer to the linguistic notion of a set of code points required to present languages, while block is more of an artifact of the Unicode code point numbering and separation into blocks of consecutive code points (so far the size of a block is some multiple of 16, like 128 or 256).

For example the Latin **script** is spread over several **blocks**, such as `Basic Latin`, `Latin 1 Supplement`, `Latin Extended-A`, and `Latin Extended-B`. On the other hand, the Latin script does not contain all the characters of the `Basic Latin` block (also known as ASCII): it includes only the letters, and not, for example, the digits or the punctuation.

For blocks see *http://www.unicode.org/Public/UNIDATA/Blocks.txt*

For scripts see UTR #24: *http://www.unicode.org/unicode/reports/tr24/*

Matching Scripts and Blocks

Scripts are matched with the regular-expression construct `\p{...}` (e.g. `\p{Tibetan}` matches characters of the Tibetan script), while `\p{Blk=...}` is used for blocks (e.g. `\p{Blk=Tibetan}` matches any of the 256 code points in the Tibetan block).

Old-style versus new-style block names

Unicode publishes the names of blocks in two different styles, though the two are equivalent under Unicode's loose matching rules.

The original style uses blanks and hyphens in the block names (except for `No_Block`), like so:

```
Miscellaneous Mathematical Symbols-B
```

The newer style replaces these with underscores, like this:

```
Miscellaneous_Mathematical_Symbols_B
```

This newer style is consistent with the values of other Unicode properties. To preserve backward compatibility, all the functions in Unicode::UCD that return block names (except one) return the old-style ones. That one function, *prop_value_aliases()* can be used to convert from old-style to new-style:

```
my $new_style = prop_values_aliases("block", $old_style);
```

Perl also has single-form extensions that refer to blocks, `In_Cyrillic`, meaning `Block=Cyrillic`. These have always been written in the new style.

To convert from new-style to old-style, follow this recipe:

```
$old_style = charblock((prop_invlist("block=$new_style"))[0]);
```

(which finds the range of code points in the block using `prop_invlist`, gets the lower end of the range (0th element) and then looks up the old name for its block using `charblock`).

Note that starting in Unicode 6.1, many of the block names have shorter synonyms. These are always given in the new style.

BUGS

Does not yet support EBCDIC platforms.

AUTHOR

Jarkko Hietaniemi. Now maintained by perl5 porters.