6

Юникод

Если вы никогда не слышали о Юникоде, вероятно, последние 20 лет вы прожили на необитаемом острове и пользовались механической печатной машинкой. Свое двадцатилетие Юникод отпраздновал в начале 2010 года. Даже если вы слышали об этом стандарте, вы можете не знать, что это такое на самом деле, и как с ним работать. Стыдиться тут совершенно нечего, ведь все до единого люди, включая и его изобретателей, все еще продолжают изучать Юникод. Мы определенно не сможем охватить все нюансы и тонкости Юникода в этой главе и даже в этой книге, но мы надеемся помочь вам начать пользоваться Юникодом в Perl.

Работа с Юникодом в наше время не является вопросом выбора: это насущная необходимость. Значительная часть текстовой информации в Сети хранится в Юникоде,[1] и многие крупные коллекции документов на 100% хранятся в Юникоде. Поскольку веб-браузеры прилагают все усилия, чтобы корректно отображать текстовую информацию, содержащую любые наборы символов, вы, вероятно, и не заметили, сколь часто на практике применяется Юникод. Языки программирования без качественной поддержки Юникода отстали на десятилетия, как и программы, написанные на этих языках. Возможно, они годились для восьмидесятых или даже для девяностых, но сегодня нам необходимы более мощные инструменты.

Так что же нас ждет здесь?

Компьютеры хранят символы в виде чисел. На заре компьютеров то были небольшие целые числа, имеющие длину 5, 6, 7 или 8 битов. Кодировка EBCDIC использовала 8 битов и основывалась на перфокартах. Кодировка ASCII использует только 7 битов, оставляя один бит в каждом байте для других целей – многих, многих других целей, порой противоречащих друг другу.

Так что в те дни практически каждый обитатель западных земель путал символы с маленькими числами в диапазоне от 0 до 127 или от 0 до 255. И хотя чисел получалось больше, чем клавиш на клавиатуре, их не хватало, и люди в разных странах имели собственные представления о том, какие символы должны представлять эти числа.

---

[1] В кодировке Юникода UTF-8, если выражаться точно.

Такого набора букв вполне хватало, чтобы отправить телеграмму на простом английском языке, но было недостаточно для представления всех символов латинского, греческого и кириллического алфавитов, не говоря уже об алфавитах, используемых в Азии. Жителям азиатских стран пришлось определять собственные, взаимно несовместимые 16-битные наборы символов. Было чрезвычайно трудно, а порой и невозможно включить текст на разных языках в один документ, поскольку разные символы из разных алфавитов могли быть представлены одними и теми же числовыми кодами.

Люди привыкли создавать наборы символов, отражающие их потребности в контексте собственной культуры. Во всякой культуре люди естественным образом ленивы, а потому обычно стремились включать только необходимые им символы и исключать ненужные. Такой исключающий подход работает, пока мы общаемся только с людьми, принадлежащими нашей собственной культуре. Но как только мы начали использовать Интернет для обмена информацией между разными культурами, такой подход стал источником проблем. Не так просто разобраться, как с помощью американской клавиатуры набирать латинские символы с акцентами, не говоря уже о более экзотических символах.

В результате появилась идея создать Юникод: единую систему символов, которую можно использовать взаимозаменяемо для набора практически любых текстов. Юникод охватывает не только современные алфавиты, но и множество древних, а также специальные символы, используемые в полиграфии, математике, лингвистике и многих других отраслях, включая даже смайлики, используемые в сотовых телефонах. См. табл. 6.1.

Таблица 6.1. Примеры символов Юникода

Латинские буквы	Å ǽ ç ð é ffi ï İ ñ ø œ š ſß ʒ þ
Греческие буквы	α ά̓ β Γ γ Δ δ ϝ θ λ Ξ ξ π Σ σ ς φ χ Ψ Ω ϕ̔
Кириллические буквы	а Б б д ж з И и к Л с т У у Ф ф Ш ь Э Ю я
Математические буквы	$\mathscr{A}$ $\mathscr{D}$ $\mathfrak{B}$ $\mathfrak{R}$ $\mathbb{E}$ $\mathbb{F}$ $\mathfrak{x}$ $\mathfrak{y}$ $\imath$ ξ ς $\mathbf{\Theta}$ $\mathbf{\Psi}$ $\mathbf{\Omega}$
Математические символы	÷ × ± ≤ ≠ ≈ ≡ ∅ ∈ $\sqrt[3]{}$ Δ Ω ⊆ ∞ ∫ ∂ ∴ $\aleph_0$
Символы валют	¢ $ £ £ € ₡ ₢ ₣ ƒ ₱ ₩ ¥
Точки	. · ‥ ○ ∘ • ¨ .. ... ⋯ ⋱ . ˙ ⋮
Дефисы и тире	- ‐ ‑ ‒ – - — ―
Кавычки	‘ “ ‘ ’ “ ” ‹ › « » ⌈ ⌋

Как видите, некоторые из них очень похожи между собой. Юникод различает символы не по внешнему виду, а по их назначению. В первую очередь Юникод – это семантический код, и только потом – код начертания, и то лишь потому, что начертание является частью семантики. В отличие от кодировки ASCII, охватывающей небольшой набор символов и определяющей малое количество свойств для этих символов, Юникод охватывает более обширные сферы. Юникод можно считать расширенной формой ASCII с огромным количеством символов. Но Юникод – это не только более обширный набор символов, это также свод правил распределения символов по категориям и работы с ними.

Помимо символов и их свойств, Юникод определяет также: порядок отображения регистров символов и приведения символов к единому регистру (свертки); комби-

нирующие символы; группы графем; формы нормализации; порядок сопоставления; свойства символов для использования при поиске по шаблону, включая имена символов, категории и алфавиты; числовые эквиваленты (например, позволяющие определить, что число, обозначаемое символом U+216B, «XII», имеет значение 12); ширину печати; двунаправленность; правила разрыва слов и строк; и вариации начертания. Список далеко не полон...

Первая предварительная поддержка Юникода появилась в Perl v5.6, однако важные задачи ввода/вывода нам удалось решить только к Perl v5.8. К версии v5.12 мы решили большинство более мелких проблем, и, начиная с версии v5.14, Perl обеспечивает полную и своевременную поддержку стандарта Юникода версии v6.0 – теперь вы можете использовать Юникод в Perl, не прибегая к каким-либо ухищрениям. Почти. Во всяком случае, в Perl этих ухищрений по-любому потребуется меньше, чем в любом другом языке.

Мы хотим сказать, что сохранили возможность решать простые задачи простым способом, при этом не сделав сложные неразрешимыми. Первая простая вещь, на которую следует обратить внимание, – Perl позволяет хранить в строках любые символы, с любыми значениями кодов. Кодировка ASCII ограничивает код символа 7 битами, Latin-1 – 8 битами, а Юникод поддерживает коды длиной до 21 бита.[1] Но символы в Perl не ограничены столь маленькими числами. В настоящее время на 64-битных архитектурах символы в Perl ограничены 64-битными значениями, но даже с учетом этого ограничения Perl позволяет представлять на 18 446 744 073 708 437 504 кодов символов больше, чем имеется в Юникоде. (Мы ведь сказали «с любыми значениями кодов»? Так-то.)

В Юникоде каждому символу соответствует уникальный номер, который называется *кодом символа*. Отсюда и происходит название Юникод: уникальный, универсальный код для каждого самостоятельного символа. Например, символ с именем LATIN CAPITAL LETTER A имеет порядковый десятичный номер 65, шестнадцатеричный 0x41. Он часто записывается как U+0041; существующие соглашения таковы, что префикс «U+» обозначает не просто число, а число, представляющее код символа Юникода.

Если вы когда-нибудь принимали по ошибке «`l`» за «1», «`O`» за «0» или «`,`» за «.», то уже знаете, как человек легко может спутать символы. Вы также знаете, что компьютер никогда не путается. Для него не имеет значения, как выглядит начертание символа в шрифте. Для него важно лишь назначение символа. Например, в табл. 6.2 перечислены символы, которые выглядят почти одинаково при использовании почти любых шрифтов.

В первой колонке табл. 6.2 приводится начертание символа. Вы сможете использовать литералы Юникода, подобные этим, в своем программном коде, только если в текущей лексической области видимости действует прагма `use utf8`. Большинство современных текстовых редакторов и оконных систем предоставляет различные способы ввода таких символов, однако они могут быть отключены по умолчанию, поэтому может потребоваться провести небольшие исследования, если вы не слишком ленивы. Но постойте, вам может пригодиться и такой способ: если вы не знаете, как ввести тот или иной символ, его можно попробовать

[1] На деле эта длина составляет лишь 20,087463 бита, поскольку Юникод содержит лишь 0x110000 кодов, а не 0x200000.

отыскать где-нибудь, выделить мышью, скопировать в буфер обмена и вставить в свой текст.

Таблица 6.2. Символы Юникода, которые легко перепутать с буквой A

Начертание	Код	Категория	Алфавит	Имя
A	U+00041	Lu	Latin	LATIN CAPITAL LETTER A
A	U+0FF21	Lu	Latin	FULLWIDTH LATIN CAPITAL LETTER A
A	U+00391	Lu	Greek	GREEK CAPITAL LETTER ALPHA
A	U+00410	Lu	Cyrillic	CYRILLIC CAPITAL LETTER A
A	U+1D5A0	Lu	Common	MATHEMATICAL SANS-SERIF CAPITAL A
A	U+1D670	Lu	Common	MATHEMATICAL MONOSPACE CAPITAL A

Если отвлечься от внешнего вида, начертание, в некотором смысле, является наименее полезным аспектом символа, потому что невозможно знать, как тот или иной символ отображается (и отображается ли вообще) с применением какого-либо другого шрифта, кроме того, что используете вы сами. У вас начертание может выглядеть очень неплохо, но не верьте своим глазам – верьте числам. Во второй колонке указаны числовые коды символов, в стандартном формате Юникода. А вот несколько способов работать с кодами символов Юникода в языке Perl:

```
if (ord($somechar) == 0x0391) { .... }
$alpha = "\x{391}";
$alpha = "\N{U+391}";
$alpha = chr(0x391);
```

Двумя наиболее важными свойствами символов, помимо их числовых кодов, являются: принадлежность к категории (колонка «Категория» в табл. 6.2) и принадлежность к алфавиту (колонка «Алфавит» в табл. 6.2). В шаблонах свойства кодов символов Юникода чаще всего используются в роли именованных классов символов, где \p{*PROPERTY*} соответствует любому коду символа с данным свойством, а \P{*PROPERTY*} соответствует любому коду символа, не обладающему данным свойством.

```
/^\p{GC=Lu}+$/                        # Все заглавные буквы
/^\p{script=Greek}+$/                 # символы греческого алфавита
/[\P{script=Latin}\P{script=Common}]/ # не пересекаются
```

Последний шаблон соответствует строкам, содержащим любые коды символов, не принадлежащие алфавитам Latin и Common. Поскольку регистр символов, наличие пробелов и символов подчеркивания в именах свойств не имеет большого значения, вы можете форматировать их как вам заблагорассудится. То есть, если вам покажется, что \p{gc=modifier_letter} читается лучше, когда используются только символы нижнего регистра, а \P{SC=INHERITED} – когда используются только символы верхнего регистра, пишите как нравится: Perl об этом не беспокоится. Или поступите наоборот, если вам того захочется.

Помимо показанных выше свойств, состоящих из двух частей, для обозначения категорий и алфавитов Perl предлагает также односоставные псевдонимы, что позволяет написать просто \p{Lu} и \p{Greek}. Например, если вам потребуется убедиться, что строка содержит только символы из алфавитов Latin и Greek, достаточно выполнить следующую проверку:

```
$mylang = qr/[\p{Latin}\p{Greek}\p{Common}\p{Inherited}]/;
if ($string =~ /\A$mylang+\z/) { ... }
```

Мы добавили псевдоним Common для обозначения кодов символов, общих для различных алфавитов, таких как цифры и знаки пунктуации, а псевдоним Inherited – для обозначения комбинационных кодов символов (как правило, диакритических знаков), которые присоединяются к алфавиту символа, с которым используются. Комбинационные коды символов не имеют аналогов в ASCII, и, случается, сбивают с толку тех говорящих на ASCII, кто впервые знакомится с Юникодом. Пожалуй, наиболее близким понятием в ASCII является перечеркивание с использованием символа забоя, за исключением того, что в Юникоде комбинационный код автоматически применяется к предшествующему базовому коду. Подробнее о них рассказывается ниже, в разделе «Графемы и нормализация».

Вы могли заметить, что в этой главе мы проводим различие между «символом» и «кодом» или «кодом символа». В других местах (включая другие главы этой книги) данные термины часто используются взаимозаменяемо, и термин «символ» также часто используется вместо термина «графема», но в данной главе нам требуется чуть большая точность. Коды – это целые числа, составляющие строку, когда она считается списком целых чисел, тогда как «символ» – достаточно расплывчатый термин, который в человеческом понимании может означать и код (или кодовый пункт), и графему. В общем случае следует помнить, что слово «символ» в разговоре может иметь до трех-четырех различных смыслов.

Последняя колонка в табл. 6.2 содержит имена символов. Ой, т.е. названия кодов символов. Чтобы получить возможность называть коды по именам в тексте программы, имена сначала нужно загрузить при помощи модуля `charnames`, а потом записывать в форме `\N{...}`, как показано ниже:

```
use charnames qw(:full);

$alpha      =     "\N{GREEK CAPITAL LETTER ALPHA}";
$alpha_code = ord "\N{GREEK CAPITAL LETTER ALPHA}";
if ($string =~    /\N{GREEK CAPITAL LETTER ALPHA}/) { ... }
```

Пользоваться названиями кодов символов намного правильнее, чем их числовыми значениями, поскольку имена делают текст программы более понятным.

Другие интересные приемы, основанные на использовании формы записи `\N{...}`, приводятся в разделе «charnames» главы 29.

Не рассказывай, а показывай

Если картина стоит тысячи слов, то возможность вставлять в программу те или иные символы определенно стоит не меньше пятидесяти. Поэтому для начала необходимо сообщить Perl, что текст вашей программы содержит символы Юникода, а не простые байты.[1] Вас никто не обязывает делать это, но данный шаг немного облегчит жизнь, если вам потребуется вставить в программный код символы Юникода.

---

[1] Возможно, вы предпочитаете называть их «октетами»; пусть так, но мы считаем эти два слова близкими синонимами, поэтому мы будем придерживаться терминологии, более близкой технарям.

Perl по сей день предполагает, что все модули с исходными текстами содержат только символы ASCII, если явно не указано иное (хотя мы осознаем, что рано или поздно кодировкой по умолчанию станет Юникод). Вы всегда можете определять кодовые пункты Юникода с помощью приемов, упомянутых выше, но литералы всегда будут интерпретироваться как отдельные байты. Встретив литерал символа в кодировке UTF-8, Perl не посчитает его одним логическим символом, а будет рассматривать его как один, два, три или даже четыре отдельных символа, с порядковыми номерами, не превышающими 256. Чтобы этого не случилось, используйте следующие объявления:

```
use v5.14;      # включает механизм unicode_strings
use utf8;       # обрабатывает литералы в кодировке UTF-8
```

Первое гарантирует, что кодовые пункты с порядковыми номерами в непростом диапазоне 128–255 будут интерпретироваться как строки Юникода, а второе сообщает компилятору Perl, что файл в целом содержит текст Юникода в кодировке UTF-8. Прагма `utf8` позволяет использовать Юникод в литералах строк и регулярных выражений.

```
my $dwarf   = "Þórinn Eikinskjaldi";
my $search  = "búsqueda";
my $measure = "Ångström";
my $how     = "à contre-cœur";
my $motto   = "👪💗🐪";
```

Такой текст намного проще читается, хотя набрать его может оказаться не так просто, как вот этот эквивалент:

```
use charnames qw(:full);

my $dwarf   = "\N{LATIN CAPITAL LETTER THORN}\N{LATIN SMALL LETTER
              O WITH ACUTE}rinn Eikinskjaldi";
my $search  = "b\N{LATIN SMALL LETTER U WITH ACUTE}squeda";
my $measure = "A\N{COMBINING RING ABOVE}ngstro\N{COMBINING DIAERESIS}m";
my $how     = "\N{LATIN SMALL LETTER A WITH GRAVE} contre-c\N{LATIN
              SMALL LIGATURE OE}ur";
my $motto   = "\N{FAMILY}\N{GROWING HEART}\N{DROMEDARY CAMEL}";
```

При этом оба способа, представленные выше, предпочтительнее использования секретных кодов в тексте программы:

```
my $dwarf   = "\x{DE}\x{F3}rinn Eikinskjaldi";
my $search  = "b\x{FA}squeda";
my $measure = "\x{C5}ngstr\x{F6}m";
my $how     = "\x{E0} contre-c\x{153}ur";
my $motto   = "\x{1F46A}\x{1F497}\x{1F42A}";
```

Но и это еще не все. В области действия прагмы `utf8` появляется возможность использовать Юникод в идентификаторах Perl.

```
# несколько наборов символов
my @IsO   = qw( Latin1 Latin2 Latin15 );
my @µsoft = qw( cp852 cp1251 cp1252 );
my @鯉    = qw( koi8-f koi8-u koi8-r );
```

```
# включать ли ответы, которые не возвращают результатов
my $INCLUÍR_NINGUNOS = 0;

# имеют ли значение диакритические знаки
my $SI_IMPORTAN_MARCAS_DIACRÍTICAS = 0;

# считайте << за оператор "имеет" :)
my @ciudades_españolas = ordenar_a_la_española(<<'LA_ÚLTIMA' =~ /\S.*\S/g);
.....
.....
LA_ÚLTIMA

my $déjà_imprimée;     # название города

# Идентификатор на греческом языке
my @ὑπέρμεγας = ( );

# А теперь мы просто выпендриваемся :-)
my $ɬndɬno = uморəрɪsdn($input);
```

На практике, если вы будете использовать идентификаторы не на английском языке, вам, вероятно, захочется писать комментарии на соответствующем языке. Perl упрощает возможность писать программы на собственном языке для людей во всем мире, не вынуждая их изучать английский язык.

В настоящее время вы должны ограничить применение Юникода только приватными переменными. Это обусловлено особенностями хранения глобальных переменных, а также особенностями отображения имен модулей в имена файлов в локальной файловой системе. Первое ограничение, как ожидается, будет снято в ближайшем будущем, а вот второе пока остается предметом для исследования.

Доступ к данным в Юникоде

Для целей внутреннего хранения любых кодов символов Perl применяет формат, совместимый с Юникодом. То есть 21 младший бит в точности соответствует диапазону кодов Юникода, так же как 8 младших битов Юникода соответствуют кодировке Latin-1. Как фактически хранятся коды символов, не так важно для рядового пользователя Perl.

При этом, как только возникает необходимость взаимодействовать с внешним миром, приходится предусматривать соответствующую интерпретацию получаемых данных и генерировать выходные данные в формате, приемлемом для принимающей их программы. Внутри Perl символы *декодируются* из внешнего представления в абстрактные символы, но когда требуется вывести эти символы, их необходимо *закодировать* в некоторое ожидаемое представление. Если забыть сделать это, программа сгенерирует то, что иногда называют «широкими символами» (wide characters) или «неправильно сформированными символами UTF-8» (malformed UTF-8 character).

В Perl имеется два основных способа определить кодировку для всего потока данных, плюс различные сокращения, упрощающие жизнь. Если поток ввода/вывода уже открыт, его кодировку можно установить, передав ее в качестве второго аргумента функции `binmode`:

```
binmode(STDIN,  ":encoding(CP1252)")
    || die "can't binmode to cp1252: $!";
binmode(STDOUT, ":encoding(UTF-8)")
    || die "can't binmode to UTF-8: $!";
```

Если поток еще не был открыт, кодировку можно назначить вместе с режимом при вызове функции `open`.

```
open(OUTPUT, "> :raw :encoding(UTF-16LE) :crlf", $filename)
             or die "can't open $filename: $!";
print OUTPUT for @stuff;
close(OUTPUT) or die "couldn't close $filename: $!";
```

При выводе данных фильтр `:crlf` выполняет преобразование символов `\n` в пары `\r\n`; при вводе – наоборот. Этот режим действует по умолчанию в Windows, при работе с текстовыми файлами, но, если это необходимо, его следует явным образом включать в UNIX. См. страницу *PerlIO* справочного руководства, где приводится дополнительная информация о фильтрах ввода/вывода.

Помимо `\n` и `\r\n` в Юникоде существуют и другие символы завершения строк. В настоящее время таких последовательностей в Юникоде существует восемь–семь перечисленных ниже символов, плюс последовательность `\r\n`, состоящая из двух кодов:

```
U+000A LINE FEED (LF)
U+000B LINE TABULATION
U+000C FORM FEED (FF)
U+000D CARRIAGE RETURN (CR)
U+0085 NEXT LINE (NEL)
U+2028 LINE SEPARATOR
U+2029 PARAGRAPH SEPARATOR
```

В Perl нет специального фильтра обобщенной обработки последовательностей завершения строки Юникода, но, если вы можете позволить себе прочитать файл целиком в память, то легко сможете преобразовать все такие последовательности в символы перевода строки:

```
$complete_file =~ s/\R/\n/g;
```

Или разбить содержимое файла на список строк, не имеющих последовательностей завершения строки:

```
@lines = split(/\R/, $complete_file);
```

Чтобы назначить кодировку для вновь открываемых файлов, можно воспользоваться прагмой `open`. Например, ниже показано, как организовать использование кодировки UTF-8 для всех файлов, которые будут играть роль стандартных потоков ввода/вывода – `STDIN`, `STDOUT` и `STDERR` – если при открытии кодировка не была указана явно:

```
use open qw( :encoding(UTF-8) :std );
```

Если вам достаточно назначить кодировку UTF-8 для стандартных потоков, можно при необходимости пользоваться ключом командной строки `-CS` или устанавливать переменную среды `PERL_UNICODE` в значение `"S"`. Если вместо `"S"` использовать значение `"D"`, все дескрипторы будут открываться в текстовом режиме и с кодировкой UTF-8 по умолчанию. См. главу 17.

Использование ключа командной строки `-C` или переменной среды `PERL_UNICODE` требует явного вызова функции `binmode` для двоичных потоков даже в программах для UNIX, что обычно требуется только в Windows или в переносимых программах. Это может нарушить работу уже имеющихся программ для UNIX, предполагающих, по умолчанию, что они работают с двоичными данными, а не с текстом. Но это не нарушит работу существующих программ, которые не умеют декодировать текст в кодировке UTF-8.

Ниже перечислены механизмы назначения кодировки для потока ввода/вывода, следующие в порядке убывания приоритета (механизм, расположенный выше, может переопределить настройки, выполненные с помощью механизма, расположенного ниже в списке):

1. Явный вызов `binmode` для уже открытого дескриптора.
2. Включение фильтра во второй аргумент функции `open`.
3. Прагма `open`.
4. Ключ командной строки `-C`.
5. Переменная среды `PERL_UNICODE`.

Единственным исключением является дескриптор `DATA`. На него не действуют прагмы `use utf8` и `open`, поэтому при необходимости вам придется указывать кодировку вручную:

```
binmode(DATA, ":encoding(UTF-8)");
```

Из-за особенностей реализации уровней кодирования `utf8` и `UTF-8`, обычно они не возбуждают исключения, встречая неправильно сформированные входные данные. Чтобы исправить ситуацию, добавьте в свой код следующую строку:

```
use warnings FATAL => "utf8";
```

В Perl (начиная с версии v5.14) существует три подгруппы предупреждений, составляющие группу предупреждений `"utf8"`. Эти типы предупреждений бывает полезно различать:

nonchar

66 кодов символов Юникода считаются «несимвольными». Все они относятся к общей категории Unassigned (Cn) и никогда не будут сопоставлены каким-либо символам. Они не могут использоваться в открытом обмене, поэтому программный код может добавлять их в символьные данные в качестве различных сигнальных меток, и такие метки всегда можно отличить от основных данных. 32 несимвольных кода занимают диапазон от U+FDD0 до U+FDEF, а еще 34 располагаются парами в конце каждого сегмента (их шестнадцатеричные коды заканчиваются, соответственно, последовательностями FFFE или FFFF). В некоторых ситуациях может потребоваться разрешить присутствие таких несимвольных кодов, например, для организации взаимодействия между процессами, совместно использующими одни и те же сигнальные маркеры. В таких случаях используйте:

```
no warnings "nonchar";
```

surrogate

Эти кодовые пункты зарезервированы для использования в кодировке UTF-16. На практике нет никаких причин разрешать их использование, и никакие

совместимые процессы не смогут обмениваться ими, потому что программы, использующие кодировку UTF-16, не способны обрабатывать их (хотя, программы, использующие кодировки UTF-8 и UTF-32, могли бы использовать их при желании).

non_unicode

Максимально допустимый код символа в Юникоде имеет значение U+10FFFF, но Perl способен символы с любыми кодами, вплоть до максимального беззнакового целого, поддерживаемого аппаратной архитектурой. В зависимости от различных настроек и фазы Луны, Perl может предупредить о попытке ввода или вывода кодов, выходящих за верхнюю границу (используя категорию предупреждений «non_unicode», которая является подкатегорией «utf8»). Например, `"uc(0x11_0000)"` вызовет такое предупреждение, вернув входной параметр в качестве результата, так как кодовым пунктом верхнего регистра любого кода символа, не принадлежащего Юникоду, является этот же код.

Из перечисленных подгрупп самой полезной является содержащая коды, выходящие за границы Юникода, и вы наверняка пожелаете разрешить использование таких символов для внутренних нужд.

```
no warnings "non_unicode";
```

Вот одно из возможных применений. То, что следующая операция делает для ASCII:

```
tr[\x00-\x7F][\x80-\xFF];
```

операция ниже делает для Юникода:

```
tr[\x{00_0000}-\x{10_FFFF}][\x{20_0000}-\x{30_FFFF}];
```

Она отображает все коды из допустимого диапазона в соответствующие им символы из недопустимого диапазона. Зачем это может понадобиться? Это один из способов пометить текст, который не должен затрагиваться поиском. Только не забудьте выполнить обратное преобразование, завершив поиск.

Модуль Encode

Стандартный модуль `Encode` используется чаще неявно, чем явно. Он загружается автоматически всякий раз, когда функции `binmode` или `open` передается аргумент `:encoding(`*`ENC`*`)`.

Но иногда приходится иметь дело с фрагментами кодированных данных, источником которых не служит поток ввода с назначенной кодировкой, поэтому возникает необходимость декодировать их вручную, прежде чем приступать к их обработке. Такие кодированные строки могут поступать в программу из разных источников за ее пределами, – скажем, из переменных среды, аргументов командной строки, параметров CGI или полей базы данных. Увы, иногда вам даже придется сталкиваться с «текстовыми» файлами, содержащими строки сразу в нескольких кодировках. Вы обязательно столкнетесь с *кракозябрами*.

Во всех этих ситуациях вам придется обратиться к модулю `Encode`, чтобы реализовать явное кодирование и декодирование данных. Чаще всего вам придется использовать функции (сюрприз!) `encode` и `decode` из этого модуля. Если у вас имеются необработанные внешние данные, хранящиеся в некоторой кодированной

форме в виде байтов, передайте их функции decode и превратить в абстрактные символы. С другой стороны, если у имеются некоторые абстрактные символы, и необходимо преобразовать их в некоторую кодированную форму, воспользуйтесь функцией encode.

```
use Encode qw(encode decode);
$chars = decode("shiftjis", $bytes);
$bytes = encode("MIME-Header-ISO_2022_JP", $chars);
```

Например, если известно, что терминал настроен на работу с кодировкой UTF-8, декодировать содержимое @ARGV можно следующим образом:

```
# действует так же, как perl -CA
if (grep /\P{ASCII}/ => @ARGV) {
    @ARGV = map { decode("UTF-8", $_) } @ARGV;
}
```

Если ваша рабочая среда не предлагает повсеместную поддержку кодировки UTF-8, не следует исходить из предположения, что терминал всегда настроен на использование кодировки UTF-8. Для него может быть установлена национальная кодировка. Стандартный модуль Encode не поддерживает операции, чувствительные к региональным настройкам, однако в архиве CPAN имеется модуль Encode::Locale, поддерживающий такие операции. Используйте его следующим образом:

```
use Encode;
use Encode::Locale;

# использовать "locale" как аргумент encode/decode
@ARGV = map { decode(locale => $_) } @ARGV;

# или как поток для binmode или open
binmode $some_fh, ":encoding(locale)";

binmode STDIN,  ":encoding(console_in)"  if -t STDIN;
binmode STDOUT, ":encoding(console_out)" if -t STDOUT;
binmode STDERR, ":encoding(console_out)" if -t STDERR;
```

Базы данных – еще одна область, где может пригодиться возможность выполнять кодирование и декодирование вручную. Конкретика, впрочем, зависит от используемой системы управления базами данных. Если речь о простых DBM-файлах, нижестоящая библиотека работает с байтами, а не строками кодов символов, поэтому текст Юникода нельзя записать непосредственно в файл DBM. Если попытаться сделать это, программа возбудит исключение Wide character in subroutine. Чтобы сохранить пары ключ/данные в DBM-хеше %dbhash, их нужно сначала преобразовать в кодировку UTF-8:

```
use Encode qw(encode decode);

# предполагается, что $uni_key и $uni_value - строки
# абстрактных символов Юникода

$enc_key   = encode("UTF-8", $uni_key);
$enc_value = encode("UTF-8", $uni_value);
$dbhash{$enc_key} = $enc_value;
```

Отсюда следует, что обратная операция извлечения значения в Юникоде требует сначала закодировать ключ перед его использованием, а затем декодировать извлеченное значение:

```
use DB_File;
use Encode qw(encode decode);

tie %dbhash, "DB_File", "pathname";

# $uni_key хранит обычную строку Perl (абстрактные символы Юникода)
$enc_key   = encode("UTF-8", $uni_key);

$enc_value = $dbhash{$enc_key};
$uni_value = decode("UTF-8", $enc_value);
```

После этого полученную строку `$uni_value` можно использовать как любую другую строку Perl. Хотя перед этим она была лишь последовательностью байтов – целых чисел, хранящихся в форме строки. (И эти целые числа ничем *не напоминали* коды символов Юникода.)

Начиная с версии v5.8.4 появилась возможность использовать стандартный модуль `DBM_Filter` для прозрачного кодирования/декодирования данных.

```
use DB_File;
use DBM_Filter;

use Encode qw(encode decode);

$dbobj = tie %dbhash, "DB_File", "pathname";
$dbobj->Filter_Value("utf8");

# $uni_key содержит обычную строку Perl (абстрактные символы Юникода)
$uni_value = $dbhash{$uni_key};
```

Ошибочные представления о регистре

Если вы знакомы только с набором символов ASCII, практически все ваши предположения относительно регистра символов Юникода будут ошибочны. В ASCII существуют буквы верхнего регистра и буквы нижнего регистра, но в Юникоде существует еще третий регистр – заглавный. Этот регистр не имеет широкого использования в английском языке, но применяется в других системах письменности, основанных на латинском или греческом алфавите.

Обычно заглавный регистр совпадает с верхним регистром, но не всегда. Он используется, когда только первая буква должна быть большой. Некоторые символы Юникода выглядят как две буквы, напечатанные рядом друг с другом, но в действительности представлены одним кодом символа. В слове, где только первая часть может быть большой, но не остальные, заглавный регистр позволит выделить размером лишь установленную часть. Эта возможность существует в основном для поддержки устаревших кодировок, и в настоящее время чаще применяются коды символов, для которых заглавный регистр отображается в два различных кода, один в верхнем регистре и один в нижнем. Ниже демонстрируется один из таких устаревших символов:

```
use v5.14;
use charnames qw(:full);
my $beast = "\N{LATIN SMALL LETTER DZ}ur";
say for $beast, ucfirst($beast), uc($beast);
```

Он выведет три слова: «dzur», «Dzur» и «DZUR», каждое из которых состоит из трех кодов.

Некоторые буквы не имеют регистра, а некоторые небуквы имеют. Регистр букв – относительно редкое явление в системах письменности. Лишь восемь алфавитов из почти 100, поддерживаемых Юникодом версии v6.0, имеют регистровые символы: Armenian, Coptic, Cyrillic, Deseret, Georgian, Glagolitic, Greek и Latin, плюс некоторые из алфавитов Common и Inherited.

При изменении регистра символов может измениться и длина строки. При *простом изменении регистра* (*simple casemapping*) измененная строка всегда имеет ту же длину, что исходная, но при *полном изменении регистра* это совершенно не обязательно. Например, строка «tschüß» после преобразования в верхний регистр «TSCHÜSS» становится на один символ длиннее.

Разные строки в одном регистре могут отображаться в одну и ту же строку в другом регистре. Обе сигмы греческого алфавита, «σ» и «ς», отображаются в одну и ту же букву верхнего регистра, «Σ», и это лишь самый простой пример. Чтобы достаточно разумно (или, хотя бы, не так безумно) решить проблему с подобными превращениями, было введено понятие свертки регистра, применяемой для сравнения символов без учета их регистра. Если после свертки регистров символов в двух строках получаются одинаковые строки, они *по определению* считаются эквивалентными без учета регистра символов.

Сопоставление без учета регистра символов в Perl всегда поддерживалось модификатором шаблонов `/i`, который сравнивает результаты *свертки регистров*. Начиная с версии v5.16, поддерживается функция `fc`, позволяющая сравнивать результаты свертки регистров двух строк, чтобы определить, являются ли они эквивалентными без учета регистра символов. До версии v5.16 функция `fc` была доступна в модуле `Unicode::CaseFold` из архива CPAN.

Загляните в свою копию страницы *perlfunc* справочного руководства и примечания к выпуску версии (*perldelta*), чтобы определить, поддерживает ли ваша версия Perl эту возможность. Если такая поддержка имеется, значит, наряду с ней поддерживается интерполируемая экранированная последовательность `\F`, действующая подобно `\L` и `\U`, но производящая свертку регистра.

Еще одна интересная особенность Юникода, несвойственная ASCII, заключается в том, что операция преобразования регистра может оказаться необратимой. Например, `lc("ß")` возвращает "ß", но `uc("ß")` возвращает "SS", а `lc("SS")` – "ss", что далеко от первоначальной формы. Но не обязательно прибегать к экзотическим двухсимвольным комбинациям, чтобы показать, что обратимость операции преобразования регистра не гарантируется. Вспомните греческие сигмы: «σ» – это обычная форма, а «ς» – форма, используемая в конце слов, и для обеих форм имеется единственная форма в верхнем регистре, «Σ». Если выполнить преобразование `lc(uc("ς"))`, результат будет отличаться от первоначального значения «ς» – вы получите символ «σ».

Однако не все символы, имеющие регистр, способны изменять его. В Юникоде символ *не обязан иметь* верхний или заглавный регистр лишь потому, что он

имеет нижний регистр. Например, `uc("M°Kinley")` вернет `"M°KINLEY"`, потому что символ MODIFIER LETTER SMALL C имеет нижний регистр, но не имеет других регистров, иначе он выглядел бы неправильно. Аналогично капитель, фактически, представлена символами нижнего регистра, потому что по высоте они соответствуют строчным буквам. В строке «Boulder Camera» первая буква каждого слова находится в верхнем регистре, а остальные – в нижнем.

Не все символы, имеющие нижний регистр, являются буквами. Регистр – это свойство, не зависящее от принадлежности к той или иной категории. Римские цифры, например, имеют регистр, сравните «Ⅷ» и «viii». Существуют даже буквы, считающиеся буквами нижнего регистра, для которых выполняется равенство `GC=Lm` и не выполняется `GC=Ll`.

В случае ASCII, чтобы перевести слово в регистр «заголовка», используется функция `ucfirst(lc($s))`, которая, однако, не гарантирует корректную работу с Юникодом, потому что перевод в заглавный регистр из нижнего регистра не всегда дает тот же результат, что и перевод в заглавный регистр оригинала. Это замечание верно и для других комбинаций. Правильный способ состоит в том, чтобы первую букву отдельно перевести в заглавный регистр, а остальные – в нижний, либо явно вызвав соответствующие функции, либо с помощью операции подстановки:

```
$tc = ucfirst(substr($s, 0, 1)) . lc(substr($s, 1));

s/(\w)(\w*)/\u$1\L$2/g;
```

Помимо основных категорий (General Categories) в Юникоде имеется довольно много категорий, связанных с регистром символов. В табл. 6.3 перечислены категории, поддерживаемые Юникодом версии v6.0. Все они являются логическими свойствами, поэтому допускается использовать форму записи с одним элементом. Иначе говоря, для проверки наличия свойства в любом кодовом пункте вместо `\p{CWCM=Yes}` и `\p{CWCM=No}` можно использовать форму записи `\p{CWCM}` и `\P{CWCM}`, соответственно.

Таблица 6.3. Свойства, связанные с регистром

Короткое	Длинное
Cased	Cased
Lower	Lowercase
Title	Titlecase
Upper	Uppercase
CWL	Changes_When_Lowercased
CWT	Changes_When_Titlecased
CWU	Changes_When_Uppercased
CWCM	Changes_When_Casemapped
CWCF	Changes_When_Casefolded
CWKCF	Changes_When_NFKC_Casefolded
CI	Case_Ignorable
SD	Soft_Dotted

Свойства `Lower` и `Upper` соответствуют всем кодам символов, обладающих соответствующими свойствами, не только буквам. В настоящее время не существует небуквенных символов, имеющие заглавный регистр, поэтому `Title` (пока) суть то же самое, что и `gc=Lt`. Однако в области действия модификатора `/i` все три свойства соответствуют свойству `Cased`, которым обладают не только буквы. При поиске без учета регистра символов эквивалентом `gc=Lt` является свойство `Case_Letter`.

Графемы и нормализация

Выше уже упоминались символы, такие как LATIN SMALL LETTER DZ, которые представлены одним кодом, но выглядят как два символа. Однако гораздо чаще встречается противоположная ситуация. То есть, для отображения одиночного символа (*графемы*) может потребоваться более одного кода. Представьте букву с диакритическим знаком (или парой знаков), скажем, «é» в слове «résumé». Каждая такая буква может обозначаться одним кодом или двумя. Может даже так случиться, что одна буква «é» в слове будет обозначена единственным кодом буквы, а другая – кодом буквы, за которым следует код диакритического знака. Глядя только на изображения символов, невозможно заметить разницу, потому что они считаются эквивалентными. Эта тонкость имеет серьезные последствия для обработке практически любого текста, и она почти противоречит тому, что говорилось выше о неважности начертаний. В данном конкретном смысле начертания приобретают особую важность.

Комбинационные символы используются, например, для превращения «n» в «ñ», «c» – в «ç», «o» – в «ô» или «u» – в «ű». В первых трех случаях требуется по одному комбинационному знаку, а в последнем – два. На практике количество комбинационных знаков не ограничивается. Вы можете добавить любое их количество, и в результате создать символы, прежде никем не виданные.

Все это требует серьезного переосмысления и переделки самого разнообразного программного обеспечения. Только представьте, какие функции возлагаются на систему управления шрифтами. (Нет, отказ не принимается.) Возможно, ваши собственные программы обработки текста нуждаются в серьезной переделке. Даже такие простые понятия, как перестановка символов строки в обратном порядке, становится большой проблемой, ведь если переупорядочивать коды, а не графемы, комбинационные знаки окажутся присоединены к другим базовым символам. Niño, María и François превратятся в этом случае в õniN, áiraM и siǫcnarF.

Рассмотрим графему, сконструированную на базе алфавитного кода, A, за которым следуют два комбинационных знака, назовем их X и Y. Имеет ли значение порядок, в котором они применяются? Являются ли графемы AXY и AYX одним и тем же символом? Иногда да, иногда нет. В таких графемах, как «ō̱», порядок не имеет значения, поскольку один знак располагается над символом, а второй – под ним. Так как порядок не имеет значения, программа должна считать эту графему за символ LATIN LETTER SMALL O, за которым следуют, в любом порядке, знаки COMBINING OGONEK и COMBINING MACRON. Но иногда оба комбинационных знака располагаются в одной и той же области символа, и тогда порядок *имеет значение*. Подробнее об этом чуть ниже.

Но и этим наши мучения не заканчиваются. Юникод содержит предварительно скомпонованные символы, назначение которых – обеспечивать двунаправленную

совместимость с устаревшими наборами символов. Например, алфавит Latin насчитывает порядка 500 таких символов,алфавит Greek – около 250. Подобные символы во множестве существуют и в других алфавитах.

Например, символу «é» соответствует код U+00E9, LATIN LETTER SMALL E WITH ACUTE. Да, всего один. А сложность вот в чем: обращаться с ним следует так же, как если бы графеме соответствовал код LATIN LETTER SMALL E, за которым следует COMBINING ACUTE ACCENT.

Для графем, включающих более одного комбинационного знака, количество вариантов представления еще больше, так как некоторые из них могут начинаться с того или иного предварительно скомпонованного символа, уже имеющего встроенный комбинационный знак, за которым следует еще один комбинационный знак.

Чтобы управиться со всем этим многообразием, Юникод вводит конкретную процедуру *нормализации*. Согласно глоссарию Юникода по адресу `http://unicode.org/glossary/`, нормализация – это «устранение из текстовых данных альтернативных представлений эквивалентных последовательностей и преобразование их в формы, эквивалентность которых может быть установлена на двоичном уровне». Иными словами, нормализация позволяет обеспечить уникальную идентификацию для каждой семантической единицы, благодаря чему устраняется связь «один-ко-многим».

Ниже перечислены четыре формы нормализации в Юникоде:

- форма нормализации D (Normalization Form D, NFD), получаемая в результате канонической декомпозиции;
- форма нормализации C (Normalization Form C, NFC), получаемая в результате канонической декомпозиции, за которой следует каноническая композиция;
- форма нормализации KD (Normalization Form KD, NFKD), получаемая в результате совместимой декомпозиции;
- форма нормализации KC (Normalization Form KC, NFKC), получаемая в результате совместимой декомпозиции, за которой следует каноническая композиция.

Обычно предпочтение отдается каноническим формам, потому что при нормализации в совместимые формы может теряться часть информации. Например, `NFKD("™")` вернет обычную двухсимвольную строку `"TM"`. Это может быть желательно для организации поиска и подобных операций, но каноническая декомпозиция для большинства ситуаций обычно дает более приемлемые результаты, чем совместимая декомпозиция.

Если не выполнить нормализацию самостоятельно, строка в программе необязательно будет иметь форму NFD или NFC; строки могут существовать в ненормализованном виде. Рассмотрим в качестве примера символ «o͂», который является всего лишь строчной латинской буквой «o» с тильдой и знаком долготы над ней (в противоположность знаку долготы с тильдой над ним). Данная графема может быть представлена различным числом кодов, от одного до трех, в зависимости от формы: `"\x{22D}"` – в NFC, `"\x{6F}\x{303}\x{304}"` – в NFD, и `"\x{F5}\x{304}"` – без нормализации. В табл. 6.4 перечислено семь вариантов строчной латинской буквы «o» с тильдой и в некоторых случаях со знаком долготы.

Таблица 6.4. Каноническая головоломка

№ п/п	Начертание	NFC?	NFD?	Литерал	Кодовые пункты
1	õ	✓	–	"\x{F5}"	LATIN SMALL LETTER O WITH TILDE
2	õ	–	✓	"o\x{303}"	LATIN SMALL LETTER O, COMBINING TILDE
3	o͞	✓	–	"\x{22D}"	LATIN SMALL LETTER O WITH TILDE AND MACRON
4	o͞	–	–	"\x{F5}\x{304}"	LATIN SMALL LETTER O WITH TILDE, COMBINING MACRON
5	o͞	–	✓	"o\x{303}\x{304}"	LATIN SMALL LETTER O, COMBINING TILDE, COMBINING MACRON
6	õ̱	–	✓	"o\x{304}\x{303}"	LATIN SMALL LETTER O, COMBINING MACRON, COMBINING TILDE
7	õ̱	✓	–	"\x{14D}\x{303}"	LATIN SMALL LETTER O WITH MACRON, COMBINING TILDE

В языке Perl за функции нормализации отвечает стандартный модуль `Unicode::Normalize`. Как правило, весь текст Юникода, получаемый извне, желательно сначала пропускать через процедуру нормализации NFD, а весь текст Юникода, который выводится вовне, – через процедуру NFC, как показано ниже:

```
use v5.14;
use strict;
use warnings;
use warnings FATAL => "utf8"; # возбуждать исключения,
                              # связанные с ошибками кодирования
use open qw(:std :utf8);

use Unicode::Normalize qw(NFD NFC);

while (my $line = <>) {
    $line = NFD($line);
    ...
} continue {
    print NFC($line);
}
```

Этот фрагмент читает текст в кодировке UTF-8 и автоматически декодирует его, возбуждая исключения при обнаружении ошибок кодирования. Первое, что делается в цикле – выполняется нормализация входной строки в форму канонической декомпозиции. Иными словами, все предварительно скомпонованные символы разбиваются на составляющие, к неописуемой радости редукционистов. При этом также переупорядочиваются все знаки, присоединяемые к разным частям базового кода[1].

Без нормализации вы не сможете даже подступиться к решению проблемы, связанной с комбинационными знаками. Рассмотрим графемы, представленные в табл. 6.4.

[1] Если заумствовать, то следует сказать «согласно их каноническим комбинационным классам».

- Номер 4 – это ненормализованная графема. Такое иногда случается.
- Если предположить, что вы выполняете нормализацию NFD, графема 1 превратится в графему 2, 3 и 4 превратятся в графему 5, а графема 7 превратится в графему 6.
- Если предположить, что вы выполняете нормализацию NFC, графема 2 превратится в графему 1, 4 и 5 превратятся в графему 3, а графема 6 превратится в графему 7.
- Это означает, что путем нормализации в *любую* форму, NFD *или* NFC, можно обеспечить возможность сравнения графем 1–2, 3–5 и 6–7 посредством простого оператора `eq`.
- Однако, обратите внимание, что это три разных набора. ☹

Одна приятная новость состоит в том, что шаблоны Perl предлагают отличную поддержку графем, главное знать, как ею пользоваться. Метасимволу `\X` в регулярных выражениях соответствует единственный, видимый пользователем, символ, который на языке Юникода называется групповой графемой.[1]

Большинство групповых графем (но не все) состоит из базового кода и нуля или более комбинационных кодов. Одной широко распространенной двухсимвольной графемой, не имеющей комбинационных знаков, является `"\r\n"`, которая обычно называется CRLF. Метасимвол `\X` соответствует комбинации CRLF как единой групповой графеме, потому что с точки зрения пользователя это единственный символ. В алфавите Japanese также имеются две расширенные графемы, не содержащие комбинационных знаков, HALFWIDTH KATAKANA VOICED SOUND MARK и HALFWIDTH KATAKANA SEMI-VOICED SOUND MARK.

Но обычно групповую графему можно представлять себе, как базовый символ (`\p{Grapheme_Base}`) с произвольным количеством комбинационных знаков, селекторов вариантов, японских огласовок, соединительных знаков нулевой ширины, а также несоединительных знаков (`\p{Grapheme_Extend}*`), следующих непосредственно за ним. Исключение составляет пара CRLF.

Фактически групповые графемы можно считать обычными графемами.[2]

Метасимвол `\X` в операциях поиска по шаблону соответствует любой из семи графем, представленных выше, и даже не требует приведения их к канонической форме. Хорошо, и что дальше? А вот с этого места начинаются сложности. Если вам потребуется знать о графеме больше, чем то, что она является графемой, вам придется проявить умеренную сообразительность при реализации поиска по шаблону. Приведение к NFD – условие *необходимое и достаточное*, чтобы:

- `/^o/` сообщал, что все семь графем начинаются символом «o»;
- `/^o\N{COMBINING TILDE}/` сообщал, что графемы 1–5 начинаются с символа «o» и тильды, но пропускал графемы 6 и 7;
- Найти все семь графем. Придется использовать `/^o\pM*?\N{COMBINING TILDE}/`.

---

[1] Строго говоря, метасимволу `\X` соответствует то, что в стандарте Юникода описывается, как расширенная групповая графема. Очевидно, разработчикам стандартов платят за количество слов.

[2] Именно это мы и делаем. Нам не платят за количество слов.

А вот попытка решить вопрос поиска полного символа, оставляющая за кадром такие тонкости, как использование \p{Grapheme_Extend} вместо \pM, а также \p{Grapheme_Base} (если уместно) вместо \PM:

```
$o_tilde_rx = qr{ o \pM *? \x{COMBINING TILDE} \pM* }x;
```

Гораздо более простой подход к сравнению строк (без учета диакритических знаков) приводится в следующем разделе, «Сравнение и сортировка строк Юникода».

Метасимвол \X – единственный в ядре Perl, знающий о существовании графем. Встроенные функции, такие как substr, length, index, rindex и pos манипулируют кодами символов, а не графемами. Поэтому \X – это ваш молоток. С ним весь набор Юникода начинает напоминать гвозди. Огромное количество гвоздей.

Представьте, что вам потребовалось обратить порядок следования кодов символов в строке «crème brûlée». Если предположить, что строка нормализована в форму NFD, вы получили бы в результате «éeÎurb emèrc», тогда как в действительности требуется получить «eélûrb emèrc». Вместо этого следует с помощью \X извлечь список графем и переупорядочить их.

```
use v5.14;
use utf8;
my $cb = "crème brûlée";
my $bc = join("" => reverse($cb =~ /\X/g));
say $bc;    # "eélûrb emèrc"
```

Предположим, что переменная $cb ниже всегда содержит строку «crème brûlée», сравним операции над кодами и операции над графемами:

```
my $char_length = length($cb);     # 15 или 12
my $graph_count = 0;
$graph_count++ while $cb =~ /\X/g; # 12
```

Извлечь первую часть можно было бы, как показано ниже:

```
my $piece  = substr($cb, 0, 5);    # "crèm" или "crème"
my($piece) = $cb =~ /(\X{5})/;     # "crème"
```

А изменить последнюю часть следующим образом:

```
substr($cb, -6) = "fraîche";       # "crème brfraîche" или "crème fraîche"
$cb =~ s/\X{6}$/fraîche/;          # "crème fraîche "
```

И вставить «bien»:

```
substr($cb, 5, 0) = " bien";   # "crèm biene brûlée" или "crème bien brûlée"
$cb =~ s/^\X{5}\K/ bien/;      # "crème bien brûlée"
```

Обратите внимание, насколько нестабилен подход, основанный на кодах. В комментариях первый ответ соответствует строкам в NFD, а второй – строкам в NFC. Может показаться, что приведение строк к NFC решит все проблемы, но это не так. С одной стороны, графем, не имеющих предварительно скомпонованных знаков, бесконечно больше, чем графем, имеющих такие знаки, поэтому приведение к форме NFC не гарантирует избавление от комбинационных знаков.

Кроме того, форма NFC в действительности сложнее в использовании, поэтому мы рекомендуем всегда приводить входящие данные к форме NFD. Взгляните, как можно было бы выделить слова с двумя «е», такие как «crème» и «brûlée». Простейший и самый надежный способ:

```
/ e .*? e /x
```

будет работать только со строками, нормализованными в форму NFD, а не NFC. А если вы думаете, что, используя форму NFC, можно гарантировать то же самое, записав:

```
/ [eéè] .*? [eéè] /x
```

то быстро поймете, что ошиблись, столкнувшись со словом «crêpes». Добавление одного «ê» как будто решает проблему, но таким путем вы очень скоро придете к совершенно сумасшедшему шаблону:

```
/ [èéêëēéęęě] [èéêëēéęęě] /x   # два символа e в строке
```

который также окажется неработоспособным, если кто-то подсунет ему «e» с подчеркиванием, поскольку этот символ не имеет предварительно скомпонованной графемы. Если (и только если) ваши строки будут приведены к форме NFD, следующий шаблон будет работать всегда:

```
/ (?: (?=e) \X ){2} /x
```

Это решение обеспечивает надежный и неразрушающий способ сопоставления без учета акцентов: используйте метасимвол `\X`, соответствующий графеме, и введите ограничение, требующее, чтобы графема начиналась с искомого базового символа. Единственное, чего нельзя добиться таким способом – реализовать поиск в строках, приведенных к форме NFD (или NFKD), букв, которые не поддаются декомпозиции, потому что они считаются самостоятельными буквами.

Например, таким способом нельзя отыскать все символы «o» в слове «smørrebrød», потому что LATIN SMALL LETTER O WITH STROKE не имеет составного представления, где символ «o» был бы выделен из графемы. В имени «Ævar Arnfjörð Bjarmason» вы сможете отыскать все буквы «o» после декомпозиции, но не сможете отыскать символы «e» и «d», потому что LATIN CAPITAL LETTER AE не разбивается на «a» и «e», а LATIN SMALL LETTER ETH не превращается в «d».

Во всяком случае, при использовании декомпозиции. Однако сравнение с помощью специализированного объекта из `Unicode::Collate` позволило бы отыскать все три символа. В следующем разделе, «Сравнение и сортировка строк Юникода», мы покажем, как это делается.

Необходимость использовать метасимвол `\X` каждый раз, когда возникает потребность в использовании встроенных строковых функций, выглядит несколько странной. Альтернативное решение заключается в использовании модуля `Unicode::GCString` из CPAN.

Обычные строки в Perl всегда интерпретируются как последовательности кодов, но этот объектно-ориентированный модуль позволяет работать со строками, как с последовательностями групповых графем Юникода. Ниже показано, как можно использовать методы из этого модуля для выполнения обсуждавшихся выше операций над последовательностями графем:

```
my $gs = Unicode::GCString("crème brûlée");

say $gs->length();
say $gs->substr(0,5);
$gs->substr(-6, 6, "fraîche");
$gs->substr( 5, 0, " bien");
```

Теперь выбор формы нормализации не имеет значения, потому что метод `length` возвращает ответ в графемах, метод `substr` оперирует графемами, и можно даже использовать методы `index` и `rindex` для поиска литеральных подстрок, получая целочисленное смещение в графемах, а не в кодах символов.

Пожалуй, самый полезный метод в этом модуле – это метод `columns`. Представьте, что необходимо вывести несколько элементов меню, как показано ниже:

```
crème brûlée        £5.00
trifle              £4.00
toffee ice cream    £4.00
```

Как обеспечить выравнивание цен по вертикали? Даже если предположить, что вывод осуществляется с использованием моноширинного шрифта, следующее решение не поможет:

```
printf("%-25s £%.2f\n", $item, $price);
```

потому что Perl предполагает, что каждый код занимает ровно одну позицию в строке, что на практике не так.

Метод `columns` возвращает количество позиций, которые займет строка при выводе. Часто это число совпадает с количеством графем в строке, но не всегда. Некоторые символы Юникода считаются «широкими», в том смысле, что при выводе они занимают две позиции. Это настолько типично для символов восточноазиатских алфавитов, что в Юникоде существуют специальные свойства `East_Asian_Width=Wide` и `East_Asian_Width=Full`, которые указывают, что символ занимает две позиции при выводе.

Некоторые символы вообще не занимают позиции при выводе, и не только потому, что они являются комбинационными знаками: это могут быть символы управления или форматирования. Плюс ко всему к этому, некоторые комбинационные знаки занимают отдельную позицию при выводе. Единственное, на что вообще можно положиться при использовании моноширинного шрифта, это то, что размер каждого символа будет кратен ширине одной позиции.

Ниже демонстрируется одно из возможных решений дополнения строки до определенной длины:

```
sub pad {
    my($s, $width) = @_;
    my $gs = Unicode::GCString->new($s);
    return $gs . (" " x ($width - $gs->columns));
}

printf("%s £%.2f\n", pad($item, 25), $price);
```

Теперь ваши строки выравняются по вертикали, даже если будут содержать форматирующие символы, комбинационные знаки или широкие символы.

Несмотря на всю свою привлекательность, модуль `Unicode::GCString` в действительности является всего лишь вспомогательным модулем для другого, более крупного модуля, который решает более сложную проблему: модуля `Unicode::LineBreak` из CPAN. Последний реализует алгоритм разбиения строк (Unicode Line Breaking Algorithm), описываемый в UAX#14, приложении к стандарту Юникода. Он может пригодиться вам, когда потребуется разбить текст Юникода на абзацы, как это делает программа UNIX *fmt*(1) из модуля `Text::Autoformat`. В качестве примера

можно предложить программу *unifmt* из модуля `Unicode::Tussle`. Она все делает правильно, даже столкнувшись с восточноазиатскими широкими символами, табуляциями, комбинационными символами и невидимыми кодами форматирования.

Сравнение и сортировка строк Юникода

Встроенные функции `sort` и `cmp` не сравнивают строки по алфавитному признаку. Вместо этого сравниваются числовые значения кодов символов в одной строке с числовыми значениями кодов соответствующих символов в другой строке. Такой подход плохо работает с текстом, где наряду с символами, общими для разных языков, встречаются символы, характерные для каждого отдельно взятого языка. Это связано не только с наличием кодов, числовые значения которых не совпадают с алфавитным порядком, – числа и другие последовательности также могут вносить беспорядок, из-за того, что некоторые были добавлены в наборы символов, когда они были еще маленькими, а другие – когда наборы символов имели уже приличный размер, как Topsy. Например, символы показателей степеней 2 и 3 появились в Latin-1 на ранних этапах его становления. Поэтому, кстати, при сортировке они располагаются первыми:

```
use v5.14;
use utf8;
my @exes = qw( x⁷ x⁰ x⁸ x³ x⁶ x⁵ x⁴ x² x⁹ x¹ );
@exes = sort @exes;
say "@exes";

# выведет: x² x³ x¹ x⁰ x⁴ x⁵ x⁶ x⁷ x⁸ x⁹
```

Поскольку числовые значения кодов не соответствуют алфавитному порядку следования символов, строки будут сортироваться в порядке, который, не будучи совсем уж случайным, не соответствует ожиданиям пользователей. Функция `sort` хороша в основном для быстрого упорядочивания, когда порядок следования всегда будет одним и тем же, пусть и не совпадающим с алфавитным. Иногда этого вполне достаточно, но иногда...

Представляем стандартный модуль `Unicode::Collate`, который реализует алгоритм упорядочивания Юникода (Unicode Collation Algorithm, UCA), обладает широкими возможностями настройки и обеспечивает многоуровневую сортировку Юникода. Модуль обладает большим количеством замысловатых особенностей, но часто его применение ограничивается вызовом его метода `sort` без дополнительных параметров:

```
use v5.14;
use utf8;
use Unicode::Collate;
my @exes = qw( x⁷ x⁰ x⁸ x³ x⁶ x⁵ x⁴ x² x⁹ x¹ );
@exes = Unicode::Collate->new->sort(@exes);
say "@exes";

# выведет: x⁰ x¹ x² x³ x⁴ x⁵ x⁶ x⁷ x⁸ x⁹
```

По умолчанию модуль обеспечивает алфавитно-цифровую сортировку. В первом приближении, его принцип действия выглядит так: из текста удаляются все неалфавитноцифровые символы, после чего оставшееся сортируется без учета реги-

стра символов, не в числовом порядке кодов, а в порядке следования в алфавите. Такого рода сортировка используется в словарях, именно поэтому она иногда называется словарной, или лексикографической сортировкой.

Прежде чем люди привыкли пользоваться компьютерами, не умеющими правильно сортировать текст, именно такой порядок сортировки считался (и по сей день зачастую считается) верным. Название книги с запятой после первого слова в перечне названий должно располагаться рядом с таким же названием, не содержащим запятой. Запятые не должны оказывать влияния на сортировку, по крайней мере, когда этого не требуется. Запятые не являются естественной частью алфавитного порядка, как буквы и цифры.

Взгляните, какие результаты дает встроенная функция `sort` (которая сортирует строки так же, как аналогичная команда, имеющаяся в оболочке интерпретатора команд и в большинстве языков программирования):

```
% perl -e 'print sort <>' little-reds
Little Red Mushrooms
Little Red Riding Hood
Little Red Tent
Little Red, More Blue
Little, Red Rider
```

Что за ерунда? Слово «More» должно предшествовать словам «Mushrooms», слова «Rider» и «Riding» должны располагаться рядом, а слово «Tent» должно оказаться в конце. Даже с точки зрения ASCII это не алфавитный порядок; вот алфавитный порядок:

```
% perl -MUnicode::Collate -e '
    print for Unicode::Collate->new->sort(<>)' little-reds
Little Red, More Blue
Little Red Mushrooms
Little, Red Rider
Little Red Riding Hood
Little Red Tent
```

Нам кажется, что вам понравятся результаты этой сортировки Юникода настолько, что вы захотите сохранить этот маленький сценарий на случай, когда потребуется сортировка обычного текста. Он предполагает, что исходный текст имеет кодировку UTF-8, и производит вывод тоже в кодировке UTF-8:

```
#!/usr/bin/perl
use warnings;
use open qw(:std :utf8);
use warnings qw(FATAL utf8);
use Unicode::Collate;
print for Unicode::Collate->new->sort(<>);
```

Более разносторонний вариант этого сценария доступен в виде программы *ucsort*, входящей в комплект `Unicode::Tussle` из архива CPAN.

Большинство людей считает, что с настройками по умолчанию модуль позволяет добиться вполне приемлемых результатов. Он уже знает, как сортировать буквы и числа, плюс умеет обращаться со всеми странностями Юникода, противоречащими ASCII-сортировке, как например упорядочение символов, которые согласно числовому порядку следования кодов оказываются далеко друг от друга, но

при сортировке должны оказываться рядом, учитывает все замысловатые правила Юникода, касающиеся регистра символов, принимает во внимание канонические эквиваленты строк и многое другое.

Плюс, если у вас имеются собственные предпочтения, модуль обладает практически неограниченным потенциалом настройки. Ниже демонстрируется несложная настройка, которая пригодится при сортировке названий книг и фильмов на английском языке. На этот раз символы верхнего регистра предшествуют символам нижнего регистра, перед сортировкой удаляются ведущие артикли и добавляются ведущие нули в числах, благодаря чему название «101 Dalmations» оказывается в списке ниже, чем «7 Brides for 7 Brothers».[1]

```
my $collator = Unicode::Collate->new(
    upper_before_lower => 1,
    preprocess => sub {
        local $_ = shift;
        s/^ (?: The | An? ) \h+ //x;        # отбросить артикли
        s/ ( \d+ ) / sprintf "%020d", $1 /xeg;
        return $_;
    },
);
```

Выше уже было показано, насколько более приемлемой выглядит алфавитная сортировка в сравнении с сортировкой по числовым значениям символов ASCII. В Юникоде ситуация обостряется. Даже если вы используете «всего лишь» английский язык, вам все равно придется сталкиваться не только с символами ASCII. Что если в ваших данных присутствует обозначение «10¢» или «£5»? Даже в тексте исключительно на английском языке могут встретиться фигурные кавычки, замысловатые дефисы и прочие специальные символы, отсутствующие в наборе ASCII. Даже если говорить исключительно о словах, которые можно найти в словаре английского языка, это не освобождает вас от необходимости предусматривать обработку особых случаев. Ниже приводится список слов из оксфордского словаря английского языка (Oxford English Dictionary), отсортированных (по колонкам) с использованием алгоритма UCA в режиме по умолчанию:

Allerød	fête	Niçoise	smørrebrød
après-ski	feuilleté	piñon	soirée
Bokmål	flügelhorn	plaçage	tapénade
brassière	Gödelian	prêt-à-porter	vicuña
caña	jalapeño	Provençal	vis-à-vis
crème	Madrileño	quinceañera	Zuñi
crêpe	Möbius	Ragnarök	α-ketoisovaleric acid
désœuvrement	Mohorovičić discontinuity	résumé	(α-)lipoic acid
Fabergé	moiré	Schrödinger	(β-)nornicotine
façade	naïve	Shijō	ψ-ionone

[1] Ведущие нули необходимы, потому что, несмотря на известность числовых значений цифровых кодовых пунктов, инструменты сортировки обычно не понимают, что число 9 должно предшествовать числу 10, если не использовать трюк, подобный этому.

Едва ли кто-то предпочел бы увидеть эти слова, отсортированные в порядке ASCII. Это не самое приятное зрелище. И это текст, содержащий только латинские символы. Взгляните на слова, встречающиеся в греческой мифологии, отсортированные по кодам символов:

Δύσις	Ἄσβολος	Διόνυσος	Φάντασος	Μεγαλήσιος
Ασβετος	Αγχίσης	Ἑσπερίς	Ἄγδιστις	Τελεσφόρος
Ασωπός	Λάχεσις	Ἕσπερος	Ἀστραῖος	Χρυσόθεμις
Θράσος	Νέμεσις	Εύνοστος	Ασκληπιός	Ἀριστόδημος
Ιάσιος	Περσεύς	Ἥφαιστος	Ἥφαιστος	Ἀριστόμαχος
Νέσσος	Ἄδραστος	Ηωσφόρος	Ἀρισταῖος	Λαιστρυγόνες
Πέρσης	Ἄλκηστις	Θρασκίας	Ἀσκάλαφος	
Πίστις	Αίγισθος	Πάσσαλος	Βορυσθενίς	
Χρύσος	Αργέστης	Πρόφασις	Ἑσπερίδες	

Даже если вы не знаете греческий алфавит, вы все равно сможете заметить, насколько сортировка по кодам не соответствует алфавитной: просто пробегите взглядом по первым буквам в каждом столбце. Видите, как они «прыгают» с места на место? При сортировке с использованием алгоритма UCA в режиме по умолчанию они располагаются правильном порядке:

Ἄγδιστις	Ασβετος	Ἑσπερίδες	Ιάσιος	Πίστις
Αγχίσης	Ἄσβολος	Ἑσπερίς	Λαιστρυγόνες	Πρόφασις
Ἄδραστος	Ἀσκάλαφος	Ἕσπερος	Λάχεσις	Τελεσφόρος
Αίγισθος	Ασκληπιός	Εύνοστος	Μεγαλήσιος	Φάντασος
Ἄλκηστις	Ἀστραῖος	Ἥφαιστος	Νέμεσις	Χρυσόθεμις
Αργέστης	Ασωπός	Ἥφαιστος	Νέσσος	Χρύσος
Ἀρισταῖος	Βορυσθενίς	Ηωσφόρος	Πάσσαλος	
Ἀριστόδημος	Διόνυσος	Θρασκίας	Περσεύς	
Ἀριστόμαχος	Δύσις	Θράσος	Πέρσης	

Нам удалось вас убедить? Теперь посмотрим, как в действительности работает алгоритм UCA, а затем познакомимся с его настройками.

Алгоритм Unicode Collation Algorithm предусматривает многоуровневую сортировку. Вы с ней уже сталкивались. Представьте теперь, что вы написали свою функцию сравнения для передачи в качестве функции обратного сравнения встроенной в язык `sort`, и ваша функция выглядит так:

```
@collated_text = sort {

    primary($a) <=> primary($b)
                ||
    secondary($a) <=> secondary($b)
                  ||
    tertiary($a) <=> tertiary($b)
                 ||
```

```
        quaternary($a) <=> quaternary($b)

    } @random_text;
```

Это – многоуровневая сортировка. Если говорить упрощенно, именно так работает алгоритм UCA. Каждая из четырех функций возвращает число, определяющее вес данного уровня. Только когда на первом уровне обнаруживается совпадение, сравнение продолжается на втором уровне и т.д.

Ниже приводится несколько упрощенное, но достаточно полное описание алгоритма:

Первый уровень: сравнение букв

Проверяется равенство базовых букв.[1] На этом уровне игнорируются все символы, не являющиеся буквами – они просто пропускаются в процессе сканирования строки. Если буквы в одних и тех же относительных позициях не совпадают, это несоответствие определяет словарный порядок их следования.

Если выполняется сортировка текста, содержащего только символы из алфавита Latin, будет получен обычный алфавитный порядок «abc...», который вы изучали в школе, т.е. слово «Fred» будет предшествовать слову «freedom», как и словосочетание «free beer». Причина предшествования «free beer» слову «freedom» состоит в том, что пятая буква в первой строке – «b», а она предшествует пятой букве во второй строке – букве «d». Разобрались с механизмом работы? Это и есть словарный порядок. Пробелы не участвуют в сортировке.

Второй уровень: сравнение диакритических знаков

Если все буквы совпадают, на втором этапе выполняется сравнение диакритических знаков (точнее, комбинационных знаков: множества диакритических и комбинационных знаков перекрываются, но не полностью). По умолчанию сопоставление диакритических знаков выполняется слева направо, но направление можно поменять на обратное, как того требуют правила французского языка. (Классическим примером обычной сортировки по диакритическим знакам в направлении слева направо может служить последовательность слов `cote < coté < côte < côté`, которая во французском языке, согласно правилу сортировки справа налево, должна быть отсортирована иначе: `cote < côte < coté < côté`; два слова в середине меняются местами. Это связано с особенностями флективной морфологии французского языка.)

Третий уровень: сравнение регистров

Если все буквы и диакритические знаки совпадают, далее выполняется сравнение регистров символов. По умолчанию символы нижнего регистра предшествуют символам верхнего регистра, однако этот порядок можно легко изменить, добавив параметр `upper_before_lower => 1` при конструировании объекта, выполняющего сопоставление.

Четвертый уровень: сравнение всего остального

Если все буквы, диакритические знаки и регистры символов совпадают, производится сравнение всех остальных символов, таких как знаки пунктуации,

---

[1] А также цифр и некоторых других символов, которые вы, возможно, не считаете буквами (а они таковыми являются) – просто имейте в виду, что в данном случае под буквами подразумеваются не только буквы в прямом понимании.

специальные и пробельные символы, которые игнорировались на предыдущих уровнях. На этом уровне содержание строк учитывается полностью.

Использование уровней – дело добровольное. Вы можете, например, задействовать только первый уровень, где учитываются лишь базовые буквы и больше ничего.

Именно так выполняется сравнение строк «без учета акцентов» с использованием метода `eq` объекта, реализующего сравнение.

Нормализация не всегда действует наверняка. Например, нормализация бессильна, если вам требуется, чтобы буквы «o», «õ» и «ø» считались одинаковыми, потому что буква `LATIN SMALL LETTER O WITH STROKE` не поддается декомпозиции в нечто другое, начинающееся с базовой буквы «o». С другой стороны, при сравнении букв `Unicode::Collate` обычно считает «o», «õ» и «ø» одной и той же буквой. Но только не в алфавитах Swedish или Hungarian.

Аналогично обстоят дела с буквами «d» и «ð» – буква LATIN SMALL LETTER ETH не поддается декомпозиции в нечто другое, имеющее в составе базовую букву «d», но реализация алгоритма UCA считает их одной и той же буквой. Исключение составляет алфавит Icelandic (код «is» региональных настроек), где «d» и «ð» – совершенно разные буквы.

Если необходимо, чтобы объект, реализующий сравнение, игнорировал регистр, но учитывал акценты, укажите ему на необходимость выполнять сравнение только на первых двух уровнях и пропускать остальные, передав конструктору параметр `level => 2`.

Вот полный синтаксис всех необязательных параметров настройки конструктора для версии модуля v0.81:

```
$Collator = Unicode::Collate->new(
    UCA_Version => $UCA_Version,
    alternate => $alternate, # псевдоним для 'variable'
    backwards => $levelNumber, # или \@levelNumbers
    entry => $element,
    hangul_terminator => $term_primary_weight,
    ignoreName => qr/$ignoreName/,
    ignoreChar => qr/$ignoreChar/,
    ignore_level2 => $bool,
    katakana_before_hiragana => $bool,
    level => $collationLevel,
    normalization => $normalization_form,
    overrideCJK => \&overrideCJK,
    overrideHangul => \&overrideHangul,
    preprocess => \&preprocess,
    rearrange => \@charList,
    rewrite => \&rewrite,
    suppress => \@charList,
    table => $filename,
    undefName => qr/$undefName/,
    undefChar => qr/$undefChar/,
    upper_before_lower => $bool,
    variable => $variable,
);
```

Дополнительную информацию о параметрах конструктора можно найти на странице справочного руководства модуля. Хотя этот модуль и является частью стан-

дартной библиотеки Perl, он также доступен в архиве CPAN. Благодаря этому есть возможность обновлять его независимо от ядра Perl. Версия Perl v5.14 поставляется с модулем `Unicode::Collate` версии v0.73, поэтому совершенно очевидно, что с тех пор модуль обновился. Вам не нужно устанавливать самую современную версию Perl, чтобы использовать последнюю версию модуля. Он поддерживает даже такие старые версии Perl, как v5.6, и обеспечивает опережающую совместимость с последними версиями стандарта Юникода посредством аргумента конструктора `UCA_Version`.

Использование UCA с функцией sort

В реальной жизни встроенная функция `sort` обычно вызывается двумя способами: вообще без подпрограммы сравнения, либо с блоком кода в виде аргумента, играющим роль подпрограммы сравнения. В первом случае с успехом можно использовать метод `sort` из модуля `Unicode::Collate`, но во втором... Во втором случае можно воспользоваться другим методом объекта, реализующего сравнение, который называется `getSortKey`.

Предположим, что имеется программа, использующая встроенную функцию `sort`, как показано ниже:

```
@srecs = sort {
    $b->{AGE} <=> $a->{AGE}
              ||
    $a->{NAME} cmp $b->{NAME}
} @recs;
```

И вот нам пришло в голову, что текст следует отсортировать по полю `NAME` в алфавитном порядке, а не по числовым значениям кодов. Для этого достаточно просто попросить объект сравнения вернуть вам двоичный ключ сортировки для каждой текстовой строки, участвующей в сортировке. В отличие от обычного текста, если передать эти двоичные ключи оператору `cmp`, он волшебным образом отсортирует их в требуемом вам порядке.

Блок кода для передачи функции `sort` теперь выглядит так:

```
my $collator = Unicode::Collate->new();
for my $rec (@recs) {
    $rec->{NAME_key} = $collator->getSortKey( $rec->{NAME} );
}
@srecs = sort {
    $b->{AGE} <=> $a->{AGE}
              ||
    $a->{NAME_key} cmp $b->{NAME_key}
} @recs;
```

Конструктору можно передавать любые необязательные аргументы для достижения желаемых результатов, включая предварительную обработку.

Объекты, реализующие сравнение, можно также использовать для простого сравнения без учета акцентов и регистра символов. В этом есть определенный смысл – если есть возможность упорядочивать строки, значит, есть возможность определять их эквивалентность при определенных настройках упорядочивания. То есть, вам остается лишь выбрать нужную семантику упорядочивания. Например, если установить уровень сравнения 1, сравниваться будут только буквы, без учета ре-

гистра символов и наличия диакритических знаков. В этом вам помогут методы объекта, реализующего сравнение, такие как `eq`, `substr` и `index`. (Однако от нормализации придется отказаться, потому что иначе изменятся смещения кодов.) Например:

```
use v5.14;
use utf8;
use Unicode::Collate;
my $Collator = Unicode::Collate->new(
                   level => 1,
                   normalization => undef,
               );

my $full = "Gabriel García Márquez";
for my $sub (qw[MAR CIA]) {
    if (my($pos,$len) = $Collator->index($full, $sub)) {
        my $match = substr($full, $pos, $len);
        say "Соответствие литералу ‹$sub› найдено в ‹$full› в виде ‹$match›";
    }
}
```

Если запустить этот фрагмент, он выведет:

```
Соответствие литералу ‹MAR› найдено в ‹Gabriel García Márquez› в виде ‹Már›
Соответствие литералу ‹CIA› найдено в ‹Gabriel García Márquez› в виде ‹cía›
```

Пожалуйста, не сообщайте об этом в ЦРУ (CIA).

Сортировка с учетом региональных настроек

Алгоритм UCA с настройками по умолчанию с успехом справляется с текстами на английском и на многих других языках, включая ирландско-гэльский, индонезийский, итальянский, грузинский, голландский, португальский и немецкий (за исключением телефонных книг!). Однако для алфавитной (или не алфавитной – кому как вздумается) сортировки текстов на многих других языках требуется применить дополнительные настройки.

Например, в скандинавских языках буквы с диакритическими знаками при сортировке должны следовать за буквой «*z*», а не располагаться рядом с похожими. Даже в испанском языке есть свои особенности: буква «*ñ*» не считается обычной буквой «n» с тильдой, как буквы «*ã*» и «*õ*» в португальском языке. В испанском алфавите это самостоятельная буква (и называется она, конечно, *eñe*), которая должна следовать за буквой «*n*» и предшествовать букве «*o*». Из этого следует, что следующие слова должны сортироваться в таком порядке: *radio*, *ráfaga*, *ranúnculo*, *raña*, *rápido*, *rastrillo*. Обратите внимание, что слово *ranúnculo* должно идти перед словом *raña*, а не после него.

Учесть национальные особенности сортировки текста Юникода позволяет модуль `Unicode::Collate::Locale`. Он распространяется в составе `Unicode::Collate` и потому входит в состав Perl версии v5.14, а также устанавливается вместе с основным модулем при установке из CPAN.

Единственное отличие в API этих двух модулей заключается в наличии у конструктора из `Unicode::Collate::Locale` дополнительного параметра: регионального кода. На момент написания этих строк поддерживалось 70 различных регио-

нальных кодов, включая такие варианты, как немецкие телефонные книги (гласные с умляутами сортируются, как если бы они были обычными гласными, за которыми следует буква «*e*»), традиционный испанский («*ch*» и «*ll*» считаются отдельными графемами со своим местоположением в алфавите), японский, и пять различных способов сортировки текста на китайском языке.

Пользоваться этим дополнительным параметром в действительности очень просто:

```
use Unicode::Collate::Locale;

$coll = Unicode::Collate::Locale->new(locale => "fr");

@french_text = $coll->sort(@french_text);
```

Поскольку `Unicode::Collate::Locale` является подклассом, наследующим `Unicode::Collate`, его конструктор принимает те же дополнительные аргументы, что и конструктор родительского класса, и объекты этого класса поддерживают те же методы, поэтому вы можете использовать их для организации поиска с учетом региональных особенностей, как было показано выше. Ниже демонстрируется пример выбора настроек сортировки, используемой в немецких телефонных книгах, где (к примеру) «*ae*» и «*ä*» считаются одной и той же буквой. Можно просто выполнить непосредственное сравнение

```
state $coll = new Unicode::Collate::Locale::
               locale => "de_ _phonebook",
           ;

if ($coll->eq($a, $b)) { ... }
```

Или выполнить поиск:

```
use Unicode::Collate::Locale;
my $Collator = new Unicode::Collate::Locale::
                locale => "de_ _phonebook",
                level => 1,
                normalization => undef,
              ;

my $full = "Ich müß Perl studieren.";
my $sub = "MUESS";
if (my ($pos,$len) = $Collator->index($full, $sub)) {
    my $match = substr($full, $pos, $len);
    say "Соответствие литералу ‹$sub› найдено в ‹$full› в виде ‹$match›";
}
```

Если запустить этот фрагмент, он выведет:

```
Соответствие литералу ‹MUESS› найдено в ‹Ich müß Perl studieren.› в виде ‹müß›
```

Дополнительные возможности

Всегда следует помнить, что такие сокращенные обозначения символьных классов в Perl, как `\w`, `\s` и даже `\d`, по умолчанию соответствуют многим символам Юникода, что диктуется определенными свойствами символов. Они перечислены в табл. 5.11 и отвечают формальным определениям из приложения «Annex C:

Compatibility Properties» к техническому стандарту Юникода «Unicode Technical Standard #18, Unicode Regular Expressions», версии 13, выпущенной в августе 2008.

Если вы привыкли в своих программах извлекать целые числа при помощи выражения (\d+), этот подход не всегда будет работать корректно с данными в Юникоде. По версии v6.0 стандарта Юникода, метасимволу \d соответствует 420 кодовых пунктов. Если вам это не подходит, используйте /\d/a или /(?a:\d)/, или задействуйте более конкретное свойство \p{POSIX_Digit}.

Если требуется извлекать любые десятичные цифры из любого алфавита и использовать их в программе как числа, можно воспользоваться функцией num из модуля Unicode::UCD.

```
use v5.14;
use utf8;
use Unicode::UCD qw(num);
my $num;
if ("৪৫৬৭" =~ /(\d+)/) {
    $num = num($1);
    printf "Найдено число: %d\n", $num;
    # Найдено число: 4567
}
```

Регулярные выражения позволяют проверять свойства символов, однако не способны определять, какими свойствами символ обладает (во всяком случае, без проверки всех свойство по списку). А иногда действительно бывает необходимо знать это. Например, если необходимо узнать, какому алфавиту принадлежит код символа, или к какой основной категории относится символ. Для этого можно использовать тот же самый модуль Unicode::UCD. Ниже приводится программа вывода свойств, которые могут пригодиться при поиске по шаблону.

```
use v5.14;
use utf8;
use warnings;

use Unicode::UCD       qw( charinfo );
use Unicode::Normalize qw( NFD );

## раскомментируйте следующую строку, чтобы использовать декомпозицию
my $mystery = ## NFD
                "৯¾ç̃Ω";
for my $chr (split //, $mystery) {
    my $ci = charinfo(ord $chr);
    print "U+", $$ci{code};
    printf ' \N{%s}'."\n\t", $$ci{name};
    print " gc=",            $$ci{category};
    print " script=",        $$ci{script};
    print " BC=",            $$ci{bidi};
    print " mirrored=",      $$ci{mirrored};
    print " ccc=",           $$ci{combining};
    print " nv=",            $$ci{numeric};
    print "\n";
}
```

После запуска эта программа выведет:

```
U+096D \N{DEVANAGARI DIGIT SEVEN} gc=Nd script=Devanagari
       BC=L mirrored=L ccc=0 nv=7
U+00BE \N{VULGAR FRACTION THREE QUARTERS} gc=No script=Common A
       BC=ON mirrored=ON ccc=0 nv=3/4
U+00E7 \N{LATIN SMALL LETTER C WITH CEDILLA} gc=Ll script=Latin
       BC=L mirrored=L ccc=0 nv=
U+1F6F \N{GREEK CAPITAL LETTER OMEGA WITH DASIA AND PERISPOMENI}
       gc=Lu script=Greek BC=L mirrored=L ccc=0 nv=
```

Если удалить комментарий, препятствующий декомпозиции NFD, будет выведено:

```
U+096D \N{DEVANAGARI DIGIT SEVEN} gc=Nd script=Devanagari
       BC=L mirrored=L ccc=0 nv=7
U+00BE \N{VULGAR FRACTION THREE QUARTERS} gc=No script=Common
       BC=ON mirrored=ON ccc=0 nv=3/4
U+0063 \N{LATIN SMALL LETTER C} gc=Ll script=Latin
       BC=L mirrored=L ccc=0 nv=
U+0327 \N{COMBINING CEDILLA} gc=Mn script=Inherited
       BC=NSM mirrored=NSM ccc=202 nv=
U+03A9 \N{GREEK CAPITAL LETTER OMEGA} gc=Lu script=Greek
       BC=L mirrored=L ccc=0 nv=
U+0314 \N{COMBINING REVERSED COMMA ABOVE} gc=Mn script=Inherited
       BC=NSM mirrored=NSM ccc=230 nv=
U+0342 \N{COMBINING GREEK PERISPOMENI} gc=Mn script=Inherited
       BC=NSM mirrored=NSM ccc=230 nv=
```

Определение пользовательских границ в регулярных выражениях

Метасимволы \b и \B, обозначающие границу слова и не-(границу слова), соответственно, опираются на текущее определение метасимвола \w (здесь подразумевается, что они изменяют свое значение наряду с \w при переходе на семантику ASCII с помощью модификатора /a или /aa).

Если это не тот тип границ, что вы ищете, всегда можно создать собственное определение границ, опираясь на произвольные условия, такие как границы алфавита. Ниже приводится такое определение \b:

```
(?(?<=    \w) # если слева символ слова
      (?! \w) #      тогда справа должен быть не символ слова
  |   (?= \w) #      иначе справа должен быть символ слова
)
```

А так выглядит определение \B:

```
(?(?<=    \w) # если слева символ слова
      (?= \w) #      тогда справа должен быть символ слова
  |   (?! \w) #      иначе справа должен быть не символ слова
)
```

Теперь, когда вы знаете, как определяются границы слов и, соответственно, точки, таковыми не являющиеся, то сможете определять собственные границы, из-

меняя условия там, где находится метасимвол \w в шаблонах выше. Нужно лишь проследить за тем, чтобы ваше определение создавало условие фиксированной ширины – так его можно будет использовать в ретроспективной проверке. Это означает, что нельзя использовать такие метасимволы, как \X или \R, соответствия которым имеют переменную длину. Проще всего для этих целей использовать свойства или классы символов. Например, для определения символов из греческого алфавита можно было бы использовать свойство \p{Greek}, но лучше будет добавить свойство Inherited, чтобы не пропустить комбинационные знаки, поэтому используйте класс [\p{Greek}\p{Inherited}].

Например, ниже представлены подпрограммы для регулярного выражения, реализующие такого рода действия:

```
(?(DEFINE)
    (?<greeklish>           [\p{Greek}\p{Inherited}] )
    (?<ungreeklish>         [^\p{Greek}\p{Inherited}] )
    (?<greek_boundary>
        (?(?<=      (?&greeklish))
            (?!     (?&greeklish))
          | (?=     (?&greeklish))
        )
    )
    (?<greek_nonboundary>
        (?(?<=      (?&greeklish))
              (?=   (?&greeklish))
            |  (?!  (?&greeklish))
        )
    )
)
```

Для классов символов, являющихся результатом сложения, вычитания, отрицания и пересечения существующих свойств Юникода, как в подпрограмме <greeklish> выше, может пригодиться возможность определять пользовательские свойства. Нестандартные свойства выглядят как самые обычные свойства. Например:

```
sub IsGreeklish {
    return <<'END';
+utf8::IsGreek
+utf8::IsInherited
END
}
```

Теперь можно использовать \p{IsGreeklish} и \P{IsGreeklish} в шаблонах, скомпилированных в том же пакете, что и подпрограммы. Как собрать все это воедино, рассказывается в следующем разделе.

Укрепляем характер созданием символов

Чтобы определить собственное свойство, необходимо написать подпрограмму с именем свойства (см. главу 7). Из соображений безопасности (неквалифицированное) имя такой подпрограммы должно начинаться с префикса Is или In. Подпрограмма должна быть определена в пакете, где используется свойство (см. главу 10), т.е. если свойство требуется в нескольких пакетах, его необходимо либо

импортировать из модуля (см. главу 11), либо наследовать как метод класса из пакета, в котором оно определяется (см. главу 12).

Помимо соответствия организационным требованиям подпрограмма должна возвращать данные в формате файлов, находящихся в каталоге *PATH_TO_PERLLIB/unicode/Is*. То есть, возвращать список символов или диапазонов символов в шестнадцатеричном виде, по одному в строке. Если возвращается диапазон, два числа, представляющих диапазон, должны разделяться символом табуляции. Допустим, что требуется создать свойство, которое имело бы истинное значение для символов, попадающих в диапазон любой из японских слоговых азбук (kana), известных как хирагана (hiragana) и катакана (katakana). Для этого можно было бы определить два диапазона:

```
sub InKana {
    return <<'END';
3040    309F
30A0    30FF
END
}
```

С другой стороны, это же свойство можно было бы определить через существующие свойства:

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

Вычитание множеств выполняется с помощью префикса «–». Допустим, что необходимо обеспечить соответствие свойства только актуальным символам, а не диапазонам. Исключить неопределенные коды можно вот так:

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

Можно также использовать дополнения наборов символов при помощи префикса «!»:

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

Пересечения определяются с помощью префикса «&», что бывает полезно для определения общих символов, входящих в два (или более) класса.

```
sub IsGraecoRomanTitle {<<'END_OF_SET'}
+utf8::IsLatin
+utf8::IsGreek
&utf8::IsTitle
END_OF_SET

sub IsGreekTitle {<<'END_OF_SET'}
+main::IsGraecoRomanTitle
-utf8::IsLatin
END_OF_SET
```

Важно помнить, что префикс «&» не может предшествовать первому множеству, иначе получится пересечение с пустым множеством, т.е. пустое множество.

В самом Perl используются точно такие же уловки для определения «традиционных» классов символов (таких как `\w`), когда вы включаете их в собственные классы символов (такие как `[-.\w\s]`). Кому то может показаться, что, чем сложнее правила, тем медленнее они будут выполняться. Но в действительности, как только Perl вычислит битовый шаблон для конкретного 64-битового образца свойства, он сохранит его, и никогда не будет выполнять повторный перерасчет шаблона. (Применение 64-битовых образцов даже не требует декодирования данных в кодировке UTF-8 при поиске.) Так что все классы символов, встроенные или ваши собственные, работают одинаково быстро.

Чтобы увидеть другой подход к настройке простым изменением синтаксиса классов символов в квадратных скобках, загляните в модуль `Unicode::Regex::Set` из архива CPAN.

Создавая пользовательские свойства с именами, назначенными пользователем, можно даже организовать управление неиспользуемыми кодами символов, не прибегая к загадочным числовым значениям. Например, в Юникод пока не был включен алфавит Tengwar (эльфийский)[1], хотя в планах он уже стоит – в конце концов, существует уже множество карт Средиземья. Но это не останавливает дизайнеров шрифтов, создающих великолепные шрифты для символов из алфавита Tengwar. Некоторые шрифты используют блоки кодов, зарезервированных непосредственно для алфавита Tengwar, но при этом большинство используемых кодов приходится на область для частного использования. В любом случае, этим кодовым числам еще не присвоены ни имена, ни свойства.

Для Perl это не является барьером, потому что он позволяет легко создавать собственные имена и свойства для символов. Один из существующих в Perl модулей поддержки алфавита Tengwar описывает имена символов следующим образом:

```
TENGWAR LETTER TINCO TENGWAR DIGIT ZERO
TENGWAR LETTER PARMA TENGWAR DIGIT ONE
TENGWAR LETTER CALMA TENGWAR DIGIT TWO
TENGWAR LETTER QUESSE TENGWAR DIGIT THREE
```

что позволяет пользоваться ими, как показано ниже:

```
if ($elvish =~ /\N{TENGWAR LETTER SILME NUQUERNA}/) {...}
```

[1] Cirth (Кирт, Кертар) и Tengwar (Тенгвар) – изобретенные Дж. Р.Р. Толкиеном два вида эльфийских алфавитов – рунический (Кертар) и буквенный (Тенгвар). – *Прим. ред.*

без всяких помех. К кодам символов из алфавита Tengwar можно даже применять charnames::viacode, чтобы получать их имена. Более того, в модуле даже определены свойства символов из алфавита Tengwar:

```
In_Tengwar              In_Tengwar_Numerals
In_Tengwar_Consonants   Is_Tengwar_Decimal
In_Tengwar_Vowels       Is_Tengwar_Duodecimal
In_Tengwar_Alphabetics  In_Tengwar_Marks
In_Tengwar_Punctuation  In_Tengwar_Alphanumerics
```

что позволяет писать такой программный код на Perl:

```
print "W" if /\p{In_Tengwar_Alphanumerics}/;
print "A" if /\p{In_Tengwar_Alphabetics}/;
print "C" if /\p{In_Tengwar_Consonants}/;
print "V" if /\p{In_Tengwar_Vowels}/;
```

или даже:

```
$TENGWAR_GRAPHEME = qr{
    (?>
        (?= \p{In_Tengwar} ) \P{In_Tengwar_Marks}
        \p{In_Tengwar_Marks} *
    ) | \p{In_Tengwar_Marks}
}x;
```

Попытки написать нечто подобное без применения имен абстракций для символов и свойств больше напоминают попытки писать программы с использованием числовых адресов в памяти вместо имен переменных. Безусловно, при большом желании это вполне возможно, но такой стиль не будет гармонировать с имеющимися возможностями, и подобные программы будет невероятно сложно читать и сопровождать. Позволяя определять собственный язык даже для таких узкоспециализированных применений, Perl помогает писать программный код, который выглядит чище и проще.

Ссылки

Язык Perl близко, насколько это возможно, придерживается стандарта Юникода во всех возможных аспектах. Этот стандарт включает различные приложения и технические отчеты. Некоторые из них имеют прямое отношение к темам, обсуждавшимся выше, включая:

UAX #44: Unicode Character Database

UTS #18: Unicode Regular Expressions

UAX #15: Unicode Normalization Forms

UTS #10: Unicode Collation Algorithm

UAX #29: Unicode Text Segmentation

UAX #14: Unicode Line Breaking Algorithm

UAX #11: East Asian Width