2

Основные навыки владения регулярными выражениями

Задачи, представленные в этой главе, не являются реальными задачами, которые будут ставить перед вами босс или ваши заказчики. Это, скорее, технические задачи, с которыми вы будете сталкиваться при создании регулярных выражений, предназначенных для решения практических задач. Например, в первом рецепте объясняется, как с помощью регулярного выражения отыскать определенный текст и как обрабатывать символы, имеющие специальное значение в регулярных выражениях. Само по себе это не может быть целью, потому что нет никакой необходимости использовать регулярные выражения, когда требуется отыскать определенный текст. Однако при создании регулярных выражений вам наверняка потребуется отыскивать соответствия с определенным текстом, поэтому необходимо знать, какие символы следует экранировать, а как раз об этом и рассказывается в рецепте 2.1.

Первыми приводятся рецепты, описывающие очень простые приемы использования регулярных выражений. Если у вас уже имеется некоторый опыт работы с регулярными выражениями, вы можете просто бегло просмотреть их или вообще пропустить. Последующие рецепты в этой главе определенно будут содержать новую для вас информацию, если, конечно, вы не прочитали от корки до корки книгу «Mastering Regular Expressions» Джеффри Фридла (Jeffrey E. F. Friedl), выпущенную издательством O'Reilly[1].

Мы придумывали рецепты для этой главы так, чтобы каждый из них описывал один аспект синтаксиса регулярных выражений. Все вместе они образуют законченный учебник по регулярным выражениям.

---

[1] Джеффри Фридл «Регулярные выражения», 3-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2008.

Рекомендуем прочитать его от начала до конца, чтобы твердо усвоить основы регулярных выражений. Однако можно сразу перейти к регулярным выражениям, имеющим практическую ценность, которые приводятся в главах с 4 по 9, и возвращаться по ссылкам к этой главе при встрече с незнакомыми синтаксическими конструкциями.

В этой вводной главе рассматриваются только регулярные выражения и полностью исключаются из рассмотрения какие-либо вопросы, связанные с программированием. Следующая глава наполнена листингами программного кода. Вы можете заглянуть в раздел «Языки программирования и диалекты регулярных выражений» главы 3, чтобы узнать, какой диалект регулярных выражений используется в вашем языке программирования. Сами диалекты, о которых будет говориться в этой главе, были представлены в главе 1 в разделе «Диалекты регулярных выражений, рассматриваемые в этой книге».

2.1. Соответствие литеральному тексту

Задача

Создать регулярное выражение, в точности соответствующее следующему восхитительному предложению: `The punctuation characters in the ASCII table are: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~`.

Цель этого рецепта – познакомить вас с символами, имеющими специальное значение в регулярных выражениях, и с символами, которые всегда соответствуют самим себе.

Решение

Предложению, указанному выше, соответствует следующее регулярное выражение:

```
The•punctuation•characters•in•the•ASCII•table•are:•↵
!"#\$%&'\(\)\*\+,-\./:;<=>\?@\[\\]\^_`\{\|}~
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Любое регулярное выражение, в котором символы `$()*+.?[\^{|` отсутствуют, просто соответствует само себе. Например, чтобы отыскать в тексте фразу `Mary had a little lamb`, достаточно воспользоваться регулярным выражением `‹Mary•had•a•little•lamb›`. При этом совершенно неважно, установлен ли флажок Regular Expression (Регулярное выражение) в диалоге поиска текстового редактора.

Двенадцать знаков препинания, которые обеспечивают всю магическую силу регулярных выражений, называются *метасимволами*. Если необходимо, чтобы регулярное выражение находило соответствие с этими

символами буквально, их следует экранировать, помещая перед ними символ обратного слэша. То есть регулярному выражению ‹\$\(\)*\+\.\?\[\\\^\{\|› соответствует текст $()*+.?[\^{|.

Обратите внимание, что в списке отсутствуют: закрывающая квадратная скобка], дефис - и закрывающая фигурная скобка }. Первые два символа интерпретируются как метасимволы, только если они стоят после неэкранированной открывающей квадратной скобки [, а } – только после неэкранированного символа {. Нет никакой необходимости экранировать символ }. Правила использования метасимволов в блоках, заключенных между символами [и], описываются в рецепте 2.3.

Экранирование любых других не алфавитно-цифровых символов не влияет на работу регулярного выражения; по крайней мере, это утверждение справедливо для всех диалектов, рассматриваемых в книге. Экранирование алфавитно-цифровых символов либо может придавать им особое значение, либо вызывать синтаксическую ошибку.

Те, кто плохо знаком с регулярными выражениями, часто стремятся экранировать каждый знак препинания, попадающий в поле зрения. Не старайтесь показать всем, что вы – новичок. Разумно используйте экранирование. Частокол ненужных символов обратного слэша осложняет чтение регулярного выражения, особенно когда все эти символы обратного слэша требуется удваивать, чтобы представить регулярное выражение как строковый литерал в исходном тексте программы.

Варианты

Экранирование блока

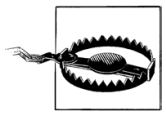
В диалектах, поддерживающих прием *экранирования блока*, есть возможность сделать решение более читаемым:

```
The•punctuation•characters•in•the•ASCII•table•are:•↵
\Q!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~\E
```

Параметры: нет
Диалекты: Java 6, PCRE, Perl

Диалекты Perl, PCRE и Java поддерживают конструкции ‹\Q› и ‹\E›. Конструкция ‹\Q› подавляет интерпретацию всех метасимволов, включая и символ обратного слэша, пока не встретится конструкция ‹\E›. Если опустить конструкцию ‹\E›, все символы, находящиеся в регулярном выражении после конструкции ‹\Q›, будут интерпретироваться буквально.

Единственное преимущество выражения ‹\Q...\E› состоит в том, что его проще читать, чем ‹\.\.\.›.



Хотя версии Java 4 и 5 поддерживают эту особенность, тем не менее лучше ею не пользоваться. Ошибки в реализации приводят к тому, что регулярному выражению с конструкцией ‹\Q···\E› соответствует совсем не то, что соответствует этому же регулярному выражению

в диалектах PCRE, Perl и Java 6. Эти ошибки были исправлены в версии Java 6, благодаря чему этот диалект дает тот же результат, что и диалекты PCRE и Perl.

Сопоставление без учета регистра символов

По умолчанию регулярные выражения чувствительны к регистру символов. Выражению ‹regex› соответствует текст regex, но не Regex, REGEX или ReGeX. Чтобы регулярному выражению ‹regex› соответствовали все эти варианты, необходимо включить режим нечувствительности к регистру символов.

В большинстве приложений для этого достаточно установить или сбросить флажок. Все языки программирования, обсуждаемые в следующей главе, имеют флаг или свойство, с помощью которого можно управлять режимом чувствительности регулярного выражения к регистру символов. В рецепте 3.4, в следующей главе, описывается, как в программном коде применять параметры регулярного выражения, соответствующие приводимому в книге регулярному выражению.

```
ascii
```

Параметры: нечувствительность к регистру символов
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Если режим нечувствительности к регистру символов не был включен за пределами регулярного выражения, это можно сделать с помощью модификатора режима ‹(?i)›, например ‹(?i)regex›. Этот прием работает в диалектах .NET, Java, PCRE, Perl, Python и Ruby. При использовании библиотеки XRegExp его можно применять и в JavaScript.

```
(?i)ascii
```

Параметры: нет
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Диалекты .NET, Java, PCRE, Perl и Ruby поддерживают локальные модификаторы режима, которые воздействуют только на часть регулярного выражения. Например, регулярному выражению ‹sensitive(?i)caseless(?-i)sensitive› соответствует текст sensitiveCASELESSsensitive, но не SENSITIVEcaselessSENSITIVE. Модификатор ‹(?i)› включает режим нечувствительности к регистру символов до конца регулярного выражения, а модификатор ‹(?-i)› отключает его до конца регулярного выражения. Они действуют как простые переключатели.

В рецепте 2.9 показано, как использовать локальные модификаторы режима с группами.

См. также

Рецепт 2.3 описывает символьные классы. Внутри символьных классов используются иные метасимволы, чем за их пределами.

Рецепт 5.14 демонстрирует, как экранировать все метасимволы в строке при встраивании ее в регулярное выражение в качестве литерала. Для этого строка преобразуется в регулярное выражение, соответствующее строке буквально.

В разделе «Пример решения на языке JavaScript» в рецепте 5.2 демонстрируется пример кода на JavaScript, экранирующий все метасимволы регулярных выражений. В некоторых языках программирования для этих целей существуют встроенные команды.

2.2. Соответствие непечатным символам

Задача

Написать регулярное выражение, которому соответствовала бы строка, состоящая из следующих управляющих символов ASCII: bell, escape, form feed, line feed, carriage return, horizontal tab, vertical tab. Эти символы имеют следующие шестнадцатеричные коды ASCII: 07, 1B, 0C, 0A, 0D, 09, 0B.

Этот рецепт демонстрирует, как использовать экранированные последовательности и ссылаться на символы по их шестнадцатеричным кодам.

Решение

```
\a\e\f\n\r\t\v
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Python, Ruby

```
\x07\x1B\f\n\r\t\v
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, Python, Ruby

```
\a\e\f\n\r\t\x0B
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Обсуждение

Для представления семи наиболее часто используемых управляющих символов ASCII имеются специальные *экранированные последовательности*. Все они состоят из символа обратного слэша и следующей за ним буквы. Это тот же самый синтаксис, что используется во многих языках программирования для представления управляющих символов в строковых литералах. В табл. 2.1 приводится перечень наиболее часто используемых непечатных символов и их представления.

В версиях Perl 5.10 и выше и в PCRE 7.2 и выше не поддерживается экранированная последовательность ‹\v› (вертикальная табуляция). В этих диалектах ‹\v› соответствует всем вертикальным пробельным символам,

в число которых входят: вертикальная табуляция, символы разрыва строк и разделители Юникода строк и абзацев. По этой причине в Perl и PCRE следует использовать иной синтаксис определения вертикальной табуляции.

Таблица 2.1. Непечатные символы

Представление	Значение	Шестнадцатеричное представление	Диалекты
‹\a›	bell	0x07	.NET, Java, PCRE, Perl, Python, Ruby
‹\e›	escape	0x1B	.NET, Java, PCRE, Perl, Ruby
‹\f›	form feed	0x0C	.NET, Java, JavaScript, PCRE, Perl, Python, Ruby
‹\n›	line feed (новая строка)	0x0A	.NET, Java, JavaScript, PCRE, Perl, Python, Ruby
‹\r›	carriage return	0x0D	.NET, Java, JavaScript, PCRE, Perl, Python, Ruby
‹\t›	horizontal tab	0x09	.NET, Java, JavaScript, PCRE, Perl, Python, Ruby
‹\v›	vertical tab	0x0B	.NET, Java, JavaScript, Python, Ruby

Диалект JavaScript не поддерживает последовательности ‹\a› и ‹\e›. Поэтому в JavaScript также необходимо использовать другое решение.

Эти управляющие символы и альтернативный синтаксис, который демонстрируется в следующем разделе, с одинаковым успехом могут использоваться в регулярных выражениях как внутри, так и за пределами символьных классов.

Варианты представления непечатных символов

26 управляющих символов

Ниже приводится другой способ представления в регулярных выражениях этих семи управляющих символов ASCII:

```
\cG\x1B\cL\cJ\cM\cI\cK
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Ruby 1.9

С помощью последовательностей от ‹\cA› до ‹\cZ› определяется соответствие с любым из 26 управляющих символов, занимающим позицию от 1 до 26 в таблице символов ASCII. Символ c должен указываться в нижнем регистре. Буква, следующая за ним, в большинстве диалектов может указываться в любом регистре, но мы рекомендуем всегда исполь-

зовать символы верхнего регистра. В диалекте Java это обязательное условие.

Этот синтаксис может оказаться удобным, если вы привыкли вводить управляющие символы в консоли, нажимая клавишу Ctrl одновременно с алфавитной клавишей. Комбинация клавиш Ctrl-H в терминалах соответствует символу забоя (backspace). В регулярных выражениях символу забоя соответствует последовательность ‹\cH›.

Этот синтаксис не поддерживается механизмом регулярных выражений языка Python и классическим механизмом Ruby в Ruby 1.8, но он поддерживается механизмом Onigurama в Ruby 1.9.

Управляющий символ escape, занимающий позицию 27 в таблице ASCII, не имеет эквивалента в английском алфавите, поэтому в наших регулярных выражениях он представлен последовательностью ‹\x1B›.

7-битный набор символов

Ниже приводится еще один способ сопоставления со списком из семи управляющих символов:

```
\x07\x1B\x0C\x0A\x0D\x09\x0B
```

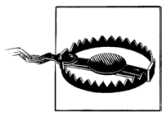
Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Последовательности из экранированного символа \x в нижнем регистре, за которым следуют две шестнадцатеричные цифры в верхнем регистре, соответствует один символ из набора ASCII. На рис. 2.1 показано соответствие шестнадцатеричных комбинаций от ‹\x00› до ‹\x7F› и символов из набора ASCII. Таблица упорядочена так, чтобы первая цифра в комбинации возрастала по направлению вниз, а вторая – вправо.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

***Рис. 2.1.** Таблица символов ASCII*

Соответствие комбинациям от ‹\x80› до ‹\xFF› зависит от того, как они интерпретируются вашим механизмом регулярных выражений и в какой кодировке набран испытуемый текст. Мы не рекомендуем использовать последовательности от ‹\x80› до ‹\xFF›. Вместо них лучше использовать кодовые пункты Юникода, описываемые в рецепте 2.7.



В случае применения Ruby 1.8 или библиотеки PCRE, скомпилированной без поддержки кодировки UTF-8, воспользоваться кодовыми пунктами Юникода будет невозможно. Ruby 1.8 и PCRE без поддержки UTF-8 представляют собой 8-битовые механизмы регулярных выражений. Они ничего не знают о кодировках символов и многобайтовых символах. Последовательность `\xAA` в этих механизмах просто соответствует байту со значением `0xAA` независимо от того, представляет ли этот байт отдельный символ с кодом `0xAA` или является частью многобайтового символа.

См. также

Рецепт 2.7, где описывается, как обеспечить совпадение регулярного выражения с конкретными символами Юникода. Если ваш механизм регулярных выражений поддерживает Юникод, этот способ можно также использовать для поиска непечатаемых символов.

2.3. Сопоставление с одним символом из нескольких

Задача

Создать регулярное выражение, которому соответствовали бы любые ошибочные написания слова `calendar`, чтобы иметь возможность отыскивать это слово в документе, не полагаясь на грамотность автора. Предполагается, что на месте любой гласной буквы может использоваться символ `a` или `e`. Создать второе регулярное выражение, которому соответствовала бы единственная шестнадцатеричная цифра. Создать третье регулярное выражение, которому соответствовал бы единственный символ, не являющийся шестнадцатеричной цифрой.

Цель этого рецепта – рассказать о такой важной и часто используемой конструкции, как *символьный класс*.

Решение

Ошибки в слове calendar

```
c[ae]l[ae]nd[ae]r
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Шестнадцатеричная цифра

```
[a-fA-F0-9]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Нешестнадцатеричная цифра

```
[^a-fA-F0-9]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Последовательность, заключенная в квадратные скобки, называется *символьным классом*. Символьному классу соответствует единственный символ, совпадающий с любым из перечисленных символов. В первом регулярном выражении указано три класса, каждому из которых соответствует символ `a` или `e`. Это обеспечивает свободу выбора. Когда с помощью этого регулярного выражения проверяется слово `calendar`, первому символьному классу соответствует символ `a`, второму – `e` и третьему – `a`.

Внутри символьных классов только четыре символа имеют специальное назначение: `\`, `^`, `-` и `]`. В диалектах Java и .NET открывающая квадратная скобка `[` также является метасимволом внутри символьных классов.

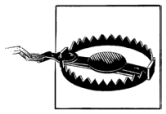
Символ обратного слэша всегда экранирует символ, следующий за ним, так же как и за пределами символьного класса. Экранированный символ может представлять единственный символ, а также начало или конец диапазона. Другие четыре метасимвола приобретают специальное назначение, только если находятся в определенной позиции. Они могут представлять литералы символов в символьных классах без дополнительного экранирования путем помещения их в позицию, где они утрачивают специальное назначение. Выражение `‹[][^-]›` иллюстрирует эту особенность. Она свойственна всем диалектам регулярных выражений, рассматриваемых в книге, кроме JavaScript, строго придерживающегося стандарта. Конструкция `‹[]›` в JavaScript всегда интерпретируется как пустой символьный класс, который никогда не находит совпадений. Однако мы рекомендуем всегда экранировать эти метасимволы; предыдущее регулярное выражение следовало бы записать так: `‹[\]\[\^\-]›`. Экранирование метасимволов упрощает понимание регулярного выражения.

Все остальные символы интерпретируются буквально и просто добавляют себя в символьный класс. Регулярному выражению `‹[$()*+.?{|]›` соответствует любой из девяти символов, перечисленных между квадратными скобками. Эти девять символов приобретают специальное значение только за пределами символьных классов. Внутри символьных классов они интерпретируются буквально. Их экранирование только ухудшит читаемость регулярного выражения.

Алфавитно-цифровые символы не должны экранироваться обратным слэшем, так как в этом случае либо может возникать ошибка, либо может создаваться специальный символ (имеющий специальное значение в регулярных выражениях). В наших пояснениях к некоторым специальным символам, например в рецепте 2.2, мы указывали, что они могут

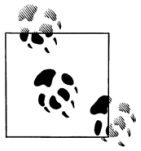
использоваться внутри символьных классов. Все эти специальные символы состоят из слэша и символа, иногда сопровождаемых рядом других символов. Так, символьному классу ‹[\r\n]› соответствует символ возврата каретки \r или перевода строки \n.

Символ крышки ^ в символьном классе означает отрицание, если он следует непосредственно за открывающей квадратной скобкой. Он обеспечивает соответствие символьному классу любых символов, которые *отсутствуют* в списке.



Во всех диалектах, обсуждаемых в этой книге, инвертированному символьному классу соответствуют символы конца строки, если они отсутствуют в инвертированном символьном классе. Не позволяйте своим регулярным выражениям по ошибке продолжать поиск в множестве строк.

Символ дефиса (-) определяет *диапазон*, когда находится между двумя символами. Диапазон в символьном классе включает в себя символ, стоящий перед дефисом, символ, стоящий после дефиса, и все символы, расположенные между ними в порядке следования их числовых кодов. Чтобы узнать, какие числовые коды соответствуют символам, необходимо обратиться к таблице символов ASCII или Юникода. Символьный класс ‹[A-z]› включает в себя все символы, расположенные в таблице ASCII между символом верхнего регистра A и символом нижнего регистра z. Этот диапазон также включает в себя некоторые знаки препинания; так, символьному классу ‹[A-Z\[\\\]\^_`a-z]› соответствуют те же самые символы, только объявлен он более явно. Мы рекомендуем создавать диапазоны, включающие только цифры или только буквы, причем только верхнего или только нижнего регистра.



Перевернутые диапазоны, такие как ‹[z-a]›, не допускаются.

Варианты

Сокращения

Шесть специальных символов, состоящих из обратного слэша и следующего за ним алфавитного символа, представляют *сокращенную форму записи символьных классов*: ‹\d›, ‹\D›, ‹\w›, ‹\W›, ‹\s› и ‹\S›. Их можно использовать как внутри, так и за пределами символьных классов. Каждой сокращенной форме записи, включающей символ нижнего регистра, соответствует сокращенная форма записи, включающая символ верхнего регистра, имеющая противоположный смысл.

Так, последовательностям ‹\d› и ‹[\d]› соответствует один цифровой символ. Последовательности ‹\D› соответствует любой символ, *не* являющийся цифрой, а ее эквивалентом является запись ‹[^\d]›.

Вот как можно использовать сокращенную форму ‹\d›, чтобы переписать регулярное выражение, совпадающее с шестнадцатеричной цифрой, представленное выше в этом рецепте:

```
[a-fA-F\d]
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

Метасимволу ‹\w› соответствует один *символ слова*. Символ слова – это символ, который может входить в состав слова. Сюда относятся алфавитные символы, цифры и символ подчеркивания. Такой набор символов может показаться странным, но он был выбран таким, потому что из этих символов обычно составляются идентификаторы в языках программирования. Метасимволу ‹\W› соответствует любой символ, который не может являться частью такого слова.

В диалектах Java (в версиях с 4 по 6), JavaScript, PCRE и Ruby метасимвол ‹\w› всегда идентичен классу ‹[a-zA-Z0-9_]›. В .NET он также включает алфавитно-цифровые символы любых других алфавитов (кириллица, тайский алфавит и прочие). В Java 7 символы других алфавитов включаются, только если установлен флаг `UNICODE_CHARACTER_CLASS`. В Python 2.x символы других алфавитов включаются, только если при создании регулярного выражения передавался флаг `UNICODE` или `U`. В Python 3.x символы других алфавитов включаются по умолчанию, но есть возможность заставить ‹\w› совпадать только с символами ASCII, если установить флаг `ASCII` или `A`. В Perl 5.14 флаг `/a` (ASCII) превращает ‹\w› в эквивалент символьного класса ‹[a-zA-Z0-9_]›, флаг `/u` (Unicode) включает все алфавиты, поддерживаемые стандартом Юникода, а флаг `/l` (locale – локаль) делает метасимвол ‹\w› зависимым от текущих национальных настроек. В версиях Perl, предшествовавших версии 5.14, а также в версии Perl 5.14 (при использовании флага `/d` (default – по умолчанию) или когда не используется ни один из флагов `/adlu`) метасимвол ‹\w› автоматически включает алфавиты Юникода, если испытуемая строка или само регулярное выражение закодировано в кодировке UTF-8, или если регулярное выражение включает коды символов со значениями выше 255, такие как ‹\x{100}›, или свойства Юникода, такие как ‹\p{L}›. В противном случае метасимвол ‹\w› соответствует только символам ASCII.

К метасимволу ‹\d› применяются те же правила во всех диалектах. В .NET цифры из других алфавитов всегда включаются в класс. В Python поведение этого метасимвола зависит от флагов `UNICODE` и `ASCII` и от версии Python – 2.x или 3.x. В Perl 5.14 его поведение зависит от флагов `/adlu`. В предыдущих версиях Perl оно зависит от кодировки испытуемой строки и регулярного выражения, а также от наличия каких-либо лексем Юникода.

Метасимволу ‹\s› соответствует любой *пробельный символ*. Сюда входят символы пробела, табуляции и символы конца строки. Метасимволу ‹\S› соответствует любой символ, не соответствующий метасимволу ‹\s›.

В .NET и JavaScript метасимволу ‹\s› также соответствуют все символы, определяемые стандартом Юникода как пробельные. В Java, Perl и Python метасимвол ‹\s› подчиняется тем же правилам, что и метасимволы ‹\w› и ‹\d›.

Примечательно, что в диалекте JavaScript для ‹\s› используется Юникод, а для ‹\d› и ‹\w› – ASCII. Еще одно противоречие несет в себе метасимвол ‹\b›. Метасимвол ‹\b› не является краткой формой представления символьного класса, а обозначает *границу слова*. Можно было бы ожидать, что ‹\b› поддерживает Юникод, когда эту кодировку поддерживает метасимвол ‹\w›, и только ASCII, когда ‹\w› поддерживает только ASCII, но это не всегда так. Подробнее об этом рассказывается в подразделе «Символы слов» в рецепте 2.6.

Поиск без учета регистра символов

```
(?i)[A-F0-9]
```

Параметры: нет
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

```
(?i)[^A-F0-9]
```

Параметры: нет
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python, Ruby

Нечувствительность к регистру символов, определяемая с помощью внешнего флага (рецепт 3.4) или с помощью модификатора режима внутри регулярного выражения (рецепт 2.1, раздел «Сопоставление без учета регистра символов»), также оказывает воздействие и на символьные классы. Только что представленные регулярные выражения эквивалентны регулярным выражениям, представленным в первоначальном решении.

Диалект JavaScript также следует этому правилу, но он не поддерживает модификатор ‹(?i)›. Чтобы отключить чувствительность регулярного выражения к регистру символов в JavaScript, необходимо установить флаг /i при его создании или использовать библиотеку XRegExp, добавляющую поддержку модификаторов режимов в начале регулярных выражений.

Особенности, характерные для разных диалектов

Разность символьных классов в .NET

```
[a-zA-Z0-9-[g-zG-Z]]
```

Параметры: нет
Диалекты: .NET 2.0 или выше

Данному регулярному выражению соответствует одна шестнадцатеричная цифра, но реализовано оно косвенным образом. Базовому символьному классу соответствует любой алфавитно-цифровой символ,

а вложенному классу, который вычитается из базового, символы от g до z. Вложенный вычитаемый класс должен находиться в конце базового класса и предваряться символом дефиса: ‹[*class*-[*subtract*]]›.

Операцию *вычитания* символьных классов удобно использовать, в частности, при работе со свойствами, блоками и алфавитами Юникода. Например, выражению ‹\p{IsThai}› соответствует любой символ тайского алфавита. Выражению ‹\P{N}› соответствует любой символ, у которого отсутствует свойство Number. Разности этих двух классов ‹[\p{IsThai}-[\P{N}]]› соответствует любая из 10 цифр тайского алфавита. Более подробно о работе со свойствами Юникода рассказывается в рецепте 2.7.

Объединение, разность и пересечение символьных классов в Java

Диалект Java допускает вложение одного символьного класса в другой. При непосредственном включении одного символьного класса в другой получается *объединение* двух классов. Допускается создавать столько уровней вложения, сколько потребуется. Выражения ‹[a-f[A-F][0-9]]› и ‹[a-f[A-F[0-9]]]› используют объединение символьных классов. Они соответствуют шестнадцатеричным цифрам, как и первоначальное выражение, но в них отсутствуют дополнительные квадратные скобки.

Регулярное выражение ‹[\w&&[a-fA-F0-9\s]]› соответствует шестнадцатеричным цифрам, используя пересечение символьных классов. Это выражение могло бы получить приз в соревновании на самое запутанное выражение. Базовому символьному классу ‹[\w]› соответствует любой символ слова. Вложенному классу ‹[a-fA-F0-9\s]› соответствует любая шестнадцатеричная цифра и любой пробельный символ. Результатом является пересечение двух классов, которому соответствуют только шестнадцатеричные цифры и ничто иное. Так как базовому классу не соответствуют пробельные символы, а вложенному классу не соответствуют символы ‹[g-zG-Z_]›, они исключаются из окончательного символьного класса и остаются только шестнадцатеричные цифры.

Выражение ‹[a-zA-Z0-9&&[^g-zG-Z]]› использует разность символьных классов. Ему соответствует одна шестнадцатеричная цифра и тоже косвенным образом. Базовому символьному классу ‹[a-zA-Z0-9]› соответствует любой алфавитно-цифровой символ. Вложенному классу ‹[^g-zG-Z]›, который вычитается из базового, – символы от g до z. Этот вложенный класс инвертируется, и ему предшествуют два символа амперсанда: ‹[*class*&&[^*subtract*]]›.

Пересечение и разность символьных классов удобно использовать, в частности, при работе со свойствами Юникода, блоками и алфавитами. Например, выражению ‹\p{IsThai}› соответствует любой символ тайского алфавита. Выражению ‹\p{N}› соответствует любой символ, у которого присутствует свойство Number. Последовательности ‹[\p{InThai}&&[\p{N}]]› соответствует любая из 10 цифр тайского алфавита.

Если вас интересуют различия между возможными комбинациями ‹\p› в регулярных выражениях, их описание вы найдете в рецепте 2.7, где также подробно описываются тонкости работы со свойствами Юникода.

См. также

Рецепт 2.2, где описывается, как организовать поиск непечатаемых символов. В рецепте 2.7 рассказывается об особенностях сопоставления с символами Юникода. Синтаксис определения непечатаемых символов и символов Юникода с успехом можно использовать в символьных классах.

В разделе «Bat, cat или rat» в рецепте 5.3 описываются некоторые наиболее частые ошибки в символьных классах, допускаемые теми, кто только начинает осваивать регулярные выражения.

2.4. Сопоставление с любым символом

Этот рецепт объясняет достоинства и недостатки метасимвола «точка».

Задача

Создать регулярное выражение, которому соответствует любой символ, заключенный в одиночные кавычки. Одному решению должен соответствовать любой одиночный символ, за исключением конца строки, заключенный в одиночные кавычки. Второму решению должен соответствовать действительно любой символ, включая символы конца строки.

Решение

Любой символ, за исключением символа конца строки

```
'.'
```

Параметры: нет (параметр «точке соответствуют границы строк» должен быть сброшен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Любой символ, включая символы конца строки

```
'.'
```

Параметры: точке соответствуют границы строк
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

```
'[\s\S]'
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Любой символ, за исключением символа конца строки

Точка – это один из старейших и простейших элементов регулярных выражений. Она означает обязательное соответствие любому одиночному символу.

Однако не всегда очевидно, что означает фраза *любой символ*. Самые ранние инструменты, обеспечивающие поддержку регулярных выражений, обрабатывали содержимое файлов строку за строкой, поэтому отсутствовала потребность искать соответствие с концом строки в испытуемом тексте. Языки программирования, рассматриваемые в этой книге, обрабатывают текст целиком, как единое целое, независимо от наличия в нем разрывов строк. Если действительно возникает потребность производить обработку текста построчно, можно дополнить программу кодом, который будет разбивать исходный текст на массив строк и затем применять регулярное выражение к каждой строке в массиве. Как это делается, показано в рецепте 3.21 в следующей главе.

Ларри Уолл (Larry Wall), создатель языка Perl, сохранил в языке Perl традиционное поведение инструментов построчной обработки текста, где точка никогда не соответствовала концу строки. Все остальные диалекты, рассматриваемые в этой книге, последовали его примеру. То есть регулярному выражению ‹.› соответствует любой одиночный символ, *за исключением* символа перевода строки.

Любой символ, включая символы конца строки

Если необходимо позволить регулярному выражению обрабатывать сразу несколько строк, следует включить параметр «точке соответствуют границы строк». Этот параметр маскируется под разными названиями. В языке Perl и во многих других он носит сбивающее с толку название «режим единственной строки» (single line mode), тогда как в Java он называется «режим ‘точка – это все’» (‘dot all’ mode). Подробное описание этого вопроса вы найдете в рецепте 3.4 в следующей главе. Независимо от того, как называется этот параметр в вашем любимом языке программирования, называйте его про себя режимом «точке соответствуют границы строк». Это все, что дает данный параметр.

Для JavaScript, где отсутствует параметр «точке соответствуют границы строк», необходимо альтернативное решение. Как описывается в рецепте 2.3, метасимволу ‹\s› соответствуют любые пробельные символы, тогда как метасимволу ‹\S› соответствуют любые другие символы, не соответствующие метасимволу ‹\s›. Объединив их, можно получить символьный класс ‹[\s\S]›, которому будут соответствовать любые символы, включая символы конца строки. Тот же эффект дают символьные классы ‹[\d\D]› и ‹[\w\W]›.

Злоупотребление точкой

Точка является элементом регулярных выражений, которым злоупотребляют наиболее часто. Выражение `‹\d\d.\d\d.\d\d›` представляет не лучший способ поиска дат. Ему прекрасно соответствует дата 05/16/08, но ему также соответствует текст 99/99/99. Хуже того, ему соответствует число 12345678.

Регулярное выражение, которому соответствуют только корректные даты, – это тема для более поздней главы (см. рецепт 4.5). Но заменить точку на более подходящий символьный класс совсем несложно. Выражение `‹\d\d[/.\-]\d\d[/.\-]\d\d›` позволяет использовать в качестве символа-разделителя слэш, точку или дефис. Данному регулярному выражению по-прежнему соответствует текст 99/99/99, но число 12345678 уже не соответствует.

Это просто совпадение, что в предыдущем примере точка включена в символьные классы. Внутри символьных классов точка интерпретируется буквально. В этом есть определенный смысл, так как в некоторых странах, например в Германии, точка используется в качестве символа-разделителя в датах.

Точку следует использовать, только если действительно допускается совпадение с любым символом. Во всех остальных случаях лучше использовать символьные классы и инвертированные символьные классы.

Варианты

Ниже показано, как обеспечить совпадение с символом в одиночных кавычках, включая границы строк, с помощью встроенного модификатора режима:

`(?s)'.'`

Параметры: нет
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python

`(?m)'.'`

Параметры: нет
Диалект: Ruby

Если не удалось включить режим «точке соответствуют границы строк» вне регулярного выражения, можно добавить модификатор режима в начало выражения. В рецепте 2.1 в подразделе «Поиск без учета регистра символов» мы объяснили концепцию модификаторов режима и говорили, что в JavaScript они не поддерживаются.

`‹(?s)›` – это модификатор режима «точке соответствуют границы строк» в .NET, Java, XRegExp, PCRE, Perl и Python. Символ `s` здесь обозначает режим «single line» (единственная строка), под которым в языке Perl подразумевается режим «точке соответствуют границы строк».

Терминология настолько запутывающая, что разработчик механизма регулярных выражений в языке Ruby допустил ошибку. Для включения режима «точке соответствуют границы строк» в Ruby используется модификатор ‹(?m)›. Здесь используется другой символ, но функциональность та же самая. Новый механизм в Ruby 1.9 продолжает использовать модификатор ‹(?m)› для включения режима «точке соответствуют границы строк». Значение модификатора ‹(?m)› в языке Perl совершенно иное; оно описывается в рецепте 2.5.

См. также

Во многих случаях в действительности требуется не совпадение с *любым* символом, а с любым символом из некоторого множества. Как это реализовать, описывается в рецепте 2.3.

Рецепт 3.4, где объясняется, как устанавливать параметры, такие как «точке соответствуют границы строк» в исходном программном коде.

При работе с текстом, содержащим символы Юникода, может оказаться предпочтительнее использовать метасимвол ‹\X›, соответствующий графеме Юникода, а не «точку», которая совпадает с кодовым пунктом Юникода. Подробнее об этом рассказывается в рецепте 2.7.

2.5. Сопоставление в начале и/или в конце строки

Задача

Создать четыре регулярных выражения. Первому выражению должно соответствовать слово alpha, но только если оно находится в самом начале испытуемого текста. Второму выражению должно соответствовать слово omega, но только если оно находится в самом конце испытуемого текста. Третьему выражению должно соответствовать слово begin, но только если оно находится в начале строки. Четвертому выражению должно соответствовать слово end, но только если оно находится в конце строки.

Решение

В начале испытуемого текста

```
^alpha
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
\Aalpha
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

В конце испытуемого текста

```
omega$
```

Параметры: нет (режим «символам ^ и $ соответствуют границы строк» должен быть выключен)
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python

```
omega\Z
```

Параметры: нет
Диалекты: .NET, Java, PCRE, Perl, Python, Ruby

В начале строки

```
^begin
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

В конце строки

```
end$
```

Параметры: символам ^ и $ соответствуют границы строк
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

Якорные метасимволы и строки

Метасимволы регулярных выражений ‹^›, ‹$›, ‹\A›, ‹\Z› и ‹\z› называются *якорями*. Они не соответствуют каким-либо символам; вместо этого они обозначают определенные позиции, фактически осуществляя привязку регулярного выражения к этим позициям.

Строка – это часть испытуемого текста, которая находится между начальной и конечной границами строки. Если в испытуемом тексте отсутствуют границы строк, то весь текст рассматривается как одна строка. Так, следующий текст состоит из четырех строк, по одной для слов `one` и `two`, затем следует пустая строка и далее строка со словом `four`:

```
one
two

four
```

В программном коде этот текст можно было бы представить так: `one`LF `two`LF LF `four`.

В начале испытуемого текста

Якорному метасимволу ‹\A› всегда соответствует точка самого начала испытуемого текста, непосредственно перед первым символом. Это единственное место в тексте, соответствующее данному метасимволу. Поместив

‹\A› в начало регулярного выражения, можно проверить, начинается ли испытуемый текст с искомого текста. Символ «A» должен быть символом верхнего регистра.

Диалект JavaScript не поддерживает метасимвол ‹\A›.

Якорный метасимвол ‹^› является эквивалентом метасимвола ‹\A› при условии, что параметр «символам ^ и $ соответствуют границы строк» отключен. Этот параметр отключен по умолчанию во всех диалектах регулярных выражений, за исключением Ruby. Ruby не предоставляет возможность отключать этот параметр.

Если вы не используете диалект JavaScript, мы рекомендуем всегда вместо метасимвола ‹^› использовать метасимвол ‹\A›. Значение метасимвола ‹\A› никогда не изменяется, что позволяет избегать путаницы и ошибок при установке параметров регулярных выражений.

В конце испытуемого текста

Якорным метасимволам ‹\Z› и ‹\z› всегда соответствует позиция в конце испытуемого текста непосредственно за последним символом. Поместив ‹\Z› или ‹\z› в конец регулярного выражения, можно проверить, оканчивается ли испытуемый текст искомым текстом.

Диалекты .NET, Java, PCRE, Perl и Ruby поддерживают оба метасимвола, ‹\Z› и ‹\z›. Диалект Python поддерживает только метасимвол ‹\Z›. Диалект JavaScript не поддерживает ни один из них.

Различия между метасимволами ‹\Z› и ‹\z› начинают проявляться, когда в испытуемом тексте последним символом является символ конца строки. В этом случае метасимволу ‹\Z› соответствует самый конец испытуемого текста, после завершающего символа конца строки, а также позиция непосредственно перед этим символом конца строки. Преимущество состоит в том, что можно выполнить поиск ‹omega\Z›, не беспокоясь об удалении завершающего символа конца строки в конце испытуемого текста. При чтении содержимого файла строка за строкой одни инструменты включают символ конца строки, другие – нет; метасимвол ‹\Z› позволяет ликвидировать это различие. Метасимволу ‹\z› соответствует только самый конец испытуемого текста, поэтому он не будет совпадать с текстом в регулярном выражении, если в нем имеется завершающий символ конца строки.

Якорный метасимвол ‹$› является эквивалентом метасимвола ‹\Z› при условии, что параметр «символам ^ и $ соответствуют границы строк» отключен. Этот параметр отключен по умолчанию во всех диалектах регулярных выражений, за исключением Ruby. Ruby не предоставляет возможность отключать этот параметр. Так же как и ‹\Z›, метасимволу ‹$› соответствует самый конец испытуемого текста, а также позиция перед завершающим символом конца строки.

Чтобы прояснить все эти тонкости и неясности, рассмотрим пример на языке Perl. Предположим, что переменная $/ (текущий символ-разде-

литель записей) имеет значение по умолчанию \n, тогда следующая инструкция на языке Perl прочитает одну строку, введенную с терминала (из потока стандартного ввода):

```
$line = <>;
```

В языке Perl символ перевода строки будет сохранен в переменной $line. Вследствие этого регулярное выражение, такое как ‹end•of•input.\z›, не совпадет с содержимым переменной. Но совпадения будут обнаружены выражениями ‹end•of•input.\Z› и ‹end•of•input.$›, потому что они игнорируют завершающий символ перевода строки.

Чтобы упростить обработку данных, программисты на языке Perl часто принудительно отбрасывают символ перевода строки инструкцией:

```
chomp $line;
```

После выполнения этой операции совпадение будет обнаружено всеми тремя выражениями. (Технически инструкция chomp удаляет из строки текущий символ-разделитель записей.)

Если вы не пользуетесь JavaScript, мы рекомендуем всегда применять метасимвол ‹\Z› вместо ‹$›. Значение метасимвола ‹\Z› никогда не изменяется, что позволяет избегать путаницы и ошибок при установке параметров регулярных выражений.

В начале строки

По умолчанию метасимволу ‹^› соответствует позиция в начале испытуемого текста, как и метасимволу ‹\A›. Только в Ruby метасимволу ‹^› всегда соответствует начало строки. Во всех остальных диалектах необходимо включать дополнительный параметр, чтобы символ крышки и знак доллара совпадали с границами строк. Этот параметр обычно называется «многострочным» (multiline) режимом.

Не следует путать этот режим с режимом «единственной строки», который более известен как режим «точке соответствуют границы строк». «Многострочный» режим оказывает влияние только на поведение символа крышки и знака доллара, а режим «единственной строки» оказывает влияние только на поведение точки, как разъяснялось в рецепте 2.4. Нет ничего невозможного в том, чтобы одновременно включить «многострочный» режим и режим «единственной строки». По умолчанию оба параметра отключены.

При корректной установке параметра метасимволу ‹^› будет соответствовать позиция в начале каждой строки испытуемого текста. Строго говоря, метасимвол будет совпадать с позицией непосредственно перед самым первым символом в файле, что происходит всегда, а также с позициями после каждого символа конца строки. Символ крышки в выражении ‹\n^› избыточен, потому что ‹^› всегда совпадает с позицией после ‹\n›.

В конце строки

По умолчанию метасимволу ‹$› соответствует позиция только в конце испытуемого текста или перед завершающим символом конца строки, как и метасимволу ‹\Z›. Только в Ruby метасимволу ‹$› всегда соответствует конец каждой строки. Во всех остальных диалектах необходимо включать дополнительный «многострочный» режим, чтобы символ крышки и знак доллара совпадали с границами строк.

При корректной установке параметра метасимволу ‹$› будет соответствовать позиция в конце каждой строки испытуемого текста. (Безусловно, он также будет совпадать с позицией за самым последним символом в тексте, потому что эта позиция всегда считается концом строки.) Символ доллара в выражении ‹$\n› избыточен, потому что ‹$› всегда совпадает с позицией перед ‹\n›.

Совпадения нулевой длины

Нет ничего необычного в том, чтобы регулярное выражение содержало только один или более якорных метасимволов. Такое регулярное выражение будет находить совпадение нулевой длины в каждой позиции, где будет обнаруживаться соответствие якорному метасимволу. При объединении нескольких якорных метасимволов все они должны совпадать с одной и той же позицией, чтобы регулярное выражение находило соответствие.

Такое регулярное выражение может использоваться в операции поиска с заменой. Замещение метасимволов ‹\A› или ‹\Z› позволяет добавлять что-либо в начало или в конец испытуемого текста. Замещение метасимволов ‹^› или ‹$› в режиме «символам ^ и $ соответствуют границы строк» позволяет добавлять что-либо в начало или в конец каждой строки испытуемого текста.

Комбинация двух якорных метасимволов позволит проверить наличие пустых строк или отсутствие ввода. Комбинации ‹\A\Z› соответствует пустая строка, а также строка, состоящая из единственного символа перевода строки. Комбинации ‹\A\z› соответствует только пустая строка. Комбинации ‹^$› в режиме «символам ^ и $ соответствуют границы строк» будет соответствовать каждая пустая строка в испытуемом тексте.

Варианты

```
(?m)^begin
```

Параметры: нет
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python

```
(?m)end$
```

Параметры: нет
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Python

Если нет возможности включить режим «символам ^ и $ соответствуют границы строк» вне регулярного выражения, в начало регулярного выражения можно поместить модификатор режима. О модификаторах режима и об отсутствии их поддержки в JavaScript рассказывалось в рецепте 2.1 в подразделе «Поиск без учета регистра символов».

‹(?m)› – это модификатор режима «символам ^ и $ соответствуют границы строк» в диалектах .NET, Java, XRegExp, PCRE, Perl и Python. Символ m соответствует названию «multiline» (многострочный) режим, под которым в языке Perl подразумевается режим «символам ^ и $ соответствуют границы строк».

Как уже говорилось выше, терминология оказалась настолько запутывающей, что разработчик механизма регулярных выражений в языке Ruby допустил ошибку. Модификатор ‹(?m)› в языке Ruby используется для включения режима «точке соответствуют границы строк». То есть в языке Ruby модификатор ‹(?m)› не оказывает влияния на поведение якорных метасимволов ‹^› и ‹$›. В Ruby метасимволам ‹^› и ‹$› всегда соответствуют начало и конец каждой строки.

За исключением путаницы в символах модификаторов, в языке Ruby выбор применения метасимволов ‹^› и ‹$› исключительно к границам строк следует признать удачным. Если вы не используете JavaScript, мы рекомендуем придерживаться такого же выбора в своих регулярных выражениях.

Ян Гойвертс следовал этой идее при разработке программ EditPad Pro и PowerGREP. В них вы не найдете флажок с надписью ^ and $ match at line breaks (символам ^ и $ соответствуют границы строк), хотя имеется флажок с надписью dot matches line breaks (точке соответствуют границы строк). Если добавить в начало регулярного выражения модификатор ‹(?-m)›, то, чтобы привязать регулярное выражение к началу или к концу файла, необходимо будет использовать метасимволы ‹\A› и ‹\Z›.

См. также

Рецепт 3.4, где объясняется, как устанавливать такие параметры, как «символам ^ и $ соответствуют границы строк», в исходном коде программ.

Рецепт 3.21, где показано, как использовать процедурный код для построчной обработки текста с помощью регулярного выражения.

2.6. Сопоставление с целыми словами

Задача

Создать регулярное выражение, которому соответствовало бы слово cat в тексте My cat is brown, но которое не находило бы соответствие в словах category или bobcat. Создать еще одно регулярное выражение, которому

соответствовала бы последовательность cat в тексте staccato, но которое не находило бы соответствие ни в одном из трех предыдущих случаев.

Решение

На границах слова

```
\bcat\b
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Не на границах слова

```
\Bcat\B
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, PCRE, Perl, Python, Ruby

Обсуждение

На границах слова

Метасимвол ‹\b› называется *границей слова*. Ему соответствует позиция начала или конца слова. Сам по себе этот метасимвол дает совпадение нулевой длины. Метасимвол ‹\b› является *якорным*, точно так же как и метасимволы, представленные в предыдущем разделе.

Строго говоря, метасимволу ‹\b› соответствуют следующие три позиции:

- Перед первым символом испытуемого текста, если первый символ является символом слова.
- После последнего символа испытуемого текста, если последний символ является символом слова.
- Между двумя символами испытуемого текста, когда один из них является символом слова, а другой – нет.

Чтобы выполнить поиск «только целого слова» с помощью регулярного выражения, нужно просто поместить искомое слово между двумя границами слов, как это предложено в решении ‹\bcat\b›. Первый метасимвол ‹\b› требует, чтобы символ ‹c› находился в самом начале строки или после символа, не являющегося символом слова. Второй метасимвол ‹\b› требует, чтобы символ ‹t› находился в самом конце строки или перед символом, не являющимся символом слова.

Символы конца строки не являются символами слова. Поэтому метасимвол ‹\b› будет совпадать с позицией после символа конца строки, если сразу же вслед за ним располагается символ слова. Он также будет совпадать с позицией перед символом конца строки, если ему непосредственно предшествует символ слова. Благодаря этому слово, занимающее всю строку, будет обнаружено в процессе поиска «только целых слов». На поведение метасимвола ‹\b› не влияет выбор «многострочного»

режима или наличие модификатора ‹(?m)›, что является одной из причин, почему в этой книге «многострочный» режим сопоставляется с режимом «символам ^ и $ соответствуют границы строк».

Ни в одном из диалектов, обсуждаемых в этой книге, нет отдельных метасимволов, которым соответствовала бы позиция только в начале или только в конце слова. Однако в них нет никакой необходимости, если только вам не потребуется регулярное выражение, не содержащее ничего, кроме границы слова. Символы перед или после метасимвола ‹\b› в регулярном выражении помогут указать, с какой позицией будет совпадать ‹\b›. Метасимвол ‹\b› в выражениях ‹\bx› и ‹!\b› мог бы совпадать только с началом слова. Метасимвол ‹\b› в выражениях ‹x\b› и ‹\b!› мог бы совпадать только с концом слова. А выражения ‹x\bx› и ‹!\b!› никогда не будут находить соответствий.

Если действительно потребуется обеспечить совпадение только в начале слова или только в конце слова, для этого можно использовать опережающую или ретроспективную проверку. Подробнее об этих проверках рассказывается в рецепте 2.16. Этот прием невозможен в диалектах JavaScript и Ruby 1.8, так как они не поддерживают ретроспективные проверки. Регулярное выражение ‹(?<!\w)(?=\w)› соответствует началу слова – оно требует, чтобы символ перед позицией совпадения не был символом слова, а символ за позицией совпадения, напротив, был символом слова. Выражение ‹(?<=\w)(?!\w)› действует с точностью до наоборот: оно соответствует концу слова, требуя, чтобы предыдущий символ был символом слова, а следующий, напротив, не был символом слова. Важно заметить, что для проверки отсутствия символа слова должна использоваться отрицательная проверка с помощью метасимвола ‹\w›, а не положительная – с помощью метасимвола ‹\W›. Выражение ‹(?<!\w)› соответствует началу сроки, так как перед началом строки отсутствует символ слова (как и любой другой символ). Но выражение ‹(?<=\W)› никогда не совпадет с началом строки. Выражение ‹(?!\w)› соответствует концу строки по той же самой причине. Поэтому две наши конструкции на основе проверок будут корректно совпадать с началом строки, если она начинается словом, и с концом строки, если она завершается словом.

Не на границах слова

Метасимволу ‹\B› соответствует любая позиция в испытуемом тексте, которая не соответствует метасимволу ‹\b›. Отсюда следует, что метасимволу ‹\B› соответствует любая позиция, не являющаяся началом или концом слова.

Строго говоря, метасимволу ‹\B› соответствуют следующие пять позиций:

- Перед первым символом испытуемого текста, если первый символ не является символом слова.
- После последнего символа испытуемого текста, если последний символ не является символом слова.

- Между двумя символами слова.
- Между двумя символами, не являющимися символами слова.
- Пустая строка.

Выражению ‹\Bcat\B› соответствует последовательность символов cat в тексте staccato, но не в текстах My cat is brown, category или bobcat.

Чтобы выполнить поиск с условием, противоположным требованию «только целое слово» (то есть чтобы совпадение обнаруживалось в словах staccato, category и bobcat, но не обнаруживалось в тексте My cat is brown), необходимо с помощью операции выбора объединить подвыражения ‹\Bcat› и ‹cat\B› в выражение ‹\Bcat|cat\B›. Подвыражению ‹\Bcat› соответствует последовательность символов cat в словах staccato и bobcat. Подвыражению ‹cat\B› соответствует последовательность символов cat в слове category (и в слове staccato, если подвыражение ‹\Bcat› не позаботилось о нем раньше). Оператор выбора рассматривается в рецепте 2.8.

Символы слов

За всеми этими разговорами о границах слов мы ничего не сказали о том, что такое *символ слова*. Символ слова – это символ, который может являться частью слова. В рецепте 2.3 в подразделе «Сокращения» говорилось о том, какие символы включены в состав символьного класса ‹\w›, который соответствует одному символу слова. К сожалению, для метасимвола ‹\b› история выглядит иначе.

Несмотря на то что все диалекты, рассматриваемые в этой книге, поддерживают метасимволы ‹\b› и ‹\B›, тем не менее они по-разному определяют, какие символы являются символами слова.

В диалектах .NET, JavaScript, PCRE, Perl, Python и Ruby метасимволу ‹\b› соответствует позиция между двумя символами, один из которых соответствует символьному классу ‹\w›, а другой – символьному классу ‹\W›. Метасимволу ‹\B› всегда соответствует позиция между двумя символами, когда оба они соответствуют либо символьному классу ‹\w›, либо символьному классу ‹\W›.

В диалектах JavaScript, PCRE и Ruby символами слов считаются только символы ASCII. Символьный класс ‹\w› в них идентичен классу ‹[a-zA-Z0-9_]›. При использовании этих диалектов можно выполнить поиск «только целого слова» в тексте, написанном на языке, где используются исключительно символы от A до Z без диакритических знаков, например на английском. Но эти диалекты не могут использоваться для поиска «только целых слов» в текстах на других языках, таких как испанский или русский.

Диалект .NET интерпретирует буквы и цифры из любых алфавитов как символы слова. В этих диалектах можно выполнить поиск «только целого слова» в текстах на любом языке, включая и те, которые не используют латинский алфавит.

В языке Python имеется возможность выбора. В Python 2.x символы, не входящие в набор ASCII, могут включаться в поиск, только если при создании регулярного выражения был передан флаг `UNICODE` или `U`. В Python 3.x символы, не входящие в набор ASCII, включаются по умолчанию, но их можно исключить, установив флаг `ASCII` или `A`. Эти флаги в равной степени воздействует на метасимволы ‹\b› и ‹\w›.

В языке Perl включение в символьный класс ‹\w› символов ASCII или Юникода, цифр и подчеркиваний зависит от версии Perl и флагов `/adlu`. Подробнее об этом рассказывается в рецепте 2.3, в подразделе «Сокращения». Во всех версиях Perl поведение метасимвола ‹\b› полностью соответствует символьному классу ‹\w›.

Диалект Java отличается некоторой непоследовательностью. В версиях Java 4–6 классу ‹\w› соответствуют только символы ASCII. В Java 7 по умолчанию классу ‹\w› соответствуют только символы ASCII, но если установить флаг `UNICODE_CHARACTER_CLASS`, ему будут соответствовать и символы Юникода. Однако метасимвол ‹\b› поддерживает Юникод во всех версиях Java и может применяться при работе с любыми алфавитами. В версиях Java 4–6 выражению ‹\b\w\b› соответствует одна буква латинского алфавита, цифра или символ подчеркивания, который не является частью слова ни на одном языке. Выражение ‹\bкошка\b› корректно совпадет со словом `кошка` в тексте на русском языке, потому что метасимвол ‹\b› поддерживает Юникод. Но выражение ‹\w+› не совпадет ни с одним русским словом в Java 4–6, потому что класс ‹\w› поддерживает только символы ASCII.

См. также

Рецепт 2.3, где описывается, какие символы соответствуют сокращенному символьному классу ‹\w›, соответствующему символу слова.

Рецепт 5.1, где демонстрируется, как использовать границы слов для поиска совпадений с целыми словами и как преодолевать проблемы, связанные с различиями поведения границ слов в разных диалектах регулярных выражений.

2.7. Кодовые пункты Юникода, категории, блоки и алфавиты

Задача

С помощью регулярного выражения отыскать символ торговой марки (™), указав в нем кодовый пункт Юникода вместо копирования и вставки фактического символа. При желании можно копировать и вставлять сам символ; в конце концов, символ торговой марки – это всего лишь обычный литерал, несмотря на то, что его нельзя ввести с клавиатуры непосредственно. Литералы обсуждались в рецепте 2.1.

Создать регулярное выражение, которому соответствовали бы любые символы из категории Юникода «Currency Symbol» (символ денежной единицы).

Создать регулярное выражение, которому соответствовали бы любые символы, принадлежащие блоку Юникода «Greek Extended».

Создать регулярное выражение, которому соответствовали бы любые символы, которые в соответствии со стандартом Юникода принадлежат греческому алфавиту.

Создать регулярное выражение, которому соответствовали бы графемы, то есть символы, состоящие из базового символа и дополнительных комбинационных знаков.

Решение

Кодовый пункт Юникода

```
\u2122
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, Python, Ruby 1.9

```
\U00002122
```

Параметры: нет
Диалекты: Python

Данные регулярные выражения будет работать в Python 2.x, только если они будут оформлены как строка Юникода: `u"\u2122"` или `u"\U00002122"`.

```
\x{2122}
```

Параметры: нет
Диалекты: Java 7, PCRE, Perl

Библиотека PCRE должна быть скомпилирована с поддержкой UTF-8; в языке PHP следует включить поддержку UTF-8 с помощью модификатора шаблона `/u`.

```
\u{2122}
```

Параметры: нет
Диалекты: Ruby 1.9

В Ruby 1.8 регулярные выражения для работы с Юникодом не поддерживаются.

Категория Юникода

```
\p{Sc}
```

Параметры: нет
Диалекты: .NET, Java, XRegExp, PCRE, Perl, Ruby 1.9

Библиотека PCRE должна быть скомпилирована с поддержкой UTF-8, в языке PHP следует включить поддержку UTF-8 с помощью модификатора шаблона /u. Диалекты JavaScript и Python не поддерживают категории Юникода. В Ruby 1.8 регулярные выражения для работы с Юникодом не поддерживаются.

Блок Юникода

```
\p{IsGreekExtended}
```

Параметры: нет
Диалекты: .NET, Perl

```
\p{InGreekExtended}
```

Параметры: нет
Диалекты: Java, XRegExp, Perl

Диалекты JavaScript, PCRE, Python и Ruby 1.9 не поддерживают блоки Юникода. Они поддерживают кодовые пункты Юникода, которые можно использовать для сопоставления с блоками, как показано в разделе «Варианты» в этом рецепте. Диалект XRegExp добавляет поддержку блоков Юникода в JavaScript.

Алфавит в Юникоде

```
\p{Greek}
```

Параметры: нет
Диалекты: XRegExp, PCRE, Perl, Ruby 1.9

```
\p{IsGreek}
```

Параметры: нет
Диалекты: Java 7, Perl

Для работы с алфавитами необходима библиотека PCRE версии 6.5 или выше, а кроме того, библиотека должна быть скомпилирована с поддержкой UTF-8. В языке PHP следует включить поддержку UTF-8 с помощью модификатора шаблона /u. Диалекты .NET, JavaScript и Python не поддерживают алфавиты Юникода. Диалект XRegExp добавляет поддержку алфавитов Юникода в JavaScript. В Ruby 1.8 регулярные выражения для работы с Юникодом не поддерживаются.

Графема в Юникоде

```
\X
```

Параметры: нет
Диалекты: PCRE, Perl

В диалектах PCRE и Perl имеется специальный метасимвол, соответствующий графемам. Библиотека PCRE должна быть скомпилирована с поддержкой UTF-8. В языке PHP следует включить поддержку UTF-8 с помощью модификатора шаблона /u.

`(?>\P{M}\p{M}*)`

Параметры: нет
Диалекты: .NET, Java, Ruby 1.9

`(?:\P{M}\p{M}*)`

Параметры: нет
Диалекты: XRegExp

Диалекты .NET, Java, XRegExp и Ruby 1.9 не имеют метасимвола, соответствующего графемам. Но они поддерживают категории Юникода, посредством которых можно имитировать сопоставление с графемами.

Диалекты JavaScript (без XRegExp) и Python не поддерживают графемы Юникода. В Ruby 1.8 регулярные выражения для работы с Юникодом не поддерживаются.

Обсуждение

Кодовый пункт Юникода

Кодовый пункт – это одна запись в базе данных символов Юникода. Кодовый пункт – это не *символ*; впрочем, это зависит от того, какой смысл вкладывается в слово «символ». То, что на экране выглядит как символ, в Юникоде называется *графемой*.

Кодовый пункт Юникода U+2122 представляет символ «знака торговой марки». Его можно отыскать с помощью конструкции ‹\u2122›, ‹\u{2122}› или ‹\x{2122}› в зависимости от используемого диалекта регулярных выражений.

При использовании метасимвола ‹\u› необходимо указывать точно четыре шестнадцатеричные цифры. Это означает, что его можно использовать для поиска кодовых пунктов Юникода в диапазоне от U+0000 до U+FFFF.

В конструкции ‹\u{…}› и ‹\x{…}› допускается указывать в фигурных скобках от одной до шести шестнадцатеричных цифр, что обеспечивает поддержку всех кодовых пунктов в диапазоне от U+000000 до U+10FFFF. Кодовый пункт U+00E0 будет соответствовать как выражению ‹\x{E0}›, так и выражению ‹\x{00E0}›. Кодовые пункты U+100000 и выше используются очень редко и в настоящее время весьма слабо поддерживаются шрифтами и операционными системами.

Механизм регулярных выражений в Python не поддерживает кодовые пункты Юникода. Литералы строк Юникода в Python 2.x и литералы текстовых строк в Python 3.x позволяют вставлять в них экранированные последовательности, соответствующие кодовым пунктам Юникода. Последовательности от \u0000 до \uFFFF представляют кодовые пункты Юникода от U+0000 до U+FFFF. Последовательности от \U00000000 до \U0010FFFF представляют все кодовые пункты Юникода. После метасимвола \U необходимо указывать точно восемь шестнадцатеричных цифр, даже для кодовых пунктов Юникода ниже U+10FFFF.

При составлении регулярных выражений в виде строковых литералов в программе на языке Python допускается использовать экранированные последовательности ‹\u2122› и ‹\U00002122› непосредственно в регулярных выражениях. Если регулярные выражения извлекаются из файла или вводятся пользователем, эти экранированные последовательности, обозначающие кодовые пункты Юникода, не будут работать, если передать прочитанную из файла или принятую от пользователя строку непосредственно в вызов `re.compile()`. В Python 2.x экранированные последовательности Юникода можно декодировать вызовом `string.decode('unicode-escape')`. В Python 3.x можно вызвать `string.encode('utf-8').decode('unicode-escape')`.

Кодовые пункты могут использоваться внутри символьных классов и за их пределами.

Категории Юникода

Каждый кодовый пункт Юникода принадлежит единственной *категории Юникода*. Всего существует 30 категорий Юникода, определяемых одно- или двубуквенными идентификаторами. Эти категории сгруппированы в 7 суперкатегорий, определяемых однобуквенными идентификаторами:

‹\p{L}› Любая буква любого языка.

‹\p{Ll}› Любая строчная буква, для которой имеется прописной вариант.

‹\p{Lu}› Прописная буква, для которой имеется строчный вариант.

‹\p{Lt}› Начальная буква слова при условии, что капитализируется только первая буква слова.

‹\p{Lm}› Специальный символ, который используется как буква.

‹\p{Lo}› Символ или идеограмма, для которого отсутствуют варианты нижнего и верхнего регистра.

‹\p{M}› Символ, который должен объединяться с другим символом (диакритические знаки, умляуты, описывающие рамки и пр.).

‹\p{Mn}› Символ, который должен объединяться с другим символом, не требующий дополнительного пространства (например, диакритические знаки, умляуты и пр.).

‹\p{Mc}› Символ, который должен объединяться с другим символом, требующий дополнительного пространства (например, огласовки во многих восточных алфавитах).

‹\p{Me}› Символ, описанный вокруг другого символа (окружность, квадрат, клавишный колпачок и пр.).

‹\p{Z}› Любые типы пробельных символов или невидимых символов-разделителей.

‹\p{Zs}› Невидимые пробельные символы, занимающие дополнительное пространство.

`\p{Zl}` Символ-разделитель строк U+2028.

`\p{Zp}` Символ-разделитель абзацев U+2029.

`\p{S}` Математические символы, знаки денежных единиц, декоративные элементы, символы для рисования рамок и пр.

`\p{Sm}` Любые математические символы.

`\p{Sc}` Любые знаки денежных единиц.

`\p{Sk}` Комбинационный знак (метка) как самостоятельный символ.

`\p{So}` Различные символы, которые не являются математическими знаками, знаками денежных единиц или комбинационными символами.

`\p{N}` Различные числовые символы в любом алфавите.

`\p{Nd}` Цифры от 0 до 9 в любом алфавите, за исключением идеографических алфавитов.

`\p{Nl}` Числа, которые выглядят как буквы, например римские цифры.

`\p{No}` Надстрочные или подстрочные цифры или числа, не являющиеся цифрами 0...9 (кроме чисел из идеографических алфавитов).

`\p{P}` Различные знаки пунктуации.

`\p{Pd}` Различные тире и дефисы.

`\p{Ps}` Различные открывающие скобки.

`\p{Pe}` Различные закрывающие скобки.

`\p{Pi}` Различные открывающие кавычки.

`\p{Pf}` Различные закрывающие кавычки.

`\p{Pc}` Знаки пунктуации, например подчеркивание, соединяющее слова.

`\p{Po}` Различные знаки пунктуации, не являющиеся дефисами, тире, скобками, кавычками или символами, соединяющими слова.

`\p{C}` Неотображаемые управляющие символы и неиспользуемые кодовые пункты.

`\p{Cc}` Управляющие символы ASCII или Latin-1 в диапазонах 0x00...0x1F и 0x7F...0x9F.

`\p{Cf}` Неотображаемые символы, являющиеся признаками форматирования.

`\p{Co}` Кодовые пункты, предназначенные для закрытого применения.

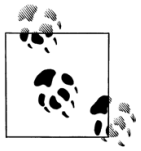
`\p{Cs}` Одна половина суррогатной пары в кодировке UTF-16.

`\p{Cn}` Кодовые пункты, которым не присвоены символы.

Комбинация `\p{Ll}` совпадает с одиночным кодовым пунктом из категории `Ll`, или «lowercase letter» (символ нижнего регистра). Комбина-

ция ‹\p{L}› является сокращенной формой записи символьного класса ‹[\p{Ll}\p{Lu}\p{Lt}\p{Lm}\p{Lo}]›, которому соответствует любой кодовый пункт из категории «letter» (буква).

Метасимвол ‹\P› – это инвертированная версия метасимвола ‹\p›. Комбинации ‹\P{Ll}› соответствует одиночный кодовый пункт, не относящийся к категории Ll. Комбинации ‹\P{L}› соответствует единственный кодовый пункт, не относящийся к суперкатегории «letter» (буква). Эта комбинация не является аналогом символьного класса ‹[\P{Ll}\P{Lu}\P{Lt}\P{Lm}\P{Lo}]›, которому соответствует любой кодовый пункт. Комбинации ‹\P{Ll}› соответствуют кодовые пункты из категории Lu (или любым другим категориям, за исключением Ll), тогда как комбинация ‹\P{Lu}› будет совпадать и с кодовыми пунктами из категории Ll. Объединение только этих двух комбинаций в символьный класс уже обеспечивает совпадение с любыми возможными кодовыми пунктами.



В Perl, а также в библиотеке PCRE версии 6.5 и выше комбинацию ‹\p{L&}› можно использовать как сокращение для выражения ‹[\p{Ll}\p{Lu}\p{Lt}]›, которому соответствуют все буквы из всех алфавитов, имеющие варианты верхнего и нижнего регистров.

Блок Юникода

Все кодовые пункты в базе данных символов Юникода подразделяются на блоки. Каждый содержит отдельный диапазон кодовых пунктов. Кодовые пункты с U+0000 по U+FFFF делятся на 156 кодовых блоков в версии 6.1 стандарта Юникода:

```
‹U+0000...U+007F \p{InBasicLatin}›
‹U+0080...U+00FF \p{InLatin-1Supplement}›
‹U+0100...U+017F \p{InLatinExtended-A}›
‹U+0180...U+024F \p{InLatinExtended-B}›
‹U+0250...U+02AF \p{InIPAExtensions}›
‹U+02B0...U+02FF \p{InSpacingModifierLetters}›
‹U+0300...U+036F \p{InCombiningDiacriticalMarks}›
‹U+0370...U+03FF \p{InGreekandCoptic}›
‹U+0400...U+04FF \p{InCyrillic}›
‹U+0500...U+052F \p{InCyrillicSupplement}›
‹U+0530...U+058F \p{InArmenian}›
‹U+0590...U+05FF \p{InHebrew}›
‹U+0600...U+06FF \p{InArabic}›
‹U+0700...U+074F \p{InSyriac}›
‹U+0750...U+077F \p{InArabicSupplement}›
‹U+0780...U+07BF \p{InThaana}›
‹U+07C0...U+07FF \p{InNKo}›
‹U+0800...U+083F \p{InSamaritan}›
‹U+0840...U+085F \p{InMandaic}›
‹U+08A0...U+08FF \p{InArabicExtended-A}›
‹U+0900...U+097F \p{InDevanagari}›
‹U+0980...U+09FF \p{InBengali}›
```

```
‹U+0A00...U+0A7F \p{InGurmukhi}›
‹U+0A80...U+0AFF \p{InGujarati}›
‹U+0B00...U+0B7F \p{InOriya}›
‹U+0B80...U+0BFF \p{InTamil}›
‹U+0C00...U+0C7F \p{InTelugu}›
‹U+0C80...U+0CFF \p{InKannada}›
‹U+0D00...U+0D7F \p{InMalayalam}›
‹U+0D80...U+0DFF \p{InSinhala}›
‹U+0E00...U+0E7F \p{InThai}›
‹U+0E80...U+0EFF \p{InLao}›
‹U+0F00...U+0FFF \p{InTibetan}›
‹U+1000...U+109F \p{InMyanmar}›
‹U+10A0...U+10FF \p{InGeorgian}›
‹U+1100...U+11FF \p{InHangulJamo}›
‹U+1200...U+137F \p{InEthiopic}›
‹U+1380...U+139F \p{InEthiopicSupplement}›
‹U+13A0...U+13FF \p{InCherokee}›
‹U+1400...U+167F \p{InUnifiedCanadianAboriginalSyllabics}›
‹U+1680...U+169F \p{InOgham}›
‹U+16A0...U+16FF \p{InRunic}›
‹U+1700...U+171F \p{InTagalog}›
‹U+1720...U+173F \p{InHanunoo}›
‹U+1740...U+175F \p{InBuhid}›
‹U+1760...U+177F \p{InTagbanwa}›
‹U+1780...U+17FF \p{InKhmer}›
‹U+1800...U+18AF \p{InMongolian}›
‹U+18B0...U+18FF \p{InUnifiedCanadianAboriginalSyllabicsExtended}›
‹U+1900...U+194F \p{InLimbu}›
‹U+1950...U+197F \p{InTaiLe}›
‹U+1980...U+19DF \p{InNewTaiLue}›
‹U+19E0...U+19FF \p{InKhmerSymbols}›
‹U+1A00...U+1A1F \p{InBuginese}›
‹U+1A20...U+1AAF \p{InTaiTham}›
‹U+1B00...U+1B7F \p{InBalinese}›
‹U+1B80...U+1BBF \p{InSundanese}›
‹U+1BC0...U+1BFF \p{InBatak}›
‹U+1C00...U+1C4F \p{InLepcha}›
‹U+1C50...U+1C7F \p{InOlChiki}›
‹U+1CC0...U+1CCF \p{InSundaneseSupplement}›
‹U+1CD0...U+1CFF \p{InVedicExtensions}›
‹U+1D00...U+1D7F \p{InPhoneticExtensions}›
‹U+1D80...U+1DBF \p{InPhoneticExtensionsSupplement}›
‹U+1DC0...U+1DFF \p{InCombiningDiacriticalMarksSupplement}›
‹U+1E00...U+1EFF \p{InLatinExtendedAdditional}›
‹U+1F00...U+1FFF \p{InGreekExtended}›
‹U+2000...U+206F \p{InGeneralPunctuation}›
‹U+2070...U+209F \p{InSuperscriptsandSubscripts}›
‹U+20A0...U+20CF \p{InCurrencySymbols}›
‹U+20D0...U+20FF \p{InCombiningDiacriticalMarksforSymbols}›
‹U+2100...U+214F \p{InLetterlikeSymbols}›
‹U+2150...U+218F \p{InNumberForms}›
```

```
‹U+2190...U+21FF \p{InArrows}›
‹U+2200...U+22FF \p{InMathematicalOperators}›
‹U+2300...U+23FF \p{InMiscellaneousTechnical}›
‹U+2400...U+243F \p{InControlPictures}›
‹U+2440...U+245F \p{InOpticalCharacterRecognition}›
‹U+2460...U+24FF \p{InEnclosedAlphanumerics}›
‹U+2500...U+257F \p{InBoxDrawing}›
‹U+2580...U+259F \p{InBlockElements}›
‹U+25A0...U+25FF \p{InGeometricShapes}›
‹U+2600...U+26FF \p{InMiscellaneousSymbols}›
‹U+2700...U+27BF \p{InDingbats}›
‹U+27C0...U+27EF \p{InMiscellaneousMathematicalSymbols-A}›
‹U+27F0...U+27FF \p{InSupplementalArrows-A}›
‹U+2800...U+28FF \p{InBraillePatterns}›
‹U+2900...U+297F \p{InSupplementalArrows-B}›
‹U+2980...U+29FF \p{InMiscellaneousMathematicalSymbols-B}›
‹U+2A00...U+2AFF \p{InSupplementalMathematicalOperators}›
‹U+2B00...U+2BFF \p{InMiscellaneousSymbolsandArrows}›
‹U+2C00...U+2C5F \p{InGlagolitic}›
‹U+2C60...U+2C7F \p{InLatinExtended-C}›
‹U+2C80...U+2CFF \p{InCoptic}›
‹U+2D00...U+2D2F \p{InGeorgianSupplement}›
‹U+2D30...U+2D7F \p{InTifinagh}›
‹U+2D80...U+2DDF \p{InEthiopicExtended}›
‹U+2DE0...U+2DFF \p{InCyrillicExtended-A}›
‹U+2E00...U+2E7F \p{InSupplementalPunctuation}›
‹U+2E80...U+2EFF \p{InCJKRadicalsSupplement}›
‹U+2F00...U+2FDF \p{InKangxiRadicals}›
‹U+2FF0...U+2FFF \p{InIdeographicDescriptionCharacters}›
‹U+3000...U+303F \p{InCJKSymbolsandPunctuation}›
‹U+3040...U+309F \p{InHiragana}›
‹U+30A0...U+30FF \p{InKatakana}›
‹U+3100...U+312F \p{InBopomofo}›
‹U+3130...U+318F \p{InHangulCompatibilityJamo}›
‹U+3190...U+319F \p{InKanbun}›
‹U+31A0...U+31BF \p{InBopomofoExtended}›
‹U+31C0...U+31EF \p{InCJKStrokes}›
‹U+31F0...U+31FF \p{InKatakanaPhoneticExtensions}›
‹U+3200...U+32FF \p{InEnclosedCJKLettersandMonths}›
‹U+3300...U+33FF \p{InCJKCompatibility}›
‹U+3400...U+4DBF \p{InCJKUnifiedIdeographsExtensionA}›
‹U+4DC0...U+4DFF \p{InYijingHexagramSymbols}›
‹U+4E00...U+9FFF \p{InCJKUnifiedIdeographs}›
‹U+A000...U+A48F \p{InYiSyllables}›
‹U+A490...U+A4CF \p{InYiRadicals}›
‹U+A4D0...U+A4FF \p{InLisu}›
‹U+A500...U+A63F \p{InVai}›
‹U+A640...U+A69F \p{InCyrillicExtended-B}›
‹U+A6A0...U+A6FF \p{InBamum}›
‹U+A700...U+A71F \p{InModifierToneLetters}›
‹U+A720...U+A7FF \p{InLatinExtended-D}›
```

```
‹U+A800...U+A82F \p{InSylotiNagri}›
‹U+A830...U+A83F \p{InCommonIndicNumberForms}›
‹U+A840...U+A87F \p{InPhags-pa}›
‹U+A880...U+A8DF \p{InSaurashtra}›
‹U+A8E0...U+A8FF \p{InDevanagariExtended}›
‹U+A900...U+A92F \p{InKayahLi}›
‹U+A930...U+A95F \p{InRejang}›
‹U+A960...U+A97F \p{InHangulJamoExtended-A}›
‹U+A980...U+A9DF \p{InJavanese}›
‹U+AA00...U+AA5F \p{InCham}›
‹U+AA60...U+AA7F \p{InMyanmarExtended-A}›
‹U+AA80...U+AADF \p{InTaiViet}›
‹U+AAE0...U+AAFF \p{InMeeteiMayekExtensions}›
‹U+AB00...U+AB2F \p{InEthiopicExtended-A}›
‹U+ABC0...U+ABFF \p{InMeeteiMayek}›
‹U+AC00...U+D7AF \p{InHangulSyllables}›
‹U+D7B0...U+D7FF \p{InHangulJamoExtended-B}›
‹U+D800...U+DB7F \p{InHighSurrogates}›
‹U+DB80...U+DBFF \p{InHighPrivateUseSurrogates}›
‹U+DC00...U+DFFF \p{InLowSurrogates}›
‹U+E000...U+F8FF \p{InPrivateUseArea}›
‹U+F900...U+FAFF \p{InCJKCompatibilityIdeographs}›
‹U+FB00...U+FB4F \p{InAlphabeticPresentationForms}›
‹U+FB50...U+FDFF \p{InArabicPresentationForms-A}›
‹U+FE00...U+FE0F \p{InVariationSelectors}›
‹U+FE10...U+FE1F \p{InVerticalForms}›
‹U+FE20...U+FE2F \p{InCombiningHalfMarks}›
‹U+FE30...U+FE4F \p{InCJKCompatibilityForms}›
‹U+FE50...U+FE6F \p{InSmallFormVariants}›
‹U+FE70...U+FEFF \p{InArabicPresentationForms-B}›
‹U+FF00...U+FFEF \p{InHalfwidthandFullwidthForms}›
‹U+FFF0...U+FFFF \p{InSpecials}›
```

Блок Юникода – это единый и неразрывный диапазон кодовых пунктов. Хотя многим блокам присвоены имена алфавитов и категорий Юникода, тем не менее здесь нет 100% совпадения. Название блока лишь указывает на его основное назначение.

Блок `Currency` **не включает в себя знаки доллара и йены. По исторически сложившимся причинам они находятся в блоках** `BasicLatin` **и** `Latin-1 Supplement`**. Но при этом оба знака попадают в категорию** `Currency Symbol`**. Поэтому для поиска любых знаков денежных единиц вместо комбинации** `‹\p{InCurrency}›` **следует использовать** `‹\p{Sc}›`**.**

Большинство блоков включают в себя кодовые пункты с неприсвоенными символами, которые попадают в категорию `‹\p{Cn}›`**. Никакие другие категории Юникода и никакой алфавит не включают кодовые пункты с неприсвоенными символами.**

Комбинация `‹\p{InBlockName}›` **может использоваться в диалектах .NET, XRegExp и Perl, а комбинацию** `‹\p{IsBlockName}›` **надо использовать в диалекте Java.**

Диалект Perl поддерживает вариант `Is`, но лучше использовать синтаксис `In`, чтобы избежать путаницы. Для работы с алфавитами в Perl поддерживаются комбинации `‹\p{Script}›` и `‹\p{IsScript}›`, но не `‹\p{InScript}›`.

В стандарте Юникода оговаривается, что имена блоков должны быть нечувствительны к регистру символов и все различия, касающиеся пробелов, дефисов или символов подчеркивания, должны игнорироваться. К сожалению, большинство диалектов регулярных выражений не отличаются такой гибкостью. Все версии .NET и Java 4 требуют записи имен блоков в «верблюжьей» нотации, как показано в списке выше. Версии Perl 5.8 и выше, а также Java 5 и выше допускают смешивать регистр символов. Диалекты Perl, Java и .NET поддерживают также нотацию с дефисами и без пробелов, использованную в списке выше. Мы рекомендуем использовать эту нотацию. Из диалектов, обсуждаемых в этой книге, только XRegExp и Perl 5.12 (и выше) полностью соответствуют требованиям стандарта Юникода в отношении пробелов, дефисов и символов подчеркивания в именах блоков Юникода.

Алфавит Юникода

Каждый кодовый пункт, за исключением кодовых пунктов с неприсвоенными символами, является частью точно одного алфавита Юникода. Кодовые пункты с неприсвоенными символами не являются частью ни одного алфавита. Кодовые пункты со значениями до U+FFFF и с присвоенными символами принадлежат следующим 72 алфавитам, определенным в версии 6.1 стандарта Юникода:

```
‹\p{Common}›                ‹\p{Lepcha}›
‹\p{Arabic}›                ‹\p{Limbu}›
‹\p{Armenian}›              ‹\p{Lisu}›
‹\p{Balinese}›              ‹\p{Malayalam}›
‹\p{Bamum}›                 ‹\p{Mandaic}›
‹\p{Batak}›                 ‹\p{Meetei_Mayek}›
‹\p{Bengali}›               ‹\p{Mongolian}›
‹\p{Bopomofo}›              ‹\p{Myanmar}›
‹\p{Braille}›               ‹\p{New_Tai_Lue}›
‹\p{Buginese}›              ‹\p{Nko}›
‹\p{Buhid}›                 ‹\p{Ogham}›
‹\p{Canadian_Aboriginal}›   ‹\p{Ol_Chiki}›
‹\p{Cham}›                  ‹\p{Oriya}›
‹\p{Cherokee}›              ‹\p{Phags_Pa}›
‹\p{Coptic}›                ‹\p{Rejang}›
‹\p{Cyrillic}›              ‹\p{Runic}›
‹\p{Devanagari}›            ‹\p{Samaritan}›
‹\p{Ethiopic}›              ‹\p{Saurashtra}›
‹\p{Georgian}›              ‹\p{Sinhala}›
‹\p{Glagolitic}›            ‹\p{Sundanese}›
‹\p{Greek}›                 ‹\p{Syloti_Nagri}›
‹\p{Gujarati}›              ‹\p{Syriac}›
‹\p{Gurmukhi}›              ‹\p{Tagalog}›
```

```
‹\p{Han}›                    ‹\p{Tagbanwa}›
‹\p{Hangul}›                 ‹\p{Tai_Le}›
‹\p{Hanunoo}›                ‹\p{Tai_Tham}›
‹\p{Hebrew}›                 ‹\p{Tai_Viet}›
‹\p{Hiragana}›               ‹\p{Tamil}›
‹\p{Inherited}›              ‹\p{Telugu}›
‹\p{Javanese}›               ‹\p{Thaana}›
‹\p{Kannada}›                ‹\p{Thai}›
‹\p{Katakana}›               ‹\p{Tibetan}›
‹\p{Kayah_Li}›               ‹\p{Tifinagh}›
‹\p{Khmer}›                  ‹\p{Vai}›
‹\p{Lao}›                    ‹\p{Yi}›
‹\p{Latin}›
```

Алфавит – это группа кодовых пунктов, используемых в различных системах письменности. Одни алфавиты, такие как `Thai`, применяются только в одном языке человеческого общения. Другие, такие как `Latin`, используются в нескольких языках. В некоторых языках применяется несколько алфавитов. Например, в Юникоде нет алфавита Japanese, вместо этого предлагаются алфавиты `Hiragana`, `Katakana`, `Han` и `Latin`, символы из которых обычно используются при составлении документов на японском языке.

Самым первым в списке, в нарушение алфавитного порядка следования, указан алфавит `Common`. Этот алфавит содержит все символы, общие для широкого диапазона алфавитов, такие как знаки препинания, пробел и различные вспомогательные символы.

Диалект Java требует, чтобы имена алфавитов предварялись префиксом `Is`, например `‹\p{IsYi}›`. В Perl префикс `Is` допускается, но не требуется. Диалекты XRegExp, PCRE и Ruby не допускают использование префикса `Is`.

В стандарте Юникода оговаривается, что имена алфавитов должны быть нечувствительны к регистру символов и все различия, касающиеся пробелов, дефисов или символов подчеркивания, должны игнорироваться. К сожалению, большинство диалектов регулярных выражений не отличаются такой гибкостью. Форма записи имен алфавитов в «верблюжьей» нотации и с символами подчеркивания между словами поддерживается всеми диалектами, рассматриваемыми в этой книге, которые поддерживают алфавиты Юникода.

Графема Юникода

Различия между кодовыми пунктами и символами становятся заметны, когда в игру вступают *комбинированные знаки*. Кодовому пункту U+0061 соответствует «строчная буква a латинского алфавита», а кодовому пункту U+00E0 – «строчная буква a латинского алфавита с диакритическим знаком». В комбинации они представляют то, что большинство людей назвали бы символом.

Кодовый пункт U+0300 – это диакритический знак. Он может использоваться только после буквы. Строка, состоящая из кодовых пунктов U+0061 U+0300, будет отображаться как символ à, так же как и кодовый пункт U+00E0. Комбинационный знак U+0300 отображается над символом U+0061.

Причина появления двух разных способов отображения символа с диакритическим знаком состоит в том, что уже давно существует множество кодировок символов, в которых символы с диакритическими знаками присутствуют в виде одиночных символов. Разработчики Юникода сочли, что будет полезно сохранить точное отображение устаревших наборов символов в дополнение к способу, когда диакритические знаки отделяются от базовых символов, который позволяет создавать произвольные комбинации, не поддерживаемые устаревшими кодировками.

Для вас как для пользователя регулярных выражений это особенно важно, так как все диалекты регулярных выражений, рассматриваемые в этой книге, оперируют кодовыми пунктами, а не графическими символами. Когда мы говорим, что ‹.› соответствует единственному символу, в действительности подразумевается соответствие единственному кодовому пункту. Если испытуемый текст будет состоять из двух кодовых пунктов U+0061 U+0300, который в языках программирования, таких как Java, может быть представлен строковым литералом "\u0061\u0300", точке будет соответствовать только кодовый пункт U+0061, или символ a без диакритического знака U+0300. А обоим кодовым пунктам будет соответствовать регулярное выражение ‹..›.

Диалекты Perl и PCRE предлагают специальный метасимвол ‹\X›, которому соответствует любая одиночная графема Юникода. По сути, это Юникод-версия метасимвола ‹.›. Метасимвол ‹\X› обнаружит два совпадения в тексте àà независимо от того, как он будет закодирован. Если он будет закодирован как \u00E0\u0061\u0300, первое совпадение будет соответствовать символу \u00E0, а второе – последовательности \u0061\u0300. Точка соответствует единственному кодовому пункту Юникода, и поэтому она найдет в такой строке три совпадения: \u00E0, \u0061 и \u0300.

Правила, определяющие, какие именно комбинации кодовых пунктов Юникода считаются графемами, очень сложны.[1] Вообще говоря, для поиска графем требуется обеспечить сопоставление с произвольным символом, не являющимся комбинационным знаком, и с произвольным комбинационным знаком, возможно, следующим за ним. Такое сопоставление можно обеспечить с помощью регулярного выражения ‹(?>\P{M}\p{M}*)›, допустимого во всех диалектах, поддерживающих Юникод, но не поддерживающих метасимвол ‹\X›. Комбинация ‹\P{M}› соот-

[1] Все эти тонкости описываются в приложении к стандарту «Unicode Standard Annex #29» по адресу *http://www.unicode.org/reports/tr29/*. Дополнительные практические рекомендации по работе с графемами Юникода можно найти в разделе «Графемы и нормализация», в главе 6 книги «Программирование на Perl, 4-е издание».

ветствует любому символу, не попадающему в категорию Mark. Комбинация ‹\p{M}*› соответствует всем следующим далее комбинационным знакам, если таковые имеются.

Мы заключили эти два метасимвола в атомарную группу, чтобы исключить возвраты при сопоставлении с ‹\p{M}*›, если следующий кодовый пункт не окажется комбинационным знаком. Выражение ‹\X{2}.› не совпадет со строкой àà, потому что после совпадения двух букв с диакритическими знаками с конструкцией ‹\X{2}› для точки ничего не останется. По той же причине не совпадет со строкой àà и выражение ‹(?>\P{M}\p{M}*){2}.›. Но выражение ‹(?:\P{M}\p{M}*){2}.› с несохраняющей группой совпадет со строкой àà, если она будет закодирована как \u00E0\u0061\u0300. Во второй итерации группы ‹\p{M}*› совпадет с \u0300. Затем сопоставление с точкой потерпит неудачу. Это вызовет возврат, что вынудит ‹\p{M}*› вернуть свое совпадение, после чего точка совпадет с \u0300.

Механизм регулярных выражений в JavaScript не поддерживает атомарную группировку. Эту особенность нельзя было добавить и в XRegExp, потому что при фактическом выполнении сопоставления XRegExp все еще опирается на механизм регулярных выражений JavaScript. Поэтому при использовании XRegExp ближайшей имитацией метасимвола ‹\X› является выражение ‹(?:\P{M}\p{M}*)›. Из-за отсутствия поддержки атомарной группировки вам придется постоянно помнить, что ‹\p{M}*› может вызвать возврат, если за ‹(?:\P{M}\p{M}*)› в регулярном выражении следует конструкция, которая может совпасть с символом из категории Mark.

Варианты

Инвертированный вариант

Метасимвол ‹\P› является обратным для метасимвола ‹\p›. Например, комбинации ‹\P{Sc}› соответствует любой символ, не имеющий свойства Юникода «Currency Symbol». Метасимвол ‹\P› поддерживается всеми диалектами, поддерживающими метасимвол ‹\p›, и применим для всех поддерживаемых категорий, блоков и алфавитов.

Символьные классы

Все диалекты, поддерживающие метасимволы ‹\u›, ‹\x›, ‹\p› и ‹\P›, допускают их использование в символьных классах. В этом случае в класс добавляется символ, представленный кодовым пунктом, или символы, принадлежащие категории, блоку или алфавиту. Например, следующему регулярному выражению будет соответствовать символ, являющийся либо открывающей кавычкой, либо закрывающей кавычкой, либо символом знака торговой марки (U+2122) :

```
[\p{Pi}\p{Pf}\u2122]
```

Параметры: нет
Диалекты: .NET, Java, XRegExp, Ruby 1.9

```
[\p{Pi}\p{Pf}\x{2122}]
```

Параметры: нет
Диалекты: Java 7, PCRE, Perl

Перечисление всех символов

Если диалект регулярных выражений не поддерживает категории Юникода, блоки или алфавиты, можно просто перечислить символы, составляющие требуемую категорию, блок или алфавит в виде символьного класса. Для блоков это выполняется очень просто: каждый блок – это просто непрерывный диапазон, заключенный между двумя кодовыми пунктами. Например, блок Greek Extended включает символы от U+1F00 до U+1FFF:

```
[\u1F00-\u1FFF]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, Python, Ruby 1.9

```
[\x{1F00}-\x{1FFF}]
```

Параметры: нет
Диалекты: Java 7, PCRE, Perl

Для большинства категорий и многих алфавитов эквивалентные символьные классы будут представлять длинные списки из отдельных кодовых пунктов и коротких диапазонов. Символы, входящие в любую из категорий и во многие алфавиты, разбросаны по всей таблице Юникода. Ниже приводится аналог алфавита Greek:

```
[\u0370-\u0373\u0375-\u0377\u037A-\u037D\u0384\u0386\u0388-\u038A↵
\u038C\u038E-\u03A1\u03A3-\u03E1\u03F0-\u03FF\u1D26-\u1D2A\u1D5D-\u1D61↵
\u1D66-\u1D6A\u1DBF\u1F00-\u1F15\u1F18-\u1F1D\u1F20-\u1F45\u1F48-\u1F4D↵
\u1F50-\u1F57\u1F59\u1F5B\u1F5D\u1F5F-\u1F7D\u1F80-\u1FB4\u1FB6-\u1FC4↵
\u1FC6-\u1FD3\u1FD6-\u1FDB\u1FDD-\u1FEF\u1FF2-\u1FF4\u1FF6-\u1FFE\u2126↵
\U00010140-\U0001018A\U0001D200-\U0001D245]
```

Параметры: нет
Диалект: Python

Мы создали это регулярное выражение с помощью веб-приложения UnicodeSet, находящегося по адресу *http://unicode.org/cldr/utility/list-unicodeset.jsp*. Мы ввели `\p{Greek}` в поле ввода Input (Ввод), отметили флажки Abbreviate (Сократить) и Escape (Экранировать) и щелкнули на кнопке Show Set (Показать множество).

Только Python поддерживает этот синтаксис определения кодовых пунктов Юникода, как описывалось выше в разделе «Кодовый пункт Юникода». Чтобы это регулярное выражение можно было использовать в других диалектах, необходимо внести некоторые изменения.

Регулярное выражение будет работать во многих других диалектах, если удалить из него кодовые пункты выше U+FFFF:

```
[\u0370-\u0373\u0375-\u0377\u037A-\u037D\u0384\u0386\u0388-\u038A↵
\u038C\u038E-\u03A1\u03A3-\u03E1\u03F0-\u03FF\u1D26-\u1D2A\u1D5D-\u1D61↵
\u1D66-\u1D6A\u1DBF\u1F00-\u1F15\u1F18-\u1F1D\u1F20-\u1F45\u1F48-\u1F4D↵
\u1F50-\u1F57\u1F59\u1F5B\u1F5D\u1F5F-\u1F7D\u1F80-\u1FB4\u1FB6-\u1FC4↵
\u1FC6-\u1FD3\u1FD6-\u1FDB\u1FDD-\u1FEF\u1FF2-\u1FF4\u1FF6-\u1FFE\u2126]
```

Параметры: нет
Диалекты: .NET, Java, JavaScript, Python, Ruby 1.9

В диалектах Perl и PCRE используется иной синтаксис определения кодовых пунктов Юникода. В оригинальном регулярном выражении следует заменить ‹\uFFFF› на ‹\x{FFFF}› и ‹\U0010FFFF› на ‹\x{10FFFF}›. Измененное регулярное выражение может использоваться также в Java 7.

```
[\x{0370}-\x{0373}\x{0375}-\x{0377}\x{037A}-\x{037D}\x{0384}\x{0386}↵
\x{0388}-\x{038A}\x{038C}\x{038E}-\x{03A1}\x{03A3}-\x{03E1}↵
\x{03F0}-\x{03FF}\x{1D26}-\x{1D2A}\x{1D5D}-\x{1D61}\x{1D66}-\x{1D6A}↵
\x{1DBF}\x{1F00}-\x{1F15}\x{1F18}-\x{1F1D}\x{1F20}-\x{1F45}↵
\x{1F48}-\x{1F4D}\x{1F50}-\x{1F57}\x{1F59}\x{1F5B}\x{1F5D}\x{1F5F}-↵
\x{1F7D}\x{1F80}-\x{1FB4}\x{1FB6}-\x{1FC4}\x{1FC6}-\x{1FD3}\x{1FD6}-↵
\x{1FDB}\x{1FDD}-\x{1FEF}\x{1FF2}-\x{1FF4}\x{1FF6}-\x{1FFE}\x{2126}↵
\x{10140}-\x{10178}\x{10179}-\x{10189}\x{1018A}\x{1D200}-\x{1D245}]
```

Параметры: нет
Диалекты: Java 7, PCRE, Perl

См. также

http://www.unicode.org – официальный веб-сайт консорциума Unicode Consortium, где можно загрузить любые официальные документы, касающиеся стандарта Юникод, таблицы символов и пр.

Юникод – это обширная тема, которой посвящены целые книги. Одна из таких книг называется «Unicode Explained», автор Джукка Корпела (Jukka K. Korpella) (O'Reilly).

Мы не можем в одном разделе описать все, что необходимо знать о кодовых пунктах Юникода, свойствах, блоках и алфавитах. Мы даже не пытаемся объяснить, зачем вам это нужно – просто вам это нужно. Комфортная простота расширенной таблицы ASCII – необитаемое место в современном глобализованном мире.

В разделах «Ограничение возможности ввода алфавитно-цифровыми символами (для любого языка)» (в рецепте 4.8) и «Ограничение числа слов» (в рецепте 4.9) описывается решение некоторых возникающих на практике задач с применением категорий Юникода.