



付録 D 正規表現リファレンス

- ・ PCRE 正規表現の詳細
- ・ バックスラッシュ
- ・ ハット記号およびドル記号
- ・ 終止符(ピリオド、ドット)
- ・ 単一バイトのマッチング
- ・ 角括弧および文字クラス
- ・ POSIX 文字クラス
- ・ 縦線
- ・ 内部オプション設定
- ・ サブパターン
- ・ 名前指定のサブパターン
- ・ 繰り返し
- ・ アトミック グルーピングおよび独占的量指定子
- ・ 後方参照
- ・ アサーション
- ・ 条件付きサブパターン
- ・ コメント
- ・ 再帰パターン
- ・ サブルーチンとしてのサブパターン
- ・ コールアウト

PCRE 正規表現の詳細

PCRE でサポートされている正規表現の構文および意味は、次のとおりです。正規表現は、Perl のマニュアルや多くの書籍でも説明されていて、その中には例が豊富に示されているものもあります。Jeffrey Friedl 著『Mastering Regular Expressions』（O'Reilly 発行）では、正規表現が詳細に説明されています。以下に記載された PCRE の正規表現に関する説明は、リファレンス用です。

PCRE の本来の処理対象は、1 バイト文字からなる文字列でした。ただし、現在は UTF-8 文字列もサポートされています。UTF-8 文字列を使用するには、UTF-8 サポートが組み込まれるように PCRE を構築してから、PCRE_UTF8 オプションを指定して `pcre_compile()` を呼び出す必要があります。この場合のパターン マッチングへの影響については、後述します。UTF-8 機能の概要は、メイン PCRE ページの UTF-8 サポートの項にも記載されています。

正規表現は、対象文字列に対して左から右にマッチするパターンです。通常、パターン内の文字はその文字自体を表し、対象文字列内の対応する文字とマッチします。一般的な例では、パターン

```
The quick brown fox
```

は、対象文字列内にある、このパターンと同じ部分とマッチします。正規表現の威力は、パターン内に選択肢および繰り返しを含めることができる点にあります。選択肢や繰り返しは、メタ文字を使用してパターン内で符号化されます。メタ文字はその文字自体を表さず、特殊な方法で解釈されます。

メタ文字には 2 種類あります。1 つは、角括弧内を除く、パターン内の任意の場所で認識される文字です。もう 1 つは、角括弧内で認識される文字です。角括弧外で使用するメタ文字は、次のとおりです。

¥	多様な用途を持つ一般的なエスケープ文字
^	文字列（複数行モードの場合は行）の開始をアサート
\$	文字列（複数行の場合は行）の終了をアサート
.	改行を除く任意の文字とマッチ（デフォルト）
[文字クラス定義の開始
	選択肢の開始
(サブパターンの開始
)	サブパターンの終了
?	(の意味を拡張 あるいは、0 または 1 回のマッチ あるいは、最小回数のマッチ
*	0 回以上の繰り返し
+	1 回以上の繰り返し あるいは、「独占的量子指定子」
{	最小/最大を指定する量子指定子の開始

パターン内の角括弧で囲まれた部分は「文字クラス」といいます。文字クラス内で使用するメタ文字は、以下に限定されます。

¥	一般的なエスケープ文字
^	クラスを否定（ただし、最初の文字に使用した場合に限る）
-	文字範囲の指定
[POSIX 文字クラス（POSIX 構文が後続する場合に限る）
]	文字クラスの終了

ここでは、各メタ文字の使用法について説明します。

バックスラッシュ

バックスラッシュ文字には、さまざまな使用法があります。最初の使用法は、英数字以外の文字が後続する場合です。この場合は、後続文字を持つ特殊な意味はなくなります。バックスラッシュをエスケープ文字として使用するこの方法は、文字クラスの内でも外でも適用されます。

たとえば、* 文字とマッチさせる場合は、パターン内で ¥* と記述します。このエスケープアクションは、後続文字が本来メタ文字として解釈されるかどうかに関係なく適用されるため、英数字以外の文字を使用する場合は、その前に常にバックスラッシュを付加して、この文字が本来の意味を表すように指定すると安全です。特に、バックスラッシュとマッチさせる場合は、¥¥ と記述します。

PCRE_EXTENDED オプションを使用してパターンをコンパイルする場合、パターン内のスペース文字（文字クラス外）、および文字クラス外の # と次の改行文字の間にある文字は無視されます。エスケープ バックスラッシュを使用すると、パターンにスペース文字または # 文字を含めることができます。

文字シーケンスから特殊な意味を削除する場合は、このシーケンスを $\$Q$ と $\$E$ で囲みます。この機能は Perl と異なります。PCRE では $\$$ および $@$ は $\$Q\dots\E シーケンス内でリテラルとして処理されますが、Perl では変数を補間します。次の例に注意してください。

パターン	PCRE マッチング	Perl マッチング
$\$Qabc\$xyz\$E$	abc\$xyz	abc のあとに \$xyz の内容が続く
$\$Qabc\$xyz\$E$	abc\$xyz	abc\$xyz
$\$Qabc\$E\$Qxyz\E	abc\$xyz	abc\$xyz

$\$Q\dots\E シーケンスは文字クラスの内でも外でも認識されます。

非表示文字

バックスラッシュの 2 番目の使用方法では、非表示文字を目に見える方法でパターン内に符号化します。パターンを終了させるバイナリ ゼロを除けば、非表示文字の使用に制限はありません。ただし、パターンをテキスト編集で作成している場合は、次に示すいずれかのエスケープシーケンスを使用する方が、このシーケンスを表すバイナリ文字を使用する方法よりも簡単です。

$\$a$	アラーム、つまり BEL 文字 (16 進 07)
$\$cx$	「control-x」(x は任意の文字)
$\$e$	エスケープ (16 進 1B)
$\$f$	改ページ (16 進 0C)
$\$n$	改行 (16 進 0A)
$\$r$	復帰 (16 進 0D)
$\$t$	タブ (16 進 09)
$\$ddd$	8 進コード ddd の文字、または後方参照
$\$xhh$	16 進コード hh の文字
$\$x\{hhh.\}$	16 進コード hhh... の文字 (UTF-8 モード専用)

$\$cx$ の正確な効果は、次のとおりです。x が小文字の場合は、大文字に変換されます。さらに、文字のビット 6 (16 進 40) が反転します。したがって、 $\$cz$ は 16 進数の 1A になり、 $\$c\{$ は 3B に、 $\$c;$ は 7B になります。

$\$x$ のあとでは、0 ~ 2 個の 16 進数が読み込まれます (大文字または小文字)。UTF-8 モードの場合は、 $\$x\{$ と $\}$ の間に任意の個数の 16 進数を挿入できます。ただし、文字コードの値は 2^{31} 未満になる必要があります (16 進数の最大値は 7FFFFFFF)。 $\$x\{$ および $\}$ の間に 16 進数以外の文字が挿入されている場合、または $\}$ で終了していない場合は、このエスケープ形式が認識されません。代わりに、最初の $\$x$ が後続数字を持たない、基本的な 16 進エスケープとして解釈され、値がゼロの文字が設定されます。

PCRE が UTF-8 モードの場合、値が 256 未満の文字列は $\$x$ の 2 つの構文のいずれかで定義できます。文字列の処理方法に違いはありません。たとえば、 $\$xdc$ は $\$x\{dc\}$ とまったく同じです。

$\$0$ のあとの 8 進数は、2 個まで読み取られます。いずれの場合も、2 桁未満の場合は、指定された値が使用されます。したがって、シーケンス $\$0\$x\$07$ は 2 つのバイナリ ゼロと、後続の BEL 文字 (コード値 7) を指定します。後続のパターン文字が 8 進数の場合は、最初のゼロのあとに 2 つの数字を指定してください。

バックスラッシュの後に 0 以外の数字が後続する場合は、処理が複雑です。文字クラスの外では、PCRE はバックスラッシュを読み取り、さらに後続数字を 10 進数として読み取ります。後続数字が 10 未満の場合、または正規表現内のこの前にあるキャプチャ用左括弧の個数がこの値以上である場合は、シーケンス全体が後方参照となります。この機能の説明は、括弧付きサブパターンの説明のあとに示します。

文字クラス内、または 10 進数が 9 より大きくて、キャプチャサブパターン数がこの値未満である場合、PCRE はバックスラッシュのあとに続く 8 進数を 3 桁まで再度読み取って、値の最下位 8 ビットから単一バイトを生成します。後続の数字はすべて本来の文字を表します。たとえば、

$\$040$	スペースのもう 1 つの表記法
$\$40$	上記と同じ (これより前のキャプチャサブパターン数が 40 未満の場合)
$\$7$	常に後方参照
$\$11$	後方参照、またはもう 1 つのタブの表記法
$\$011$	常にタブ
$\$0113$	タブのあとに文字「3」が続いたもの
$\$113$	後方参照、または 8 進コードが 113 の文字
$\$377$	後方参照、または全体が 1 ビットで構成されるバイト
$\$81$	後方参照、または 2 つの文字

「8」および「1」が後続するバイナリ ゼロ

3桁を超える 8 進数値は読み取られないため、100 以上の 8 進数値には先行ゼロを付加しないでください。

シングルバイト値または単一の UTF-8 文字 (UTF-8 モードの場合) を定義するすべてのシーケンスは、文字クラスの内でも外でも使用できます。また、文字クラス内では、シーケンス `¥b` はバックスペース文字 (16 進 08) として、シーケンス `¥X` は文字「X」として解釈されます。文字クラス外では、これらのシーケンスはさまざまな意味を持ちます (「Unicode 文字のプロパティ」を参照)。

汎用文字タイプ

バックスラッシュの 3 番目の使用方法では、汎用文字タイプを指定します。常に次のように認識されます。

<code>¥d</code>	任意の 10 進数
<code>¥D</code>	10 進数以外の任意の文字
<code>¥s</code>	任意のスペース文字
<code>¥S</code>	スペース文字以外の任意の文字
<code>¥w</code>	任意の「ワード」文字
<code>¥W</code>	任意の「非ワード」文字

エスケープシーケンスはペアを形成し、各ペアによって文字セットは 2 つに分割されます。指定された任意の文字は、各ペアのいずれか 1 方のみとマッチします。

これらの文字タイプシーケンスは文字クラスの内でも外でも指定できます。これらはそれぞれ、該当するタイプの 1 文字とマッチします。現在のマッチング位置が対象ストリングの末尾である場合は、マッチする文字が存在しないため、マッチングは常に失敗します。

Perl との互換性を保つために、`¥s` は VT (垂直タブ) 文字 (コード 11) とマッチしません。このため、`¥s` は POSIX の「スペース」クラスと異なります。`¥s` 文字は HT (水平タブ) (9)、LF (改行) (10)、FF (改ページ) (12)、CR (復帰) (13)、およびスペース (32) です。

「ワード」文字は下線、または 256 未満の任意の文字 (文字または数字) です。文字および数字の定義は、PCRE の低値文字テーブルで制御されます。ロケール固有のマッチングを実行する場合は、定義が変化することがあります (`pcreapi` ページの「ロケールのサポート」を参照)。たとえば、「fr_FR」(フランス語) ロケールでは、128 よりも大きい文字コードの一部はアクセント文字に使用され、`¥w` とマッチします。

UTF-8 モードの場合、値が 128 よりも大きい文字は `¥d`、`¥s`、または `¥w` とマッチしないで、必ず `¥D`、`¥S`、および `¥W` とマッチします。Unicode 文字プロパティサポートを使用できる場合も、このようになります。

Unicode 文字のプロパティ

PCRE が Unicode 文字プロパティサポートを使用して構築されている場合に、UTF-8 モードが選択されていれば、汎用文字タイプとマッチする 3 つの追加エスケープシーケンスを使用できます。これらのエスケープシーケンスは次のとおりです。

<code>¥p{xx}</code>	<code>xx</code> プロパティを持つ文字
<code>¥P{xx}</code>	<code>xx</code> プロパティを持たない文字
<code>¥X</code>	拡張 Unicode シーケンス

上記の `xx` で表されたプロパティ名は、Unicode の一般カテゴリプロパティに限定されます。各文字は、このようなプロパティを 1 つ持ち、プロパティは 2 文字の短縮形で指定されます。Perl との互換性を保つために否定を指定するには、左波括弧とプロパティ名の間にハット記号を追加します。たとえば、`¥p{^Lu}` は `¥P{Lu}` と同じです。

`¥p` または `¥P` で指定された文字が 1 つのみの場合は、この文字で開始するすべてのプロパティが指定されます。この場合、否定を指定しなければ、エスケープシーケンス内の波括弧は省略できます。次に示す 2 つの例は効果が同じです。

```
¥p[L]
¥pL
```

次のプロパティコードがサポートされています。

<code>C</code>	その他
<code>Cc</code>	制御文字
<code>Cf</code>	フォーマット文字
<code>Cn</code>	割り当てられていない文字
<code>Co</code>	プライベート文字
<code>Cs</code>	代用文字

L	文字
LI	小文字
Lm	修飾文字
Lo	その他の文字
Lt	タイトル文字
Lu	大文字
M	マーク
Mc	スペース記号
Me	囲み記号
Mn	非スペース記号
N	数字
Nd	10 進数
NI	数値を表す文字
No	その他の数字
P	句読記号
Pc	連結用句読記号
Pd	ダッシュ
Pe	閉じ句読記号
Pf	末尾句読記号
Pi	先頭句読記号
Po	その他の句読記号
Ps	開き句読記号
S	記号
Sc	通貨記号
Sk	修飾記号
Sm	数学記号
So	その他の記号
Z	セパレータ
Zl	行セパレータ
Zp	段落セパレータ
Zs	スペース文字

「Greek」や「InMusicalSymbols」などの拡張プロパティは、PCRE ではサポートされていません。

大文字と小文字を区別しないマッチングを指定しても、これらのエスケープ シーケンスには影響しません。たとえば、`¥p{Lu}` は常に大文字にのみマッチします。

¥X エスケープは、拡張 Unicode シーケンスを構成する任意の個数の Unicode 文字とマッチします。¥X は次のシーケンスと同等です。

`(?>¥PM¥pM*)`

このシーケンスは、「マーク」プロパティを持つ 0 個以上の文字があとに続く「マーク」プロパティを持たない文字とマッチし、これらをアトミック グループ(以下を参照)として処理します。「マーク」プロパティを持つ文字は、アクセント記号など、直前の文字に影響するものです。

Unicode プロパティによる文字マッチングは低速です。PCRE は 15,000 以上の文字データを含む構造を検索する必要があるためです。このため、PCRE では `¥d` や `¥w` など従来のエスケープ シーケンスに Unicode プロパティを使用しません。

単純なアサーション

バックスラッシュの 4 番目の使用法では、単純なアサーションを作成します。アサーションは、マッチ内の特定の位置で満たさなければならない条件を指定し、対象ストリング内の文字を消費しません。サブパターンを使用した、より複雑なアサーションについては、あとで説明します。バックスラッシュを使用したアサーションは、次のとおりです。

¥b	ワード境界でマッチ
¥B	ワード境界以外でマッチ
¥A	対象ストリングの先頭でマッチ
¥Z	対象ストリングの末尾、または終端の改行前でマッチ
¥z	対象ストリングの末尾でマッチ
¥G	対象ストリングの最初のマッチング位置でマッチ

これらのアサーションは文字クラスでは使用できません(ただし、文字クラスでは ¥b はバックスペース文字という別の意味を持つことに注意してください)。

ワード境界は、対象ストリングにおいて、現在の文字および直前の文字が ¥w または ¥W と同時に一致しない(つまり一方が ¥w にマッチし、もう一方が ¥W にマッチする)位置、またはストリングの先頭または末尾の位置(先頭文字や最終文字がそれぞれ ¥w とマッチする場合)です。

¥A、¥Z、および ¥z のアサーションは、従来のハット記号やドル記号(次の項を参照)と異なり、オプション設定に関係なく、対象ストリングの先頭および末尾でのみマッチします。したがって、これらのアサーションは複数行モードから独立しています。これら 3 つのアサーションは、ハット記号およびドル記号のメタ文字の動作にのみ影響する PCRE_NOTBOL または PCRE_NOTEOL オプションの影響を受けません。ただし、`pcre_exec()` の `startoffset` 引数がゼロでなく、対象ストリングの先頭以外の位置でマッチングが開始する場合は、¥A はマッチできません。¥Z と ¥z の違いは、¥Z はストリングの最終文字である改行の前およびストリングの末尾でマッチするのに対して、¥z は末尾でのみマッチすることです。

¥G のアサーションが true になるのは、現在のマッチング位置が、`pcre_exec()` の `startoffset` 引数で指定されたマッチの開始位置である場合のみです。`startoffset` の値がゼロでない場合は、¥A と異なります。適切な引数を指定して `pcre_exec()` を何度も呼び出すことにより、Perl の /g オプションと同様の機能を実行できます。¥G が役立つのは、このように実行した場合です。

ただし、現在のマッチの先頭として定義されている PCRE での ¥G の解釈は、直前のマッチの末尾として定義されている Perl の解釈と微妙に異なります。Perl では、直前のマッチ ストリングが空の場合、現在のマッチの先頭と、直前のマッチの末尾が異なることがあります。PCRE は一度に 1 つのマッチしか実行しないため、この動作を再現することはできません。

パターンのすべての選択枝が ¥G で開始する場合、正規表現はマッチング開始位置に固定され、コンパイルされた正規表現に「固定」フラグが設定されます。

ハット記号およびドル記号

デフォルト マッチング モードで、文字クラスの外にハット記号を配置した場合、この記号は、現在のマッチング位置が対象ストリングの先頭であるときのみ true になるアサーションです。`pcre_exec()` の `startoffset` 引数がゼロでない場合、PCRE_MULTILINE オプションが設定されていないければ、ハット記号はマッチしません。文字クラス内では、ハット記号はまったく別の意味になります(「角括弧および文字クラス」および「POSIX 文字クラス」を参照)。

複数の選択枝を呼び出す場合に、パターンの先頭文字をハット記号にする必要はありません。ただし、パターンが選択枝にマッチする場合、ハット記号が必要となる選択枝ごとに先頭をハット記号にする必要があります。有効なすべての選択枝がハット記号で始まる場合、つまりパターンが対象ストリングの先頭でのみマッチするように制限されている場合は、「固定」パターンといえます(ほかに、パターンを固定パターンにする要素はあります)。

ドル記号は、(デフォルトで)現在のマッチング位置が対象ストリングの末尾、あるいはストリング内の最終文字である改行文字の直前である場合のみ true となるアサーションです。複数の選択枝を呼び出す場合に、パターンの最終文字をドル記号にする必要はありません。ただし、ドル記号が必要となる選択枝では、末尾をドル記号にする必要があります。文字クラス内では、ドル記号に特殊な意味はありません。

コンパイル時に PCRE_DOLLAR_ENDONLY オプションを設定することで、対象ストリングの末尾でのみマッチするように、ドル記号の意味を変更できます。このオプション設定は、¥Z アサーションに影響しません。

PCRE_MULTILINE オプションを設定すると、ハット記号およびドル記号の意味は変更されます。この場合、ハット記号およびドル記号は、対象ストリングの先頭と末尾でマッチするだけでなく、内部の改行文字の直前および直後でもそれぞれマッチします。たとえば、パターン `^abc$` は、複数行モードでは対象ストリング「`defnabc`」とマッチします(¥n は改行を表します)が、その他のモードではマッチしません。したがって、単一行モードで固定されたパターンは、すべての選択枝が ^ で開始するために複数行モードでは固定されません。`pcre_exec()` の `startoffset` 引数がゼロでない場合は、ハット記号をマッチできます。PCRE_MULTILINE が設定されている場合、PCRE_DOLLAR_ENDONLY オプションは無視されます。

どちらのモードでも、シーケンス ¥A、¥Z、および ¥z を使用して、対象ストリングの先頭および末尾をマッチできます。また、パターン内のすべての選択枝が ¥A で開始する場合は、PCRE_MULTILINE が設定されているかどうかに関係なく、パターンは常に固定になります。

終止符(ピリオド、ドット)

パターン内の文字クラスの外にあるドットは、対象文字列内の任意の 1 文字とマッチします。マッチする文字に非表示文字は含まれますが、(デフォルトでは)改行は含まれません。UTF-8 モードでは、ドットは任意の UTF-8 文字とマッチします。長さが 1 バイトを超えても構いませんが、(デフォルトでは)改行は含まれません。PCRE_DOTALL オプションを設定した場合、ドットは改行にもマッチします。ドットの処理は、ハット記号およびドル記号の処理から完全に独立しています。唯一の関係は、両方とも改行文字が含まれることです。文字クラス内では、ドットに特殊な意味はありません。

単一バイトのマッチング

UTF-8 モードであるかどうかに関係なく、文字クラスの外では、エスケープシーケンス `¥C` は任意の 1 バイトとマッチします。ドットと異なり、`¥C` は改行とマッチングできます。UTF-8 モードで個別のバイトにマッチするために、Perl にこの機能が組み込まれています。UTF-8 文字が各バイトに分割されるため、対象文字列内の残りの文字が不正な UTF-8 スtring となることがあります。このため、`¥C` エスケープシーケンスは使用しないことを推奨します。

UTF-8 モードの場合に `¥C` を後方検索アサーションで(以下を参照)使用すると、後方検索の長さを計算できなくなるため、PCRE では `¥C` を後方検索アサーションを使用できません。

角括弧および文字クラス

文字クラスは左角括弧で開始し、右角括弧で終了します。右角括弧自体は特殊な意味を持ちません。クラスのメンバーとして右角括弧が必要な場合は、クラス内の最初のデータ文字(最初のハット記号がある場合は、そのあと)にするか、またはバックスラッシュでエスケープする必要があります。

文字クラスは対象文字列内の単一文字とマッチします。UTF-8 モードでは、文字の占有バイト数が 1 バイトを超えることがあります。マッチする文字は、クラスで定義された文字セット内の文字でなければなりません。ただし、クラス定義内の先頭文字がハット記号の場合は、クラスで定義された文字セット内の文字とマッチできません。クラスのメンバーとして実際にハット記号が必要な場合は、先頭文字に使用しないか、またはバックスラッシュでエスケープする必要があります。

たとえば、文字クラス `[aeiou]` は任意の小文字の母音とマッチしますが、`[^aeiou]` は小文字の母音以外の任意の文字とマッチします。ハット記号は、クラスに含まれない文字を列挙することで、クラスに含まれる文字を指定する場合に便利な表記法です。ハット記号で開始するクラスは、アサーションではありません。このようなクラスは対象文字列の文字を消費するため、現在位置が文字列の末尾である場合は、マッチに失敗します。

UTF-8 モードで、値が 255 を超える文字をクラスに含めるには、複数のバイトからなるリテラル文字列として含めるか、または `¥x{` エスケープメカニズムを使用します。

大文字と小文字を区別しないマッチングを設定した場合、クラス内の任意の文字は大文字と小文字の両方を表します。したがって、たとえば、`[aeiou]` は「A」および「a」とマッチします。また、大文字と小文字を区別しない場合、`[^aeiou]` は「A」とマッチしませんが、大文字と小文字を区別する場合は「A」とマッチします。PCRE が UTF-8 モードで動作している場合に、値が 128 を超える文字で大文字と小文字の概念がサポートされるのは、Unicode プロパティ サポートを使用してコンパイルしたときのみです。

PCRE_DOTALL または PCRE_MULTILINE オプションの設定に関係なく、文字クラス内では、改行文字は特殊な方法で処理されません。`[^a]` などのクラスは、常に改行とマッチします。

マイナス(ハイフン)文字を使用すると、文字クラス内の文字範囲を指定できます。たとえば、`[d-m]` は d と m を含む、この範囲内のすべての文字とマッチします。クラス内にマイナス文字が必要な場合は、バックスラッシュでエスケープするか、範囲を示すものとして解釈されることがない位置で(通常はクラスの先頭や末尾の文字として)使用する必要があります。

リテラル文字「`]`」を範囲の終了文字として使用することはできません。`[W-]46]` などのパターンは、2 つの文字(「W」および「-」)を含むクラスのあとに、リテラル文字「46」が続くと解釈されるため、「W46」や「-46」とマッチします。ただし、「`]`」がバックスラッシュでエスケープされている場合は範囲の末尾として解釈されるため、`[W-¥]46]` は範囲と、そのあとに続くその他の 2 つの文字で構成されたクラスとして解釈されます。範囲の末尾には、「`]`」の 8 進または 16 進表記も使用できます。

文字範囲には、文字の値の照合順序が使用されます。`¥000-¥037]` など、数値で指定された文字も範囲に使用できます。UTF-8 モードでは、`¥x{100}-¥x{2ff}]` など、値が 255 を超える文字を範囲に含めることができます。

大文字小文字を区別しないマッチングが設定されている場合に、文字を含む範囲を使用すると、大文字と小文字の両方の文字とマッチします。たとえば、`[W-c]` は `[[]¥¥^_`wxyzabc]` と同じであり、大文字と小文字を区別しないでマッチします。UTF-8 以外のモードで、「`fr_FR`」ロケールの文字テーブルを使用する場合、`¥xc8-¥xcb]` は大文字と小文字の両方のアクセント記号付き E とマッチします。PCRE が UTF-8 モードで動作している場合に、値が 128 を超える文字で大文字と小文字の概念がサポートされるのは、Unicode プロパティ サポートを使用してコンパイルしたときのみです。

文字タイプ `¥d`、`¥D`、`¥p`、`¥P`、`¥s`、`¥S`、`¥w`、および `¥W` は文字クラス内で使用し、マッチする文字をクラスに追加することもできます。たとえば、`¥d[ABCDEF]` は任意の 16 進数とマッチします。ハット記号を大文字の文字タイプと併用することで、マッチする小文字の

文字タイプよりも限定的な文字セットを簡単に指定することができます。たとえば、`[^W_]` は任意の文字または数字とマッチしますが、下線とはマッチしません。

文字クラス内で認識されるメタ文字はバックスラッシュ、ハイフン(範囲の指定として解釈できる場合のみ)、ハット記号(先頭のみ)、左角括弧 (POSIX クラス名の開始として解釈できる場合のみ、次の項を参照)、および最後の右角括弧のみです。ただし、その他の英数字以外の文字をエスケープしても問題はありません。

POSIX 文字クラス

Perl は文字クラスの POSIX 表記をサポートします。この表記法では、角括弧内の名前を `[: および :]` で囲んで使用します。PCRE でもこの表記法がサポートされています。たとえば、

```
[01[:alpha:]]%
```

は「0」、「1」、任意の英数字、または「%」とマッチします。サポートされているクラス名は、次のとおりです。

alnum	文字および数字
alpha	文字
ascii	0 ~ 127 の文字コード
blank	スペースまたはタブのみ
cntrl	制御文字
digit	10 進数 (¥d と同じ)
graph	スペース以外の表示文字
lower	小文字
print	スペースを含む表示文字
punct	文字と数字を除く表示文字
space	スペース文字 (¥s と異なる)
upper	大文字
word	「ワード」文字 (¥w と同じ)
xdigit	16 進数

「スペース」文字は HT(9)、LF(10)、VT(11)、FF(12)、CR(13)、およびスペース(32)です。このリストには、VT 文字(コード 11)が含まれます。これにより、(Perl との互換性のため)VT が含まない ¥s と「スペース」が区別されます。

名前「ワード」は Perl の拡張、「ブランク」は Perl 5.8 の GNU 拡張です。もう 1 つの Perl 拡張は、コロンのあとの ^ 文字で示される否定です。たとえば、

```
[12[:^digit:]]
```

は「1」、「2」、または数字以外の任意の文字とマッチします。PCRE (および Perl) は POSIX 構文 `[.ch.]` および `[=ch=]` も認識します (「ch」は「照合要素」)。ただし、これらはサポートされていないため、これらを検出するとエラーが生成されます。

UTF-8 モードの場合、値が 128 を超える文字は POSIX のどの文字クラスともマッチしません。

縦線

縦線は選択パターンを区切る場合に使用します。たとえば、次のパターン

```
gilbert|sullivan
```

は「gilbert」または「sullivan」とマッチします。選択肢の個数に制限はなく、また、空の選択肢の使用も可能です (空のストリングとマッチ)。マッチング プロセスでは各選択肢を左から右に順に試行して、マッチした最初の選択肢を使用します。選択肢がサブパターン (以下で定義) に含まれる場合は、サブパターン内の選択肢だけでなく、メイン パターンの残りともマッチしたときに「成功」となります。

内部オプション設定

PCRE_CASELESS、PCRE_MULTILINE、PCRE_DOTALL、および PCRE_EXTENDED オプションの設定は、Perl オプション文字を「(?)と」で囲んで、パターン内から変更できます。オプション文字は次のとおりです。

```
i PCRE_CASELESS
m PCRE_MULTILINE
s PCRE_DOTALL
x PCRE_EXTENDED
```

たとえば、(?im) は大文字と小文字を区別しない、複数行マッチングを設定します。オプション文字の前にハイフンを付加することで、これらのオプションの設定を解除できます。(?im-sx) のように設定と解除を組み合わせて、PCRE_CASELESS および PCRE_MULTILINE を設定し、PCRE_DOTALL および PCRE_EXTENDED を設定解除することもできます。文字がハイフンの後に指定された場合は、オプションが設定解除されます。

オプション変更がトップレベル(サブパターンの括弧外)で指定された場合、変更は後続パターンの残りの部分に適用されます。オプション変更がパターンの先頭で指定された場合、PCRE はこの変更を取り出してグローバル オプションに設定します(したがって、`pcre_fullinfo()` 関数で抽出されたデータ内に表示されます)。

サブパターン内で指定したオプション変更が作用するのは、現在のパターンの後続部分のみです。したがって、

```
(a(?i)b)c
```

は abc および aBc とマッチし、その他のストリングとはマッチしません(PCRE_CASELESS が使用されていない場合)。この方法により、パターンの部分ごとにオプション設定を変更できます。特定の選択肢内の変更は、同じサブパターン内の後続の選択肢にも適用されます。たとえば、

```
(a(?i)b|c)
```

は「ab」、「aB」、「c」、および「C」とマッチします。「C」とマッチングする場合、最初の選択肢は放棄されていますが、オプション設定は有効になっています。これは、オプション設定の効果がコンパイル時に生じるためです。さもないと、非常に奇妙な動作が発生することがあります。

PCRE 固有のオプション PCRE_UNGREEDY および PCRE_EXTRA を Perl 互換オプションと同じ方法で変更するには、文字 U および X をそれぞれ使用します。(?X) フラグ設定は特殊であり、トップレベルにおいても、ほかの追加機能を有効にする前にパターン内で指定する必要があります。このフラグは最初に指定することを推奨します。

サブパターン

サブパターンは丸括弧で区切られた、ネスト可能なパターンです。パターンの一部をサブパターンにすると、次の 2 つのことが可能になります。

ステップ 1 一連の選択肢の範囲を限定します。たとえば、次のパターン

```
cat(aract|erpi|lar|)
```

は「cat」、「cataract」、または「caterpillar」とマッチします。括弧を省略した場合は、「cataract」、「erpillar」、または空のストリングとマッチします。

ステップ 2 サブパターンをキャプチャ サブパターンとして設定します。つまり、パターン全体がマッチした場合、対象ストリングのうちサブパターンとマッチした部分が、`pcre_exec()` の `ovector` 引数を介して発信側に渡されます。左括弧の数が左から右にカウントされて(1 から開始)、キャプチャ サブパターンの番号が取得されます。

たとえば、対象ストリング「the red king」を次のパターンとマッチさせた場合、

```
the ((red|white) (king|queen))
```

キャプチャされるサブストリングは「red king」、「red」、および「king」であり、番号はそれぞれ 1、2、3 になります。

通常の括弧に 2 つの機能があることが便利であるとはかぎりません。通常は、サブパターンをグループ化する必要はあっても、キャプチャする必要はありません。左括弧のあとに疑問符およびコロンを付加すると、サブパターンはキャプチャを行わず、後続のキャプチャ サブパターン数を計算するときに、カウントされません。たとえば、対象ストリング「the white queen」を次のパターンとマッチさせた場合、

```
the ((?:red|white) (king|queen))
```

キャプチャされるサブストリングは「white queen」および「queen」で、番号はそれぞれ 1 および 2 です。キャプチャ サブパターンの最大数は 65535 で、すべてのサブパターンの最大ネストレベルは、キャプチャ サブパターン、非キャプチャ サブパターンの両方とも 200 です。

簡略形として、非キャプチャ サブパターンの先頭でオプション設定が必要な場合は、「?」と「:」の間にオプション文字を挿入できます。したがって、次の 2 つのパターン

```
(?i:saturday|sunday)
(?: (?i) saturday|sunday)
```

は、まったく同じストリング セットとマッチします。選択肢は左から右に試行され、サブパターンの末尾に到達しないかぎりオプションはリセットされないため、1 つの選択肢のオプション設定は後続の選択肢に影響も作用します。そのため、上記パターンは「saturday」と「sunday」にマッチします。

名前指定のサブパターン

キャプチャ用括弧を番号で指定する方法は簡単ですが、複雑な正規表現では番号の管理が非常に困難なことがあります。さらに、表現を変更した場合、番号も変更されることがあります。この問題を解決するために、PCRE では、Perl では備わっていない、サブパターンに名前を付ける機能をサポートしています。この機能には、Python 構文 (?P<name>...) を使用します。サブパターンの名前は英数字および下線からなり、パターン内で一意である必要があります。

名前を指定したキャプチャ用括弧には、名前だけでなく、引き続き番号も割り当てられます。PCRE API には、コンパイルされたパターンから名前/番号変換テーブルを抽出する関数呼び出し機能があります。また、キャプチャされたサブストリングを名前で抽出する便利な機能もあります。詳細については、pcreapi のマニュアルを参照してください。

繰り返し

繰り返しを指定するには、量指定子を使用します。量指定子は、次の項目のあとに指定できます。

- リテラル データ文字
- メタ文字
- ¥C エスケープ シーケンス
- ¥X エスケープ シーケンス (UTF-8 モードで Unicode プロパティを使用する場合)
- 単一文字とマッチする ¥d などのエスケープ
- 文字クラス
- 後方参照 (次の項を参照)
- 括弧で囲まれたサブパターン (アサーションは除く)

繰り返しの一般的な量指定子は、カンマで区切られた 2 つの数字を波括弧で囲んで、マッチ可能な最小数および最大数を指定します。量指定子の値は 65536 未満、最初の値は 2 番めの値以下にする必要があります。たとえば、

```
z{2,4}
```

は「zz」、「zzz」、または「zzzz」とマッチします。右括弧自体は特殊な意味を持ちません。2 番めの値を省略したにもかかわらず、カンマを指定している場合は、上限が設定されません。2 番めの値とカンマを両方とも省略した場合は、必要なマッチ数が正確に指定されます。つまり、

```
[aeiou]{3,}
```

は、少なくとも 3 回連続する母音とマッチしますが、それ以上連続する母音ともマッチすることもあります。一方、

¥d{8}

は 8 桁の数字にのみマッチします。量指定子を指定できない位置に配置された左波括弧、または量指定子の構文に適合しない左波括弧は、リテラル文字となります。たとえば、{6} は量指定子でなく、4 つの文字からなるリテラル スtring です。

UTF-8 モードの場合、量指定子は各バイトでなく UTF-8 文字に適用されます。したがって、たとえば、¥x{100}{2} は、それぞれ 2 バイトシーケンスで表される 2 つの UTF-8 文字とマッチします。同様に、Unicode プロパティ サポートを使用できる場合、¥X{3} は、3 つの Unicode 拡張シーケンスとマッチします。この場合、各シーケンスのバイト長は数バイトになることがあります (各バイト長が異なることがあります)。

量指定子 {0} を指定して、直前の項目および量指定子が存在しない場合と同様に正規表現を機能させることができます。

便宜上 (互換性を保つために)、最も一般的な 3 つの量指定子には 1 文字の省略形があります。

```
*   {0,} と同じ
+   {1,} と同じ
?   {0,1} と同じ
```

次の例のように、どの文字にもマッチしない可能性のあるサブパターンのあとに、上限を持たない量指定子を指定すると、無限ループになる可能性があります。

```
(a?)*
```

初期バージョンの Perl および PCRE では、このようなパターンをコンパイルするときに、エラーを生成していました。ただし、このようなパターンが便利な場合もあるため、現在は許可されていますが、サブパターンの繰り返しを実際にどの文字にもマッチしない場合は、ループは強制的に中断されます。

デフォルトでは、量指定子は「greedy (貪欲)」です。つまり、残りのパターンに失敗しない範囲内で、できるだけ多くの回数 (許可されている最大数まで) のマッチングを行います。この動作が問題となる一般的な例は、C プログラムのコメントとマッチする場合です。C プログラムのコメントは /* と */ で囲まれています。コメント内に * や / 文字が個別に使用されていることもあります。パターン

```
/¥*. *¥*/
```

を String

```
/* first comment */ not comment /* second comment */
```

に適用して C コメントをマッチさせると、マッチングに失敗します。* が貪欲であるため、String 全体とマッチしてしまうためです。ただし、量指定子のあとに疑問符を指定すると、貪欲でなくなり、できるだけ少ない回数とマッチします。したがって、パターン

```
/¥*. *?¥*/
```

は C コメントでも正しくマッチします。各量指定子の意味は変更されず、マッチ回数のみが変更されます。この疑問符の使用法と、量指定子としての使用法を混同しないでください。疑問符には 2 つの使用法があるため、

```
¥d??¥d
```

のように、二重に使用されることがあります。このパターンは、可能なかぎり 1 桁の数字とマッチしますが、2 桁の数字とマッチすることが残りのパターンのマッチ条件となる場合は、2 桁の数字にマッチできます。

PCRE_UNGREEDY オプション (Perl では使用できないオプション) が設定されている場合、量指定子がデフォルトで貪欲ではなくなります。ただし、量指定子の後に疑問符を付加することで、個別に貪欲にできます。つまり、デフォルト動作を逆にすることができます。

括弧で囲まれたサブパターンに、1 より大きな最小回数、または上限回数を指定すると、その最小回数または最大回数のサイズに応じて、コンパイルされたパターンに必要なメモリ サイズが大きくなります。

パターンが . * または {0,} で開始し、PCRE_DOTALL オプション (Perl の /s と同じ) が設定されているため、. が改行とマッチできる場合は、パターンが暗黙的に固定されます。これは、* または {0,} のあとに続くパターンのマッチングが対象 String 内のすべての文字位置で試行されるため、先頭以外の位置で全体マッチングを再試行しても意味がないからです。PCRE は通常、このようなパターンを、¥A が前に付加されたパターンとして処理します。

対象 String に改行が含まれていない場合は、PCRE_DOTALL を設定してこのような最適化を行ったり、^ を使用して固定パターンであることを明示的に指定することもできます。

ただし、このような最適化を使用できない場合が 1 つあります。キャプチャ用括弧内に .* があり、この括弧がパターン内のほかの場所における後方参照の対象となっている場合は、先頭でのマッチングに失敗し、そのあとのマッチングに成功することがあります。例を検討します。

```
(.*)abc¥1
```

対象ストリングが「xyz123abc123」の場合、マッチ位置は 4 番目の文字です。このため、このようなパターンは暗黙的な固定パターンではありません。

キャプチャ サブパターンを繰り返した場合、キャプチャされる値は最後の繰り返しでマッチしたサブストリングとなります。たとえば、

```
(tweedle[dume]{3}¥s*)+
```

を「tweedledum tweedledee」とマッチさせた場合、キャプチャされるサブストリング値は「tweedledee」です。ただし、キャプチャ サブパターンがネストされている場合、対応するキャプチャ値は、この前の繰り返しで設定された値であることがあります。たとえば、

```
/(a|(b))+/
```

を「aba」とマッチさせた場合、2 番めにキャプチャされたサブストリングの値は「b」になります。

アトミック グルーピングおよび独占的 量指定子

繰り返しの最大回数および最小回数を両方とも指定した場合に、その後の処理に失敗すると、通常は繰り返された項目が再評価され、異なる繰り返し回数でパターンの残りの部分をマッチできるかどうか調べられます。パターンの作成者がマッチングを継続する必要がないと判断した場合は、この処理を禁止して、マッチングの性質を変更するか、または本来よりも早くマッチングに失敗させると便利な場合があります。

たとえば、パターン ¥d+foo を次の対象ストリングに適用したとします。

```
123456bar
```

6 桁すべてとマッチして、「foo」とマッチしなかった場合、通常は 5 桁のみを使用して ¥d+ 項目とのマッチングが再試行されます。次に 4 桁と順に試行され、最終的にマッチングに失敗します。「アトミック グルーピング」(Jeffrey Friedl の著書から引用)では、サブパターンをマッチングしたあとに、このような再評価がされないように指定できます。

上記の例でアトミック グルーピングを使用した場合は、最初に「foo」のマッチングに失敗した時点で即座にマッチングが中断されます。この場合の表記は、次の例のように、(?> で始まる特殊な括弧を使用します。

```
(?>¥d+) foo
```

この種類の括弧で囲まれたパターン内の部分は、一度マッチングすると「ロック」されます。そのパターンのマッチングに再び失敗しても、ロック パターンへのバックトラックは発生しません。ただし、ロック パターンよりも前の項目へのバックトラックは、正常に機能します。

別の説明をすると、このタイプのサブパターンは、対象ストリング内の現在位置に固定されている場合に、同じスタンドアロン パターンがマッチする文字列とマッチします。

アトミック グルーピング サブパターンはキャプチャ サブパターンではありません。上記の例のような単純な場合では、できるだけ多くのものを包含するように繰り返し回数が最大化されます。したがって、¥d+ および ¥d+? はパターンの残り部分がマッチするようにマッチング桁数が調整されますが、(?>¥d+) は一連の数字全体にのみマッチされます。

一般に、アトミック グループには複雑なサブパターンを任意に含めたり、ネストすることができます。ただし、アトミック グループのサブパターンが、上記の例のように 1 回のみ繰り返される項目である場合、「独占的 量指定子」という、より簡単な表記を使用できます。独占的 量指定子は、量指定子のあとに + 文字を追加します。この表記を使用すると、上記の例は次のように書き直すことができます。

```
¥d++foo
```

独占的 量指定子は常に貪欲であり、PCRE_UNGREEDY オプションの設定は無視されます。独占的 量指定子は、アトミック グループの形式を簡素化する便利な表記法です。ただし、独占的 量指定子と、同等なアトミック グループには、意味または処理上の違いはありません。

独占的 量指定子の構文は、Perl 構文を拡張したものです。最初にこの構文が組み込まれたのは、Sun の Java パッケージでした。

パターン内のサブパターンに無制限の繰り返しが含まれていて、さらにそのサブパターン自体にも無制限の繰り返しが許可されている場合に、著しく長期化するマッチング処理の失敗を回避する唯一の方法は、アトミック グループを使用することです。次のパターン

```
(¥D+|<¥d+>)*[!/?]
```

は、数字以外の文字、または <> で囲まれた数字に ! や ? が続く、任意の個数のサブストリングとマッチします。マッチした場合は、短時間で処理されます。ただし、このパターンを次のストリングに適用するとします。

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

この場合、マッチの失敗を報告するまでに長い時間がかかります。これは、対象ストリングが内部の ¥D+ の繰り返しと外部の * の繰り返しの間でいくつもの方法により分割されることがあり、それらをすべて試行する必要があるためです(この例では、末尾に単一文字でなく [!/?] を使用しています。これは、単一文字を使用した場合、PCRE と Perl は両方とも最適化され、マッチングが早期に失敗するためです。マッチングに必要な最後の 1 文字が記憶され、対象ストリング内にこの文字がない場合は、早期に失敗します)。アトミック グループを使用するようにパターンを変更すると、次のようになります。

```
((?)¥D+|<¥d+>)*[!/?]
```

数字以外の部分を分割できなくなるため、マッチングは短時間で失敗します。

後方参照

文字クラスの外では、バックスラッシュのあとに 0 よりも大きな 1 桁の数字 (通常は複数桁) を指定すると、パターン内の前方 (左側) のキャプチャ サブパターンに対する後方参照になります。そのためには、この値と同数のキャプチャ用左括弧がその前にある必要があります。

ただし、バックスラッシュのあとに続く 10 進数が 10 未満の場合は、常に後方参照と解釈されます。エラーが発生するのは、パターン全体にこの値と同数のキャプチャ用左括弧がない場合のみです。つまり、参照される括弧は、番号が 10 未満の場合、参照の左側にある必要はありません。バックスラッシュのあとの数字の処理方法については、「非表示文字」を参照してください。

後方参照は、サブパターン自体とマッチするストリングでなく、現在の対象ストリング内のキャプチャ サブパターンと実際にマッチしたストリングとマッチします(このマッチング方法については、「サブルーチンとしてのサブパターン」を参照)。したがって、次のパターン

```
(sens|respons)e and ¥1ibility
```

は「sense and sensibility」および「response and responsibility」とマッチしますが、「sense and responsibility」とはマッチしません。後方参照で大文字と小文字を区別したマッチングが強制される場合は、大文字と小文字が考慮されます。たとえば、

```
((?i)rah)¥s+¥1
```

は「rah rah」および「RAH RAH」とマッチします。ただし、元のキャプチャ サブパターンが大文字と小文字を区別しないマッチングを行うにもかかわらず、「RAH rah」とはマッチしません。

名前指定のサブパターンに対する後方参照では、Python 構文 (?P=name) を使用します。上記の例は次の用のように書き換えることができます。

```
((?<p1>(?)rah)¥s+(?P=p1)
```

同じサブパターンに対する後方参照は複数回実行できます。特定のマッチングで実際に使用されなかったサブパターンに後方参照を実行すると、常に失敗します。たとえば、次のパターン

```
(a|bc)¥2
```

は、はじめに「bc」でなく「a」とのマッチングを開始した場合は、常に失敗します。パターン内には多数のキャプチャ用括弧を配置できるため、バックスラッシュのあとのすべての数字は、後方参照用数字の一部とみなされます。パターンのあとに数字が続く場合は、後方参照を終了するためのデリミタを使用する必要があります。PCRE_EXTENDED オプションが設定されている場合は、デリミタにスペース文字を使用できます。それ以外の場合は空のコメントを使用できます(「コメント」を参照)。

後方参照が参照先の括弧内に含まれている場合は、サブパターンを最初に使用した時点で失敗します。したがって、(a¥1) はマッチングしません。ただし、繰り返されるサブパターンの内部では、このような参照が便利なことがあります。たとえば、次のパターン

```
(a|b¥1)+
```

は任意の個数の「a」、および「aba」、「ababbaa」などにマッチします。サブパターンを繰り返すたびに、後方参照はそれ以前の繰り返しに対応する文字列とマッチします。この処理を機能させるには、最初の繰り返しで後方参照とのマッチングが不要になるようなパターンを作成する必要があります。そのためには、上記の例のような選択肢を使用するか、または最小値がゼロの量指定子を使用します。

アサーション

アサーションは、文字を実際に消費しない、現在のマッチング位置の前後の文字に対するテストです。¥b、¥B、¥A、¥G、¥Z、¥z、^、および \$ として符号化された単純なアサーションについては、[前述](#)してあります。

より複雑なアサーションは、サブパターンとして符号化します。アサーションのサブパターンには 2 種類あります。1 つは、対象ストリング内の現在位置の前方を検索し、もう 1 つは後方を検索します。アサーション サブパターンは、現在のマッチング位置が変更されない点を除いて、通常の方法でマッチングされます。

アサーション サブパターンはキャプチャ サブパターンではありません。また、同じ内容を複数回アサートしても意味がないため、繰り返すことができません。いずれかの種類のアサーションにキャプチャ サブパターンが含まれている場合、これらのサブパターンはパターン全体のキャプチャ サブパターンに番号を設定するためだと考えられます。ただし、サブストリングのキャプチャはネガティブアサーションでは意味をなさないため、ポジティブアサーションに限って実行されます。

前方アサーション

前方アサーションは、ポジティブアサーションでは (?=、ネガティブアサーションでは (?! で開始します。たとえば、

```
¥w+(?=;)
```

はセミコロンが続くワードとマッチしますが、マッチにセミコロンは含まれません。また、

```
foo(?!bar)
```

は後ろに「bar」が続かない「foo」の文字列とマッチします。一見、類似している次のパターン

```
(?!foo)bar
```

は、「foo」以外の文字列のあとに付加された「bar」を検索するものではありません。このパターンでは、次の 3 文字が「bar」である場合、アサーション (?!foo) は常に true となるため、「bar」はどんな場合でも検索されます。これと異なる効果を実現するには、後方アサーションを実行する必要があります。

パターン内の位置でマッチングを強制的に失敗させる場合は、(?!) を使用する方法が最も簡単です。この場合、空のストリングが常にマッチするため、空のストリングが存在してはならないアサーションが常に失敗するためです。

後方アサーション

後方アサーションは、ポジティブアサーションでは (?<=、ネガティブアサーションでは (?<! で開始します。たとえば、

```
(?!foo)bar
```

は前に「foo」が付加されない「bar」の文字列を検索します。後方アサーションの内容は、マッチするすべてのストリングが固定長となるように制限されます。ただし、選択肢がいくつかある場合、すべてを同じ固定長にする必要はありません。つまり、

```
(?<=bullock|donkey)
```

は許可されますが、

```
(?!dogs?|cats?)
```

を使用すると、コンパイル時にエラーが発生します。長さが異なるストリングとマッチする選択肢を指定できるのは、トップレベルの後方アサーションのみです。これは、すべての選択肢が同じ長さのストリングとマッチする必要がある Perl (5.8 以上) に対する拡張機能です。次のようなアサーション

```
(?<=ab(c|de))
```

は、トップレベルにある単一の選択肢が 2 つの異なる長さでマッチすることがあるため、許可されません。ただし、トップレベル選択肢を 2 つ使用するように書き換えると、パターンは有効になります。

```
(?<=abc|abde)
```

後方アサーションを実行すると、選択肢ごとに一時的に現在位置が固定幅分だけ後方に移動してから、マッチが試行されます。現在位置の前に、必要な文字数がない場合は、マッチに失敗します。

PCRE では、後方アサーションで ¥C エスケープ (UTF-8 モードで単一バイトとマッチ) を使用すると、後方の長さを計算できなくなるため、このエスケープは使用できません。¥X エスケープを指定すると、さまざまなバイト数とマッチすることがあるため、同様に使用できません。

アトミック グループを後方アサーションと併用して、対象ストリングの末尾で効率よくマッチングを指定できます。次のような単純なパターン

```
abcd$
```

を、マッチしない長いストリングに適用するとします。マッチングは左から右に処理されるため、PCRE は対象ストリング内の各「a」を検索してから、後続文字列が残りのパターンとマッチするかどうかを調べます。次のパターン

```
^. *abcd$
```

が指定されている場合、最初の .* はまずストリング全体とのマッチングを行います。マッチングに失敗すると (後続「a」がないため)、バックトラックを行って最後の文字を除くすべての文字をマッチングし、その後、最後の 2 文字を除くすべての文字をマッチングします。以下、同様に処理します。その後、ストリング全体に対して右から左に「a」が再検索されるため、この処理は有効ではありません。ただし、このパターンを次のように書き換えるか、

```
^(?>. *) (?<=abcd)
```

あるいは、同等な独占的量指定子構文を使用すると、

```
^. *+(?<=abcd)
```

.* 項目に関するバックトラックは発生しなくなるため、ストリング全体にのみマッチできます。以降の後方アサーションでは、最後の 4 文字に対するテストが 1 回実行されます。このテストに失敗すると、マッチは即座に失敗します。ストリングが長い場合は、この方法によって処理時間を大幅に短縮できます。

複数のアサーションの使用

複数の (任意の種類) のアサーションを連続して実行できます。たとえば、

```
(?<=¥d {3}) (?<!999) foo
```

は「999」以外の 3 桁の数字のあとに続く「foo」にマッチします。各アサーションは対象ストリング内の同じ位置で、それぞれ独立して適用されます。最初に、その前にある 3 文字がすべて数字であるかどうかをチェックし、その後、同じ 3 文字が「999」でないことを確認します。このパターンは、「foo」の前に 6 文字あり、その最初の 3 文字が数字で最後の 3 文字が「999」でない場合は、マッチしません。たとえば、「123abcfoo」にはマッチしません。このような文字列とマッチするパターンは、次のとおりです。

```
(?<=¥d {3} . . .) (?<!999) foo
```

この場合、最初のアサーションが先行の 6 文字を検索し、最初の 3 文字が数字であることをチェックしてから、2 番目のアサーションで先行の 3 文字が「999」でないことを確認します。

アサーションは自由な組み合わせでネストできます。たとえば、

```
(?<=(?!foo)bar)baz
```

は、前に「bar」があり、その前に「foo」が配置されていない「baz」の文字列にマッチしますが、

```
(?<=¥d{3}(?!999)...)foo
```

は、3桁の数字と「999」以外の任意の3文字のあとにある「foo」にマッチする別のパターンです。

条件付きサブパターン

条件付きでサブパターンに従ってマッチング処理させることができます。また、アサーションの結果や、その前のキャプチャサブパターンのマッチ状況に応じて、2つの選択肢を含むサブパターンのいずれかを選択することができます。条件付きサブパターンには、2つの形式があります。

```
(?(condition)yes-pattern)
(?(condition)yes-pattern|no-pattern)
```

条件が満たされている場合は、yes-pattern が使用されます。条件が満たされていない場合は、no-pattern があれば、これが使用されます。サブパターン内に選択肢が3つ以上ある場合は、コンパイル時にエラーが発生します。

条件には3つの種類があります。括弧内のテキストが一連の数字で構成されている場合は、この数字に対応するキャプチャサブパターンが以前にマッチしていれば、条件が満たされます。0より大きい数字を指定する必要があります。次のパターンについて検討します。このパターンでは、読みやすくするために重要でないスペースが挿入され(PCRE_EXTENDED オプションを使用していると想定)、説明しやすくするために3つに分割されています。

```
( ¥( )?  [ ^() ]+  (?(1) ¥) )
```

最初の部分はオプションの左括弧とマッチします。この文字が存在する場合は、最初にキャプチャされるサブstringとして設定します。2番目の部分は、括弧以外の1つまたは複数の文字とマッチします。3番目の部分は、最初の括弧のセットがマッチしたかどうかをテストする条件付きサブパターンです。マッチした場合、つまり、対象stringが左括弧で開始した場合は、条件がtrueになり、yes-pattern が実行され、右括弧が必要となります。マッチしなかった場合は、no-pattern が指定されていないため、サブパターンはどの文字列ともマッチしません。つまり、このパターンは任意に括弧で囲まれている括弧なしのシーケンスとマッチします。

条件がstring(R)の場合は、パターンまたはサブパターンの再帰呼び出しが行われた場合に、条件が満たされます。「トップレベル」では、条件がfalseになります。これはPCREの拡張機能です。再帰パターンについては、次の項を参照してください。

条件が一連の数字または(R)でない場合、条件をアサーションにする必要があります。ポジティブまたはネガティブ、前方または後方のアサーションを使用できます。次のパターンについて検討します。重要でないスペース文字が含まれ、2番目の行には選択肢が2つあります。

```
(?(?=[^a-z]*[a-z])
 ¥d{2}-[a-z]{3}-¥d{2} | ¥d{2}-¥d{2}-¥d{2} )
```

条件は、文字以外のあとに1文字が続く任意のシーケンスとマッチするポジティブ前方アサーションです。つまり、対象string内に少なくとも1文字が存在するかどうかをテストされます。文字が見つかった場合、対象stringは最初の選択肢とマッチします。それ以外の場合は、2番目の選択肢とマッチします。このパターンは、dd-aaa-dd または dd-dd-dd の2つの形式のいずれかのstringとマッチします。aaa は文字、dd は数字です。

コメント

コメントはシーケンス(?# で開始し、次の右括弧で終了します。括弧をネストすることはできません。コメント内の文字列は、パターンマッチングにはまったく関与しません。

PCRE_EXTENDED オプションが設定されている場合、文字クラスの外にあるエスケープされていない#文字からもコメントは開始され、パターン内の次の改行文字で終了します。

再帰パターン

括弧を無制限にネストできる場合に、括弧内のストリングをマッチングする場合の問題について検討します。再帰を使用しない場合の最適な方法は、マッチングするネストレベルの上限が設定されているパターンを使用することです。任意のネストレベルを処理することはできません。Perl には、(多数の機能の 1 つとして)正規表現を再帰化する機能があります。この機能は、実行時に正規表現内の Perl コードを補間して実行されます。Perl コードは正規表現自体を参照できます。括弧の問題を解決するための Perl パターンは、次のように作成できます。

```
$re = qr {¥( (? ( ?>[^( )]+ ) | (?p{$re}) ) * ¥ ) } x;
```

(?p{...}) 項目は実行時に Perl コードを補間します。この場合は、このコードが挿入されたパターンを再帰的に参照します。PCRE が Perl コードの補間をサポートできないのは明白です。その代わりに、パターン全体の再帰、およびサブパターンの個別的な再帰に関する特殊な構文をいくつかサポートしています。

(? のあとにゼロより大きな値および右括弧が続く特殊な項目が、指定された番号を持つサブパターン内にある場合、この項目はこのサブパターンの再帰呼び出しです。この項目が、指定された番号を持つサブパターン内でない場合、この項目は「サブルーチン」呼び出しです(次の項を参照)。特殊項目 (?R) は正規表現全体の再帰呼び出しです。

たとえば、次の PCRE パターンでは、ネストされた括弧の問題が解決されます(スペース文字が無視されるように PCRE_EXTENDED オプションが設定されていると想定)。

```
¥( ( ?>[^( )]+ ) | (?R) ) * ¥
```

最初に、このパターンは左括弧とマッチします。次に、括弧以外の文字からなるサブストリング、またはパターン自体(正しい括弧が設定されたサブストリング)と再帰的にマッチングするサブストリングと、何回でもマッチングします。最後に、右括弧とマッチします。

このパターンが、より大きなパターンの一部である場合は、パターン全体を再帰化する代わりに次のパターンを使用できます。

```
( ¥( ( ?>[^( )]+ ) | (?1) ) * ¥ )
```

パターンを括弧で囲み、再帰がパターン全体でなく括弧に適用されるようにしました。より大きなパターンでは、括弧数の追跡が複雑になることがあります。この方法の代わりに、名前指定の括弧を使用すると、処理はもっと簡単になります。このため、PCRE は (?P>name) という、PCRE が名前付き括弧に使用する Python 構文に対する拡張機能を使用します(Perl には名前付き括弧機能がありません)。上記の例は次の用のように書き換えることができます。

```
(?P<pn> ¥( ( ?>[^( )]+ ) | (?P>pn) ) * ¥ )
```

この特定のパターン例では、繰り返しを無制限にネストすることができます。したがって、マッチしないストリングにパターンを適用する場合は、括弧なしストリングのマッチングにアトミック グルーピングを使用することが重要です。たとえば、このパターンを次のストリングに適用した場合、

```
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa)
```

即座に「マッチしない」と報告されます。ただし、アトミック グルーピングを使用しない場合は、+ と * の繰り返しによって対象ストリングを分割する方法が多く、そのすべてを失敗報告前にテストする必要があるため、マッチングの所要時間が非常に長くなることがあります。

マッチングの最後に、キャプチャ サブパターン用に設定される値は、サブパターン値が設定されている最も外側の再帰レベルから取得されます。中間の値を取得する場合は、コールアウト関数を使用します(「サブルーチンとしてのサブパターン」および `pcrecallout` のマニュアルを参照)。上記パターンが以下とマッチする場合

```
(ab(cd)ef)
```

キャプチャ用括弧の値は、トップレベルで取得された最後の値である「ef」です。さらに括弧を追加して、次のストリングを指定した場合、

```
¥( ( ( ?>[^( )]+ ) | (?R) ) * ) ¥
```

キャプチャされる文字列は、トップレベルの括弧の内容である「ab(cd)ef」になります。

1つのパターン内のキャプチャ用括弧数が15を超える場合、PCREは再帰中にデータを格納するための追加メモリを取得する必要があります。そのためには、`pcre_malloc` を使用してメモリを取得して、あとで `pcre_free` を使用してメモリを解放します。メモリを取得できない場合、マッチングは失敗し、`PCRE_ERROR_NOMEMORY` エラーが発生します。

(?R) 項目と、再帰をテストするための条件 (R) を混同しないでください。次のパターンについて検討します。このパターンは山括弧内のテキストをマッチングし、任意にネストを指定できます。ネストされた括弧内(つまり再帰内)では数字のみを指定できますが、その他のレベルでは任意の文字を指定できます。

```
< (? ( (?R) ¥d++ | [^\<]*+ ) | (?R) ) * >
```

このパターン内の (?R) は条件付きサブパターンの開始を意味します。再帰と非再帰の両方に対応する2つの選択肢があります。(?R) 項目は実際に再帰を呼び出します。

サブルーチンとしてのサブパターン

再帰サブパターンを参照する構文(番号指定または名前指定)を参照先括弧の外側で使用した場合は、プログラミング言語のサブルーチンと同様に機能します。上記の例では、次のパターン

```
(sens|respons)e and ¥ibility
```

は「sense and sensibility」および「response and responsibility」とマッチしますが、「sense and responsibility」とはマッチしません。代わりに、次のパターン

```
(sens|respons)e and (?1)ibility
```

を使用した場合、上記の2つの文字列に加えて、「sense and responsibility」にもマッチします。ただし、このような参照は、参照先のサブパターンのあとに記述する必要があります。

コールアウト

Perlには、シーケンス (?{...}) を使用して、正規表現のマッチング中に任意の Perl コードを適用する機能があります。これにより、繰り返しがある場合に、同じ括弧のペアとマッチする別のサブ文字列の抽出などが可能になります。

PCREにも同様な機能がありますが、任意の Perl コードを適用することはできません。この機能を「コールアウト」といいます。PCREの呼び出し元はグローバル変数 `pcre_callout` に入力口を設定して、外部関数を使用できるようにします。デフォルトでは、この変数には、すべてのコールアウトをディセーブルにする NULL が含まれています。

正規表現の (?C) は、外部関数の呼び出し位置を示します。別の呼び出し位置を識別する場合は、文字 C のあとに256未満の数字を設定します。デフォルト値は0です。たとえば、次のパターンには2つのコールアウト位置があります。

```
(?C1)¥dabc (?C2) def
```

`PCRE_AUTO_CALLOUT` フラグが `pcre_compile()` に渡されると、パターン内の各項目よりも先にコールアウトが自動的に導入されます。これらのコールアウトにはすべて255の番号が付けられます。

マッチング中に PCRE がコールアウト位置に到達すると(`pcre_callout` が設定されている場合)、外部関数が呼び出されます。外部関数にはコールアウトの番号、パターン内の位置、およびオプションとして本来 `pcre_exec()` の呼び出し元が提供する1つのデータ項目が与えられます。コールアウト関数はマッチングを継続したり、バックトラックしたり、失敗させることができます。コールアウト関数のインターフェイスの詳細については、`precallout` のマニュアルを参照してください。

最終更新日: 2004年9月9日

Copyright © 1997-2004 University of Cambridge.