



Découvrez nos magazines :



- GNU/Linux Magazine
  - [Dernier Numéro Paru](#)
  - [Tous les anciens Numéros](#)
  - [Abonnements : GNU/Linux Magazine](#)
  - [Toutes les offres d'abonnement avec ce titre](#)



- GNU/Linux Magazine HORS-SÉRIE
  - [Dernier Numéro Paru](#)
  - [Tous les anciens Numéros](#)
  - [Abonnements : GNU/Linux Magazine HS](#)
  - [Toutes les offres d'abonnement avec ce titre](#)



- Linux Pratique
  - [Dernier Numéro Paru](#)
  - [Tous les anciens Numéros](#)
  - [Abonnements : Linux Pratique](#)
  - [Toutes les offres d'abonnement avec ce titre](#)



- Linux Pratique HORS-SÉRIE
  - [Dernier Numéro Paru](#)
  - [Tous les anciens Numéros](#)
  - [Abonnements : Linux Pratique HS](#)
  - [Toutes les offres d'abonnement avec ce titre](#)



- Linux Pratique Essentiel
  - [Dernier Numéro Paru](#)
  - [Tous les anciens Numéros](#)
  - [Abonnements : Linux Pratique Essentiel](#)
  - [Toutes les offres d'abonnement avec ce titre](#)



- Linux Pratique Essentiel HORS-SÉRIE
  - [Dernier Numéro Paru](#)
  - [Tous les anciens Numéros](#)
  - [Abonnements : Linux Pratique Essentiel HS](#)
  - [Toutes les offres d'abonnement avec ce titre](#)



- Misc
  - [Dernier Numéro Paru](#)
  - [Tous les anciens Numéros](#)
  - [Abonnements : MISC](#)
  - [Toutes les offres d'abonnement avec ce titre](#)



- Misc HORS-SÉRIE
  - [Dernier Numéro Paru](#)
  - [Tous les anciens Numéros](#)
  - [Abonnements : MISC HS](#)
  - [Toutes les offres d'abonnement avec ce titre](#)



- Open Silicium
  - [Dernier Numéro Paru](#)

- o  [Tous les anciens Numéros](#)
- o  [Abonnements : Open Silicium](#)
- o  [Toutes les offres d'abonnement avec ce titre](#)

Il y a **1,110** articles/billets en ligne.

Mots-clés

Dans

Publié(s) dans le(s) magazine(s) :

GNU/Linux Magazine	Linux Pratique	Linux Pratique Essenti
GNU/Linux Magazine	Linux Pratique	Linux Pratique Essenti
Hors-séries	Hors-séries	Hors-sé

## [News](#) Les nouveautés de Perl 5.10 - deuxième partie

06/04/2009



Posté par [La rédaction](#) | Signature : Sébastien Aperghis-Tramoni

Tags : [GLMF](#)

 [0 Commentaire](#) | [Ajouter un commentaire](#)



Retrouvez cet article dans : [Linux Magazine 106](#)

Nous avons vu dans l'article précédent une bonne partie des nouveautés de Perl 5.10 qui touchent au langage ou à son utilisation. Nous allons cette fois-ci regarder ce qui constitue le plus gros morceau des ajouts de cette version : les nouvelles fonctionnalités des expressions régulières.

## 1. Expressions régulières

*Perl has often been tagged as a language in which it's easy to write programs that are difficult to read, and it's no secret that regular expression syntax that has been the chief culprit. Funny that other languages have been borrowing Perl's regular expressions as fast as they can...*

Larry Wall, Apocalypse 5

S'il y a bien une chose que même les détracteurs de Perl reconnaissent à ce langage, c'est son support exceptionnel des expressions régulières, intégré au cœur de la syntaxe. C'est aussi Perl et sa communauté qui ont contribué à améliorer de façon radicale les possibilités de cette classe de mini-langages, faisant exploser les limites du moteur POSIX, et contribuant ainsi à les populariser.

Les développeurs d'autres langages et programmes, ne voulant pas rester en retrait, ont donc écrit un moteur d'expressions régulières, différent du moteur interne de Perl, permettant d'implémenter la syntaxe des expressions régulières de Perl. Ainsi naquit PCRE (Perl Compatible Regular Expressions), originellement pour le MTA Exim, mais maintenant utilisé par un nombre croissant de programmes et de langages.

De manière intéressante, PCRE ne s'est pas contenté de suivre Perl, mais a commencé à ajouter de nouvelles fonctionnalités. Dans certains cas, ce n'est pas au niveau de PCRE, mais au niveau de son inclusion dans un langage spécifique (par exemple C#) que certaines fonctionnalités sont apparues. L'un dans l'autre, Perl avait donc pris du retard par rapport à PCRE et donc aux autres langages.

Perl 5.10 permet à la fois de rattraper ce retard et de reprendre de l'avance en ajoutant de nouvelles sémantiques expérimentées dans des modules CPAN ou inspirées, là encore, de Perl 6. Par ailleurs, au niveau interne, le moteur de Perl a été retravaillé en profondeur. Voyons tout ceci plus en détail.

## 2. Syntaxe et sémantique

### 2.1 Nouveaux quantifieurs

Pour compléter les quantifieurs déjà disponibles, une nouvelle série a été introduite, les quantifieurs possessifs, qui, quand un motif ne correspond pas, l'empêchent de revenir en arrière (backtrack).

*+	correspond 0 ou plusieurs fois, et ne rend jamais
++	correspond 1 ou plusieurs fois, et ne rend jamais
?+	correspond 0 ou 1 fois, et ne rend jamais
{n}+	correspond exactement n fois et ne rend jamais
{n,}+	correspond au moins n fois et ne rend jamais

`{n,m}+` correspond au moins `n` fois mais pas plus que `m` fois,  
et ne rend jamais

Voici un exemple pour comprendre la différence avec les quantifieurs déjà existants :

```
"aaaa" =~ /(a+)a/; # $1 contient "aaa"
"aaaa" =~ /(a?)a/; # $1 contient "a"
"aaaa" =~ /(a++)a/; # échoue
```

La troisième expression, qui utilise un quantifieur possessif, échoue, car `/a++/`, comme `/a+/` va essayer de faire correspondre le plus possible de "a", mais contrairement à `/a+/`, il n'essaye pas avec moins de caractères si cela échoue au premier coup. Cela permet donc d'éviter de coûteux retours arrière dans les expressions complexes.

## 2.2 Nouveaux verbes

Le moteur d'expressions régulières de Perl est basé sur un automate à états finis non-déterministe ou NFA (Nondeterministic Finite Automaton), qui utilise le retour arrière (backtracking) pour offrir des fonctionnalités avancées comme les références arrières (ce qui fait d'ailleurs que ces expressions ne sont pas très régulières, au sens mathématique du terme). En principe, les expressions régulières sont déclaratives, et ne dépendent pas de leur implémentation. Toutefois, le retour arrière a un coût non négligeable en termes de temps d'exécution, et c'est pourquoi, au fil du temps, la grammaire s'est enrichie pour permettre un meilleur contrôle du retour arrière. On vient d'en voir un exemple avec les nouveaux quantifieurs, qui permettent d'empêcher le retour arrière sur des sous-motifs. Comme c'était malgré tout insuffisant pour certaines utilisations très avancées, des verbes ont été ajoutés afin d'offrir un contrôle très fin, pour limiter ou au contraire forcer le retour arrière.

La forme générale de ces verbes est `(*VERB:ARG)`, l'argument étant généralement optionnel, et dans certains cas totalement absent. Quand un motif contient un de ces verbes, Perl positionne les variables ~~globales~~ `$REGERROR` et `$REGMARK`:

- en cas de succès, ~~`$REGERROR`~~ est fausse et ~~`$REGMARK`~~ contient le nom du dernier `(*MARK:NAME)` exécuté ;

- en cas d'échec, ~~\$REGMARK~~ est fausse et ~~\$REGERROR~~ contient l'argument passé au dernier verbe exécuté ou, s'il n'y en avait pas, le nom du dernier (~~\*MARK:NAME~~) exécuté ou sinon une valeur vraie.

Ces variables ne sont pas magiques, mais sont simplement globales, comme ~~\$AUTOLOAD~~, et peuvent donc être rendues locales à un bloc avec ~~local~~.

Ces verbes sont marqués expérimentaux, mais, au vu des précédentes fonctionnalités des expressions marquées expérimentales, il est assez probable qu'ils seront conservés. Les verbes actuellement disponibles sont les suivants :

- (~~\*FAIL~~), (~~\*F~~)  
Le plus simple verbe, qui est véritablement du sucre syntaxique pour (~~?!~~), permet de forcer le moteur à effectuer des retours arrière afin de déterminer toutes les correspondances possibles. C'est donc similaire à la recherche exhaustive de Perl 6.
- (~~\*ACCEPT~~)  
Ce verbe provoque l'acceptation de la correspondance courante du motif, au point où est situé (~~\*ACCEPT~~). Yves Orton le décrit comme un genre d'instruction return pour les expressions régulières.
- (~~\*PRUNE~~), (~~\*PRUNE:NAME~~)  
Quand le moteur effectue un retour arrière sur ce verbe, il fait échouer la correspondance à la position de départ courante. C'est similaire à (~~?>..~~) sauf que (~~\*PRUNE~~) est unaire alors que (~~?>..~~) est une construction de sous-motif.  
On l'appelle " prune " (" tailler " en anglais, un terme de jardinage) parce qu'il sert à tailler l'arbre de recherche de retours arrière en supprimant les branches qu'on sait ne pas vouloir parcourir.
- (~~\*SKIP~~), (~~\*SKIP:NAME~~)  
(~~\*SKIP~~) est similaire à (~~\*PRUNE~~), à la différence qu'en cas de retour arrière par échec, il indique au moteur de sauter jusqu'au point où il se situe. Si un nom est donné en argument, le moteur sautera jusqu'au point marqué avec ce nom par (~~\*MARK~~).
- (~~\*MARK:NAME~~), (~~\*:NAME~~)  
Ajouté pour répondre aux besoins de certains programmes exigeants comme SpamAssassin, ce verbe permet de marquer la position courante dans la chaîne et de lui donner un nom.

```
/zlonk+(*MARK:zlonk)|qunck+(*MARK:qunck)|cr+a+ck(MARK:crack)/
```

Cela permet entre autres de voir le chemin suivi par le moteur pour

faire correspondre le motif à la chaîne, et facilite donc grandement le débogage.

- ~~(\*THEN), (\*THEN:NAME)~~

Comme son nom le laisse supposer, ce verbe fournit un genre d'instruction ~~if/then~~ pour expressions régulières. Conçu pour être utilisé au sein d'un groupe d'alternatives, revenir sur ce verbe fait passer le moteur à la branche suivante, réalisant donc l'équivalent d'un ~~else~~.

```
/( COND1 (*THEN) SUBPATTERN1 | COND2 (*THEN) SUBPATTERN2 )/x
```

Utilisé en dehors d'un groupe d'alternatives, ce verbe est équivalent à ~~(\*PRUNE)~~. Il existe en Perl 6 sous le nom d'opérateur `cut group`, noté `::`.

- ~~(\*COMMIT)~~

Ce verbe est l'équivalent du motif `commit` de Perl 6, où il s'écrit ~~<commit>~~ ou `::`. Il est similaire à ~~(\*SKIP)~~, sauf que quand le moteur fait un retour arrière dessus pour cause d'échec, il provoque un arrêt immédiat par échec de la correspondance.

## 2.3 Captures

Les captures sont disponibles en Perl depuis le début, mais la manière de s'y référer a évolué (en bien) au fil des années. Ainsi, au début, on pouvait se référer aux parenthèses capturantes avec la syntaxe traditionnelle ~~\N~~, où N est le numéro de la parenthèse ouvrante (en comptant à partir de la gauche).

```
"moo" =~ /(\w)\1/ # détecte des caractères dupliqués
```

Cette notation marche, mais pose problème quand on a plus de 10 captures. En effet, Perl a normalement l'habitude de traduire ~~\10~~ par ~~\010~~, c'est-à-dire le caractère ASCII 8 (backspace) en notation octale. Pour éviter les problèmes, Perl ne résout ~~\10~~ comme une référence de capture que si au moins 10 captures ont été déclarées. Et de même pour ~~\11~~, ~~\12~~, etc. Néanmoins, cela rend le code assez confus et cette syntaxe ne peut être utilisée qu'au sein d'une expression régulière. Hors de l'expression, on dispose des variables numérotées (cette fois-ci en base décimale) ~~\$1~~, ~~\$2~~, ~~\$3~~... pour se référer aux captures et récupérer leurs valeurs.

Un autre aspect qui rend l'utilisation de nombreuses captures assez complexe

est qu'elles sont numérotées dans l'ordre d'apparition, de gauche à droite, sans tenir compte de la notion d'alternative. H. Merijn Brand a proposé pour cela une première solution, la remise à zéro de branche (branch reset pattern). C'est un groupe non capturant, comme  $(?|...)$ , mais noté  $(?|...)$  et qui permet de numéroté les captures de chaque branche de l'alternative comme si elle était seule. Les captures suivant l'alternative sont ensuite numérotées par rapport à la branche contenant le plus de captures. L'exemple suivant, qui pourrait lire un format de log, illustre comment s'effectue cette numérotation :

```
m{
  # 1: date ISO, 2: hôte, 3: processus
  (\d+T\d+) \s+ (\S+) \s+ (\S+) \s+
  (?|
    # 4: msgid, 5: liste de clés-valeurs
    (\w+): \s+ ( (? \w+=\S+,?)+ \s+ )w
    |
    # 4: msgid, 5: sens, 6: liste de clés-valeurs
    (\S+) \s+ (=>|<|=) \s+ ( (? \w+=\S+)+ \s+ )
    |
    # 4: msgid, 5: reste de la ligne
    (\d+\.\d+) \s+ (.+?)
  )
  # 7: statut
  -> status: (\w+)
}x;
```

Étant assez simple à implémenter, cette fonctionnalité a rapidement été ajoutée dans PCRE (voir la section Duplicate subpattern numbers).

Néanmoins, le fait même de numéroté pose problème dès que l'on désire construire de grandes expressions par concaténation et interpolation de morceaux atomiques. On se retrouve obligé de jongler pour s'y retrouver, et cela devient rapidement un casse-tête si certains morceaux sont dynamiquement modifiables. Imaginez ce que cela donne pour un module comme ~~Regexp::Common~~ !

Perl 5.10 offre une réponse en proposant une nouvelle syntaxe,  $\backslashg\{N\}$ , où N peut maintenant être un entier positif ou négatif. Positif, cela correspond au numéro habituel de la capture. Négatif, c'est une référence relative arrière, qui correspond donc à la N-ième capture précédente. Ainsi,  $\backslashg\{-1\}$  se réfère à la précédente capture et  $\backslashg\{-2\}$  à celle qui la précède (la pénultième). On peut ainsi écrire un morceau d'expression qui détecte les mots en double :

```
my $find_dup = qr/ (\w+) \s+ \g{-1} /x;
```

et qui peut maintenant être interpolé sans problème au sein d'une expression plus importante.

Encore mieux, on peut en plus donner un nom aux captures, afin de s'y retrouver plus facilement. La syntaxe pour déclarer et référencer les captures nommées est reprise de celle de .Net : on utilise la syntaxe `(?<name>pattern)` pour les déclarer et `\k<name>` pour les rappeler. On peut aussi utiliser `\g{name}` si on préfère. La variation `(?'name'pattern)\k'name'`, ainsi que la syntaxe Python `((?namepattern)/())` sont supportées pour faciliter la passage des expressions régulières entre les différents langages. Le détecteur de doublons peut maintenant s'écrire :

```
my $find_dup = qr/ (?<dup_word>\w+) \s+ \k<dup_word> /x;
```

Il faut noter que `\k<name>` se réfère toujours à la première capture nommée "`name`" définie (c'est-à-dire la première qui a réussi), y compris quand plusieurs captures utilisent le même nom. Tout ceci rend donc quasiment trivial l'utilisation de bouts d'expressions comme ce détecteur de doublons au sein d'expressions plus grandes.

Hors de l'expression même, on dispose des variables `%+` et `%-` pour accéder aux captures : `#{name}` qui est identique à `\k<name>`, et `#{name}` (qui est une référence vers un tableau) pour avoir toutes les captures de ce nom, même celles qui n'ont pas correspondu. Il faut noter que ces variables ne sont disponibles que dans le bloc lexical correspondant à l'expression régulière de définition.

```
while ($text =~ /$find_dup/gsm) {
    print "mot en double : #{dup_word}\n";
}
```

Un dernier point intéressant est que `%+` et `%-` sont en réalité de simples variables rendues spéciales par le module `Tie::Hash::NamedCapture`, ce qui autorise donc l'utilisation d'autres variables si on trouve que celles par défaut ont un nom trop obscur.

```
use Tie::Hash::NamedCapture;
tie my %last_match, "Tie::Hash::NamedCapture";
# %last_match se comporte maintenant comme %+
```

```
tie my %reg_buffer, "Tie::Hash::NamedCapture", all => 1;
# %reg_buffer se comporte maintenant comme %-
```

## 2.4 Motifs récursifs

Perl avait déjà un mécanisme pour construire des expressions régulières récursives, utilisant le constructeur de motifs dynamiques (~~??{ code }~~). Ainsi, l'exemple suivant permet de reconnaître en Perl 5.8.X des groupes de parenthèses imbriquées :

```
$re = qr{ (      # groupe #1
  \(          # parenthèse ouvrante
 (?:
    (?> [^()]+ ) # groupe sans retour arrière
    |
    (??{ $re }) # groupe avec parenthèses
  )*
  \)          # parenthèse fermante
}x;
```

Cela fonctionne par une interpolation de l'expression au sein d'elle-même, réalisée à l'exécution du programme. Mais c'est assez inélégant, car c'est une solution externe au moteur d'expressions régulières, ce qui impose des limitations (la variable à interpoler doit être globale) et des lourdeurs (l'expression est complexe, et c'est globalement lent).

Pour cette raison, la syntaxe existante de PCRE, bien plus complète et efficace, a été importée dans Perl 5.10. Le principe est de permettre de ré-invoquer un groupe capturant avec (~~?PARNO~~), où ~~PARNO~~ est le numéro de parenthèse (c'est-à-dire du groupe). Si ce nombre est précédé d'un signe, alors il est compris de manière relative. Ainsi, (~~?2~~) invoque le 2e groupe déclaré, (~~?-1~~) le dernier groupe déclaré, et (~~?+1~~) le prochain groupe qui sera déclaré. (~~?0~~) ou (~~?R~~) permet de ré-invoquer le motif complet. L'exemple précédent peut se récrire :

```
$re = qr{ (      # groupe #1
  \(          # parenthèse ouvrante
 (?:
    (?> [^()]+ ) # groupe sans retour arrière
    (?1)        # groupe avec parenthèses
  )*
  \)          # parenthèse fermante
}x;
```

```
}x;
```

Ce qui est un peu plus clair, et permet de rester entièrement dans le moteur. Bien sûr, essayer d'invoquer un groupe qui n'existe pas provoque une erreur fatale.

Plus intéressant, on peut invoquer un groupe nommé en utilisant la syntaxe `(?&name)`. La syntaxe Python `(?P>name)` est aussi supportée.

On dispose aussi d'une construction conditionnelle `(?(condition)yes-pattern|no-pattern)` qui accepte les conditions possibles suivantes :

- un numéro de groupe `(1)`, `(2)`... pour tester si le groupe de ce numéro a capturé quelque chose ;
- un nom de groupe `<name>` ou `'name'` pour tester si le groupe de ce nom a capturé quelque chose ;
- un bout de code Perl `(?{CODE})`, qui est traité comme la condition ;
- le symbole `(R)` pour tester si l'expression a été évaluée au sein d'une récursion ; il peut être suivi du numéro d'un groupe `((R1)`, `(R2)`..) ou du nom d'un groupe `((R&name))` pour tester si l'expression a été évaluée pendant l'exécution du groupe en question ; à noter que cela ne regarde que le niveau juste au-dessus ;
- `(DEFINE)` est un cas un peu particulier, car seule la partie `yes-pattern` est autorisée, mais elle n'est jamais directement exécutée. On peut, par contre, entrer en récursion dans les sous-motifs qu'elle contient. En leur donnant un nom par la syntaxe usuelle, cela permet donc de définir des genres de fonctions de motifs, que l'on peut "cacher" dans un coin, mais qu'on peut librement invoquer. Par exemple :

```
m{
    (?<src_addr>(?!ip_addr))    # adresse IP source
    \s+ -> \s+
    (?<dest_addr>(?!ip_addr))  # adresse IP destination
    (? (DEFINE)
        (?<ip_addr>...)        # motif pour reconnaître une
    )                          # adresse IP
}x
```

Un groupe de capture défini au sein d'un `(?(DEFINE)...)` n'est pas accessible au retour de la récursion, raison pour laquelle on doit inclure son invocation au sein d'un autre groupe capturant.

## 2.5 Nouvelle assertion Keep

Proposée et testée par Jeff Pinyan dans son module `Regexp::Keep`, l'assertion `\K` (pour Keep), permet d'effectuer des recherches et remplacements sans que la partie à gauche de `\K` ne soit touchée. C'est donc un genre d'assertion de présence arrière (positive look-behind assertion) autorisant un motif de longueur variable, mais de manière bien plus efficace.

Cela a typiquement été conçu pour pouvoir récrire des constructions comme `s/(save)delete/$1/` en `s/save\Kdelete//`. L'expression utilisant `\K` est plus rapide que celle utilisant la capture, car on évite l'allocation des structures de gestion des buffers. Techniquement, cela fonctionne en modifiant le pointeur `PL_regstartp` (qui est derrière les variables `$&` ou `${^MATCH}`) pour qu'il pointe vers l'endroit courant de la chaîne où se situe `\K` une fois que la partie gauche a correspondu. Cette astuce permet mine de rien de diviser par deux le temps d'exécution.

Le module `Regexp::Keep`, disponible sur le CPAN, autorise les mêmes gains et fonctionne sur Perl 5.6 et 5.8.

Perl 6 dispose d'assertions similaires, `<<` qui est équivalente à `\K` pour le début de la chaîne, et `>>` qui est son pendant en fin de chaîne.

## 2.6 Nouvelles classes de caractères

Quelques nouvelles classes de caractères ont été ajoutées, toutes liées à différents types d'espaces :

- `\v` correspond à une espace verticale, et `\V` à l'inverse.
- `\h` correspond à une espace horizontale, et `\H` à l'inverse.
- `\R` correspond à un saut de ligne, y compris le saut de ligne réseau CRLF ; plus précisément, cette classe est exactement équivalente à `(?>\x0D\x0A?|[\x0A-\x0C\x85\u{2028}\u{2029}])` (mais en bien plus facile à taper et à comprendre).

## 3. Changements internes

Généralement, les changements internes sont moins intéressants pour l'utilisateur final que les ajouts au niveau syntaxique et sémantique, mais, dans le cas présent, les modifications apportées au moteur interne sont

véritablement importantes de par leurs conséquences (positives) pour l'utilisateur et méritent donc d'être détaillées plus avant.

### 3.1 Dérécursion du moteur

En premier lieu, le moteur d'expressions régulières a été dérécursé par Dave Mitchell. La conséquence directe est qu'un grand nombre de bugs et de limitations liés à la nature précédemment récursive du moteur ont disparu.

Si on regarde plus dans le détail, il faut voir plus précisément comment fonctionne le moteur d'expressions régulières. Les principales opérations sont une analyse de l'expression qui construit l'arbre syntaxique correspondant, une phase d'optimisation de l'arbre généré, et enfin l'exécution proprement dite, qui est elle-même précédée d'une recherche de positionnement optimale. C'est la phase d'exécution de l'expression régulière, et en particulier la fonction `regmatch()`, qui a été dérécursée.

En effet, jusqu'à présent, cette fonction était récursive et s'appuyait sur la pile d'appels C pour gérer la sauvegarde des états au fur et à mesure de l'exécution. Méthode facile à mettre en œuvre, mais qui présente l'inconvénient d'être coûteuse au niveau mémoire, car on empile à chaque nouvel appel tout le contexte de la fonction en cours. Dans le cas d'expressions régulières complexes, appliquées sur des données de grande taille, on peut aboutir à une saturation de la mémoire après quelques dizaines de milliers d'appels récursifs.

Cela n'est désormais plus le cas depuis que la fonction `regmatch()` a été réécrite pour gérer manuellement la récursion et ainsi ne sauver que les seules données liées à l'état de l'automate. Il a pour cela utilisé (à peu de choses près) la technique classique de transformation d'une boucle récursive en boucle itérative, mais la tâche était ici assez ardue du fait de la complexité assez élevée de ce code. Il en résulte donc un gain considérable en termes de consommation mémoire, mais aussi de temps d'exécution, étant donné que le processeur a moins de données à manipuler. Les gains sont par contre, sans surprise, surtout visibles sur des expressions complexes appliquées à des données conséquentes, quand la profondeur d'examen est suffisamment

élevée.

## 3.2 Optimisations

Yves Orton, non content d'avoir apporté de nouvelles fonctionnalités au moteur d'expressions régulières de Perl, a aussi implémenté de nouvelles optimisations, dont nous allons examiner les trois principales.

### 3.2.1 Classes de caractères

La première est que les classes de caractères composées d'un seul caractère sont maintenant remplacées par le caractère en question. Cela signifie qu'une classe comme `/[a]/` est maintenant comprise comme le seul littéral `/a/`. Cette optimisation permet donc d'éviter la mécanique liée aux classes, d'où un gain en termes de mémoire et d'exécution assez sensible, car cela interférait avec les optimisations de Boyer-Moore. On peut facilement observer un doublement de la vitesse d'exécution entre Perl 5.8 et 5.10.

### 3.2.2 Trie

La seconde est l'utilisation, quand c'est possible, d'un trie. Un trie (de l'anglais retrieval) ou prefix-tree, est une structure d'arbre ordonné utilisée pour stocker un tableau associatif (l'équivalent générique des hashes) dont les clés sont des chaînes de caractères. Cela ressemble un peu à un arbre binaire de recherche, mais contrairement à ce dernier, un trie ne stocke pas les clés dans chaque nœud, car celles-ci sont construites par le chemin parcouru pour arriver jusqu'à chaque nœud. Il en résulte un gain de place et un accès très rapide, en  $O(m)$  (avec  $m$  la longueur de la clé) pour un accès en  $O(\log N)$  pour un arbre binaire (avec  $N$  le nombre de nœuds dans l'arbre).

Une autre application des tries est de s'en servir pour déterminer le plus long préfixe commun entre deux motifs, ce qui est typiquement employé pour écrire les dictionnaires utilisés par les correcteurs d'orthographe. C'est ici utilisé comme technique d'optimisation des branches de séquences, c'est-à-dire des alternatives de motifs (`/pattern|pattern/`) : les alternatives littérales sont fusionnées en un trie et peuvent être comparées simultanément. Cela signifie que la comparaison de  $N$  alternatives s'effectue maintenant en temps constant

au lieu d'un temps d'exécution précédemment linéaire  $O(N)$ . Une nouvelle variable spéciale, `RE_TRIE_MAXBUF`, a été introduite afin de permettre de régler plus finement cette optimisation.

### 3.2.3 Aho-Corasick

Enfin, Yves a implémenté l'optimisation Aho-Corasick de placement du début de comparaison. À la base, cet algorithme construit un trie additionné de liens entre les nœuds, chaque nœud pointant vers le nœud représentant le plus long suffixe correspondant. Au sein du moteur de Perl, cela s'appuie donc sur l'optimisation précédente, quand une série d'alternatives littérales a été convertie en un trie. Le moteur cherche alors, pour chaque état dans le trie, les états d'échec, c'est-à-dire les plus longs suffixes qui sont aussi des préfixes d'autres alternatives du trie. Ainsi, dans cet exemple :

```
"abcdgu" =~ /abcdefg|cdgu/
```

quand on arrive au caractère "**d**", on est encore en train d'examiner le premier mot de l'alternative. Le caractère suivant, "**g**" va générer un état d'échec pour ce mot, mais le trie et ses liens de suffixes permettent de sauter d'une alternative à la suivante en restant au même état pour la chaîne examinée. Le moteur va donc directement mettre en correspondance le "**g**" de la chaîne avec celui du second mot, et peut continuer, conduisant au succès.

### 3.2.4 En pratique

Ces optimisations peuvent sembler un peu absconses, mais elles correspondent en pratique à des cas très couramment rencontrés. Par exemple, pensez au nombre de fois que vous avez écrit un code similaire à celui-ci pour capturer un mot parmi une liste connue :

```
my $words_str = join "|", @words;
if ($string =~ /($words_str)/) {
    # le mot capturé est dans $1
}
```

C'est typiquement le genre de cas qui s'exécutera plus rapidement grâce aux tries, et, suivant les mots à tester, l'optimisation Aho-Corasick pourra aussi entrer en jeu.

## 3.3 Moteur modulaire

Dernière nouveauté, les différents nettoyages et remaniements internes, tant de Perl en général que du moteur d'expressions régulières en particulier, ont permis d'envisager quelque chose de radicalement nouveau : la possibilité de remplacer entièrement le moteur habituel de Perl par un autre !

Ce travail a été réalisé par Yves Orton et Ævar Arnfjörð Bjarmason qui ont d'abord retravaillé le moteur de Perl pour qu'il soit le moins lié possible aux internes de Perl, et ont défini une interface en partie inspirée de l'expérience sur PCRE. Celle-ci est documentée dans `perlreapi` et permet l'utilisation de nouveaux moteurs sous la forme de plugins. Histoire de bien faire les choses jusqu'au bout, le changement de moteur se réalise à la volée et lexicalement, en s'appuyant sur le mécanisme des pragmas lexicales déjà présenté.

Bien sûr, la syntaxe reconnue et les sémantiques associées sont liées au moteur utilisé. Plusieurs plugins permettant de sélectionner des moteurs alternatifs sont d'ores et déjà disponibles sur le CPAN :

- `re::engine::PCRE` permet d'utiliser le moteur PCRE, bien connu et utilisé par un grand nombre de programmes et langages. La syntaxe est par la nature même de PCRE quasiment la même que celle du moteur natif de Perl, mais PCRE dispose encore de quelques extensions non reconnues par le moteur natif :

```
use re::engine::PCRE;
if ("Hello, world" =~ /(?<=Hello|Hi), (world)/) {
    say "OH HAI \U$1"
}
```

L'exemple utilise ici une recherche arrière de longueur variable. Si on essaye sans `re::engine::PCRE`, Perl renvoie l'erreur "~~Variable length lookbehind not implemented~~".

- `re::engine::POSIX` permet d'utiliser le moteur POSIX (IEEE Std 1003.1-2001) de votre système Unix, tel que disponible dans `regex.h`. On retrouve donc la syntaxe des expressions telles qu'on les passe à un programme comme `grep(1)` par exemple :

```
use re::engine::POSIX;
if ("mooh!" =~ /\([mo]*\)/) {
    say $1    # "moo"
```

```
}

```

**re::engine::Plan9** permet d'utiliser le moteur du feu Plan9, inclus dans la distribution du module (le moteur, pas l'OS !).

**re::engine::PCR** permet d'utiliser le moteur Pugs::Compiler::Rule, qui est un compilateur de règles (c'est-à-dire d'expressions régulières Perl 6), implémenté en pur Perl 5.

```
use re::engine::PCR;
if ("123" =~ q/$<x> := (.) $<y> := (.)/) {
    # on stocke le premier caractère dans le buffer
    # nommé "x" et le second dans le buffer "y"
    say "x=${x}, y=${y}";    # "x=1, y=2"
}

```

- **re::engine::Oniguruma** permet d'utiliser le moteur Oniguruma, employé par Ruby (depuis sa version 1.9) et par un module de PHP 5. Il a pour particularité de supporter une trentaine d'encodages de caractères différents, là où les moteurs les plus courants n'en supportent qu'un ou deux (généralement ASCII et un encodage d'Unicode comme UTF-8 ou UTF-16). Onigurama propose la même syntaxe que PCRE et Perl 5.10 à quelques parties près non supportées.

À noter que la distribution du module Perl inclut Onigurama, étant donné que ce moteur n'est pas très répandu.

- **re::engine::TRE** permet d'utiliser le moteur TRE, inclus dans la distribution du module, qui implémente une syntaxe POSIX que l'auteur prétend strictement conforme à la spécification :

```
use re::engine::TRE;
if ("the string" =~ /\(.{\3}\)/) {
    say $1    # "the"
}

```

TRE propose par ailleurs des ajouts comme le support d'une correspondance floue, en utilisant la distance de Levenshtein.

Ævar a aussi publié **re::engine::Plugin**, qui permet d'écrire un moteur en Perl, sans devoir manipuler du C/XS et sans devoir trifouiller les internes. C'est typiquement le genre de choses qui étaient auparavant réalisées de manière très expérimentale et hasardeuse dans des modules comme **Regexp::Keep** et qui sera donc maintenant beaucoup plus facile à tester. Il a d'ailleurs utilisé

re::engine::Plugin pour écrire re::engine::PCR, qui est justement en pur Perl.

## 3.4 Débogage lexical

Dernière conséquence de cette restructuration, la pragma de débogage des expressions régulières est devenue, elle aussi, lexicale :

```
use re "debug";
"moo" =~ /m[aeiouy]+/;
no re "debug";
"meh" =~ /m\w+/;
```

donne à l'exécution :

```
Compiling REX "m[aeiouy]+"
Final program:
  1: EXACT <m> (3)
  3: PLUS (15)
  4:  ANYOF[aeiouy] (0)
 15: END (0)
anchored "m" at 0 (checking anchored) minlen 2
Guessing start of match in sv for REX "m[aeiouy]+" against "moo"
Found anchored substr "m" at offset 0...
Gussed: match at offset 0
Matching REX "m[aeiouy]+" against "moo"
  0 <> <moo>   | 1:EXACT <m>(3)
  1 <m> <oo>   | 3:PLUS(15)
                ANYOF[aeiouy] can match 2 times out of 2147483647...
  3 <moo> <>   | 15: END(0)
Match successful!
Freeing REX: "m[aeiouy]+"
```

Comme on le voit, le débogage n'est activé que pour la première expression régulière. Mais il y a encore mieux :

```
use re "debug";
$reg = qr/m[aeiouy]+/;
no re "debug";
"moo" =~ $reg;
```

affiche :

```
Compiling REX "m[aeiouy]+"
Final program:
  1: EXACT <m> (3)
  3: PLUS (15)
  4:  ANYOF[aeiouy] (0)
 15: END (0)
anchored "m" at 0 (checking anchored) minlen 2
Guessing start of match in sv for REX "m[aeiouy]+" against "moo"
```

```

Found anchored substr "m" at offset 0...
Guessed: match at offset 0
Matching REX "m[aeiouy]+" against "moo"
  0 <> <moo>      | 1:EXACT <m>(3)
  1 <m> <oo>      | 3:PLUS(15)
                    ANYOF[aeiouy] can match 2 times out of 2147483647...
  3 <moo> <>      | 15: END(0)
Match successful!
Freeing REX: "m[aeiouy]+"

```

Traduction : l'expression régulière, ayant été compilée dans un contexte où le débogage est activé, conserve cet attribut même quand elle est exécutée dans un contexte où il n'est pas activé. Très utile donc quand on a besoin de vérifier comment une expression en particulier se comporte.

## 3.5 Plus d'informations

Les lecteurs désirant approfondir le sujet des expressions régulières pourront lire les pages suivantes de la documentation Perl :

- **perlre**: description détaillée des expressions régulières de Perl 5.10, <http://perldoc.perl.org/perlre.html>
- **perlretut**: tutoriel pour apprendre à maîtriser les expressions régulières de Perl, <http://perldoc.perl.org/perlretut.html>
- **perlreapi**: description de l'API de plugin du moteur d'expressions régulières, <http://perldoc.perl.org/perlreapi.html>
- **perlreguts**: description du fonctionnement interne du moteur d'expressions régulières de Perl, <http://perldoc.perl.org/perlreguts.html>

Et bien sûr, on ne saurait trop conseiller la bible en la matière, *Mastering Regular Expressions, Third Edition* de Jeffrey Friedl, publiée chez O'Reilly (ISBN 0-596-52812-4). Sa deuxième édition est disponible en français sous le titre *Maîtrise des expressions régulières*, publiée chez O'Reilly (ISBN 2-84177-236-5).

## Conclusion

Après ce tour d'horizon des nouveautés de Perl 5.10, l'auteur espère avoir convaincu ses lecteurs de mettre à jour afin de profiter des nouvelles fonctionnalités de cette version. Nous reviendrons dans un prochain article sur le futur de chaque branche de Perl, et aborderons dans un autre les techniques

permettant d'écrire du code rétro-compatible entre des versions offrant des ensembles de fonctionnalités aussi variés.

Merci aux Mongueurs de Perl et à Rafaël Garcia-Suarez pour leur relecture et leurs suggestions.

Retrouvez cet article dans : [Linux Magazine 106](#)

## Laissez une réponse

Vous devez avoir ouvert une [session](#) pour écrire un commentaire.

[Identifiez-vous](#)

[Inscription](#)

[S'abonner à UNIX Garden](#)

Catégories

[Administration réseau](#)

[Administration système](#)

[Agenda-Interview](#)

[Audio-vidéo](#)

[Bureautique](#)

[Comprendre](#)

[Distribution](#)

[Embarqué](#)

[Environnement de bureau](#)

[Graphisme](#)

[Jeux](#)

[Matériel](#)

[News](#)

[Programmation](#)

[Réfléchir](#)

[Sécurité](#)

[Utilitaires](#)

[Web](#)

## • Pages

- - [A propos](#)
  - [Nuage de Tags](#)
  - [Contribuer](#)

## • Archives

- - [avril 2011](#)
  - [mars 2011](#)
  - [février 2011](#)
  - [janvier 2011](#)
  - [décembre 2010](#)
  - [novembre 2010](#)
  - [octobre 2010](#)
  - [septembre 2010](#)
  - [août 2010](#)
  - [juillet 2010](#)
  - [juin 2010](#)
  - [mai 2010](#)
  - [avril 2010](#)
  - [mars 2010](#)
  - [février 2010](#)
  - [janvier 2010](#)
  - [décembre 2009](#)
  - [novembre 2009](#)
  - [octobre 2009](#)
  - [septembre 2009](#)
  - [août 2009](#)
  - [juillet 2009](#)
  - [juin 2009](#)
  - [mai 2009](#)
  - [avril 2009](#)
  - [mars 2009](#)
  - [février 2009](#)
  - [janvier 2009](#)
  - [décembre 2008](#)
  - [novembre 2008](#)
  - [octobre 2008](#)
  - [septembre 2008](#)
  - [août 2008](#)
  - [juillet 2008](#)
  - [juin 2008](#)
  - [mai 2008](#)
  - [avril 2008](#)
  - [mars 2008](#)
  - [février 2008](#)
  - [janvier 2008](#)
  - [décembre 2007](#)
  - [novembre 2007](#)
  - [février 2007](#)

[GNU/Linux Magazine](#)  
[GNU/Linux Magazine HS](#)  
[Linux Pratique](#)  
[Linux Pratique HS](#)  
[Linux Pratique Essentiel](#)  
[Linux Pratique Essentiel HS](#)  
[Misc](#)  
[Misc HS](#)

## • **Articles secondaires**

- 15/3/2009  
[Smart Middle Click 0.5.1 : ouvrez les liens JavaScript dans des onglets](#)

Tout d'abord, un petit raccourci utile : quand vous voulez ouvrir un lien dans un onglet, plutôt que d'utiliser le menu contextuel, cliquez simplement dessus avec le bouton du milieu. Hop, c'est ouvert ! C'est simple et diablement efficace, parfois un peu trop.....

[Voir l'article...](#)

30/10/2008

[Google Gears : les services de Google offline](#)

Lancé à l'occasion du Google Developer Day 2007 (le 31 mai dernier), Google Gears est une extension open source pour Firefox et Internet Explorer permettant de continuer à accéder à des services et applications Google, même si l'on est déconnecté....

[Voir l'article...](#)

7/8/2008

[Trois questions à...](#)

Alexis Nikichine, développeur chez IDM, la société qui a conçu l'interface et le moteur de recherche de l'EHM....

[Voir l'article...](#)

11/7/2008

[Protéger une page avec un mot de passe](#)

En général, le problème n'est pas de protéger une page, mais de protéger le répertoire qui la contient. Avec Apache, vous pouvez mettre un fichier

~~ht~~access dans le répertoire à protéger...

[Voir l'article...](#)

6/7/2008

[hypermail : Conversion mbox vers HTML](#)

Comment conserver tous vos échanges de mails, ou du moins, tous vos mails reçus depuis des années ? mbox, maildir, texte... les formats ne manquent pas. ...

[Voir l'article...](#)

6/7/2008

[iozone3 : Benchmark de disque](#)

En fonction de l'utilisation de votre système, et dans bien des cas, les performances des disques et des systèmes de fichiers sont très importantes....

[Voir l'article...](#)

Les Éditions Diamond - Tous droits réservés.