

Les regex et grammaires de Perl 6

Une puissance expressive sans précédent



Par [Laurent Rosenfeld](#)

Date de publication : 6 novembre 2015

Dernière mise à jour : 23 juillet 2016

DÉBUTANT

La puissance des expressions régulières de Perl 5 a longtemps fait de ce langage l'instrument de choix par excellence pour analyser des données textuelles. Depuis, de nombreux autres langages de programmation ont copié les expressions régulières de Perl, ce qui a en partie atténué cet avantage que Perl avait sur d'autres langages dans ce domaine.

Le nouveau langage Perl 6, dérivé de Perl 5, crée un nouveau modèle de reconnaissance textuelle dérivé des expressions régulières, mais considérablement plus puissant et plus expressif, et si éloigné des expressions régulières d'origine qu'il a été décidé de leur donner un nouveau nom, les *regex*.

Non seulement le mécanisme des regex de Perl 6 est-il considérablement plus puissant que tous les systèmes d'expressions régulières existants, mais il a été conçu de façon à pouvoir combiner des regex pour construire des *grammaires* contextuelles, c'est-à-dire des systèmes capables de réaliser l'analyse lexicale et syntaxique (*lexing* et *parsing*) de données beaucoup plus complexes, comme des textes HTML, XML, XHTML, JSON, YAML, qui, hors cas triviaux, sont hors de portée des expressions régulières. Ces grammaires peuvent notamment analyser des programmes informatiques de tous niveaux. Un programme Perl 6 est lui-même compilé avec sa propre grammaire écrite en Perl 6.

Même si elles sont loin d'être la seule innovation de Perl 6, nous pensons que les regex et les grammaires de Perl 6 vont révolutionner les langages informatiques au moins aussi profondément, et peut-être beaucoup plus, que les expressions régulières de Perl 5 ne l'avaient fait en leur temps.

Une discussion sur ce tutoriel est ouverte sur le forum Perl à l'adresse suivante :
Commentez

En complément sur Developpez.com

- [De Perl 5 à Perl 6- Partie 1 : les bases](#)
- [De Perl 5 à Perl 6 - Partie 2 : les nouveautés](#)
- [De Perl 5 à Perl 6: approfondissements](#)
- [De Perl 5 à Perl 6 - Annexe 1 : ce qui change entre Perl 5 et Perl 6](#)
- [De Perl 5 à Perl 6 - Annexe 2 : Les nouveautés de Perl 6](#)

1 - Petite introduction aux expressions régulières.....	5
1-1 - Une analogie avec la recherche de fichiers.....	5
1-2 - Bref historique des expressions régulières.....	5
1-3 - Ce qui est nouveau avec les regex de Perl 6.....	6
1-4 - Les opérateurs, fonctions et méthodes associés aux regex.....	7
2 - Les regex de Perl 6.....	9
2-1 - Conventions lexicales.....	9
2-2 - Littéraux.....	10
2-3 - Métacaractères et classes de caractères.....	11
2-3-1 - Caractère d'échappement et classes de caractères prédéfinies.....	11
2-3-2 - Propriétés Unicode.....	11
2-3-3 - Classes de caractères énumérées et intervalles.....	13
2-4 - Quantificateurs.....	13
2-4-1 - Avidité et frugalité des quantificateurs.....	14
2-5 - Alternatives (reconnaître ceci ou cela).....	14
2-5-1 - Conjonction (reconnaître ceci et cela).....	15
2-6 - Ancres.....	15
2-7 - Regroupements et captures.....	16
2-7-1 - Regroupements.....	16
2-7-2 - Captures.....	16
2-7-3 - Regroupements sans capture.....	17
2-7-4 - Numérotation des captures.....	17
2-7-5 - Captures nommées.....	17
2-8 - Sous-règles ou règles nommées.....	18
2-9 - Adverbes.....	19
2-9-1 - Adverbes de regex.....	20
2-9-1-1 - L'adverbe « ratchet » : pas de retour arrière.....	20
2-9-1-2 - L'adverbe « sigspace » (espaces blancs significatifs).....	21
2-9-1-3 - L'adverbe « Perl5 » pour retrouver la syntaxe de Perl 5.....	22
2-9-2 - Les adverbes de reconnaissance.....	23
2-9-2-1 - Adverbe « continue » (position de départ de la reconnaissance).....	23
2-9-2-2 - Adverbe « exhaustive » (toutes les reconnaissances).....	23
2-9-2-3 - Adverbe « global » (chaque reconnaissance).....	23
2-9-2-4 - Adverbe « pos ».....	24
2-9-2-5 - Adverbe « overlap » (avec chevauchement).....	24
2-10 - Regarder devant et derrière (assertions).....	24
2-10-1 - Assertions avant.....	24
2-10-2 - Assertions arrière.....	25
3 - Grammaires.....	25
3-1 - Les « briques » de construction d'une grammaire.....	25
3-2 - Créer une grammaire.....	26
3-2-1 - Syntaxe de définition d'une grammaire.....	26
3-2-2 - Héritage de grammaires.....	27
3-3 - Utiliser une grammaire.....	28
3-4 - Les classes et objets d'actions.....	28
3-4-1 - Exécuter du code lors d'une reconnaissance.....	28
3-4-2 - Autres façons d'exécuter du code dans une grammaire.....	30
3-5 - Une grammaire pour valider des noms de modules Perl.....	31
3-5-1 - La grammaire de validation.....	31
3-5-2 - Ajout d'un objet d'actions.....	32
3-6 - Une grammaire pour analyser du JSON.....	33
3-6-1 - Structure d'un document JSON.....	33
3-6-2 - Exemple de document JSON.....	33
3-6-3 - Écrire pas à pas les éléments de la grammaire JSON.....	34
3-6-3-1 - Les nombres.....	34
3-6-3-2 - Les chaînes de caractères.....	34
3-6-3-3 - Les objets JSON.....	35
3-6-3-4 - Les tableaux JSON.....	35

3-6-3-5 - Les valeurs.....	35
3-6-4 - La grammaire JSON.....	36
3-6-5 - Ajouter des actions.....	37
3-7 - Une grammaire pour analyser du (pseudo) XML.....	38
3-8 - Grammaires : concepts avancés et perspectives.....	39
3-8-1 - Les règles paramétrées.....	39
3-8-2 - Règles récursives et variables dynamiques.....	39
3-8-3 - Les règles nommées de type proto.....	41
3-8-4 - Héritage et grammaires mutables.....	41
3-8-5 - Modifications de la grammaire et extensibilité du langage.....	42
3-8-6 - Perspectives.....	42
4 - Bonnes pratiques et pièges à éviter.....	42
4-1 - Formatage du code.....	43
4-2 - Limiter la taille.....	43
4-2-1 - Reconnaître un nombre à virgule flottante.....	43
4-2-2 - Reconnaître un nombre complexe.....	44
4-2-3 - Reconnaître une URL.....	44
4-2-4 - Reconnaître une adresse IP.....	45
4-2-5 - Une grammaire pour reconnaître une URL.....	46
4-3 - Que reconnaître ?.....	46
4-4 - Reconnaître des espaces blancs.....	47
4-5 - Déboguer des regex ou des grammaires Perl 6.....	48
5 - Conclusion.....	49
6 - Voir aussi/Sources.....	49
7 - Remerciements.....	50

1 - Petite introduction aux expressions régulières

Les *expressions régulières* (ou *expressions rationnelles*) sont un concept issu de la théorie mathématique puis informatique des langages formels dans lequel une chaîne de caractères généralement appelée *motif* (ou *pattern*) permet de décrire tout un ensemble (fini ou non) de chaînes de caractères ayant des traits communs définis par le motif, selon une syntaxe prédéfinie et sans tenir compte du contexte. La reconnaissance de motifs (*pattern matching*) est le processus consistant à appliquer ces motifs à des échantillons de texte de façon à essayer d'y retrouver des fragments de texte correspondant à ces motifs.

Installer Perl 6 sur votre ordinateur

Si vous désirez utiliser Perl 6, nous vous conseillons de télécharger Rakudo Star à [cette adresse](#). Des informations complémentaires relatives à l'installation sont disponibles dans la première partie du tutoriel [De Perl 5 à Perl 6 - Partie 1 : les bases du langage](#).

Ce document évoquait à l'époque (2014) la possibilité de choisir entre deux machines virtuelles (Parrot et MoarVM). À l'heure où nous écrivons (octobre 2015), il est fortement recommandé de choisir MoarVM, qui offre un Perl 6 presque complet et bien plus abouti.

1-1 - Une analogie avec la recherche de fichiers

Pour rechercher dans un répertoire tous les fichiers dont le nom commence par la lettre « a » et dont l'extension est .txt, on peut écrire à l'invite du système :

```
ls a*.txt      # shell Unix, Linux, etc.
# ou :
dir a*.txt     # console DOS/cmd ou Powershell Windows, VMS, etc.
```

Le nom de la commande employée (ls ou dir) diffère selon le système d'exploitation, mais le *motif* employé se trouve être ici le même : a*.txt. Il signifie plus précisément quelque chose du genre : lettre « a », suivie d'un nombre quelconque de caractères quelconques, suivis de la chaîne de caractères « .txt ».

La commande employée affichera à l'écran tous les fichiers du répertoire courant dont le nom respecte le motif a*.txt. Cette façon de filtrer les noms des fichiers du répertoire (ou *dossier* en terminologie Dos/Windows) avec un motif est le principe de fonctionnement des expressions régulières : le motif décrit, généralement de gauche à droite, une série d'éléments que l'on doit rencontrer dans la chaîne de caractères cible (ici le nom de fichier) : d'abord la lettre « a », puis des caractères quelconques, puis la chaîne « .txt ».

L'analogie s'arrête là, car la syntaxe des expressions régulières communément employées dans les langages de programmation donne un sens différent au métacaractère « * » (voir § 2.4.).

1-2 - Bref historique des expressions régulières

Les premières utilisations d'expressions régulières en informatique ont été mises en œuvre dans les années 1970 par Ken Thompson, l'un des créateurs du système Unix, dans les éditeurs qed et ed (qui ne sont plus guère employés) et dans la commande Unix grep, encore très utilisée de nos jours.

Par exemple, sous Unix ou Linux, la commande grep suivante permet d'afficher toutes les lignes du fichier texte.txt qui contiennent les lettres « ab », suivies d'un caractère quelconque, suivies de la lettre « d » :

```
$> grep ab.d texte.txt
```

Dans le motif `ab.d`, le métacaractère « `.` » signifie « un seul caractère quelconque », si bien que la commande ci-dessus pourra par exemple afficher les lignes suivantes du fichier :

```
abcd
abvd ...
... abyd
... xyabcdz ...
xyab3dgh
```

D'autres utilitaires Unix encore très employés de nos jours et utilisant aussi les expressions régulières ont rapidement vu le jour : `sed`, `vi`, `awk`, `lex`, `emacs`, `egrep`, etc.

Le langage Perl (Perl 1 est apparu en décembre 1987) est sans doute le premier langage de programmation généraliste (hormis `awk`, qui n'est pas un langage généraliste et qui est assez particulier) à avoir intégré les expressions régulières, suivi plus tard par d'autres.

Très rapidement, Perl (en particulier Perl 5 depuis 1994) a considérablement étendu ses expressions régulières, au point qu'elles ont cessé depuis bien longtemps d'être « régulières » ou « rationnelles » au sens d'origine strict du terme, mais offrent en revanche un surcroît d'expressivité tel qu'elles ont fini par être copiées par bon nombre de langages de programmation tels que Tcl, Python, PHP, Ruby, .NET, Java, JavaScript, Delphi, etc. L'influence de Perl 5 sur ces langages est telle qu'ils utilisent pour la plupart une bibliothèque appelée « Perl Compatible Regular Expressions » (PCRE) : bref, pour attirer l'utilisateur, il fallait mettre en avant le fait que la solution proposée suivait la syntaxe des « expressions régulières étendues » de Perl.

1-3 - Ce qui est nouveau avec les regex de Perl 6

Avec Perl 6, les expressions régulières utilisent toujours la même méthode générale de reconnaissance progressive de gauche à droite d'un motif, mais elles sont encore moins régulières (au sens strict des langages formels non contextuels d'origine) et encore plus puissantes que celles de Perl 5, si bien qu'il a été décidé d'abandonner le terme « expressions régulières » pour les désigner et de les nommer *regex*. L'ajout au fil du temps de nouvelles fonctionnalités avait fini par rendre la syntaxe des expressions régulières de Perl 5 quelque peu touffue, en grande partie parce que Perl 5 s'est toujours attaché à préserver dans toute la mesure du possible la rétrocompatibilité.

Les regex de Perl 6 ne sont plus celles de Perl 5

Les expressions régulières de Perl (en particulier de Perl 5) ont considérablement influencé de nombreux autres langages de programmation, au point qu'elles sont devenues de facto une norme communément adoptée (comme en témoigne la bibliothèque PCRE).

Attention, bien qu'elles conservent de nettes ressemblances, les regex de Perl 6 ont été refondues et ne respectent plus ce standard de fait établi par Perl 5. Cette refonte a permis de les rendre plus claires et surtout bien plus puissantes. Il en résulte cependant que, contrairement à par exemple PCRE, les regex de Perl 6 ne sont donc pas compatibles avec celles de Perl 5.

L'avenir dira si cette décision assez radicale de rompre avec la norme sera suivie par d'autres langages et si les regex de Perl 6 deviendront à leur tour une norme de fait. Nous pensons qu'elles le méritent amplement.

Remarque : il est cependant possible d'utiliser l'adverbe `:P5` ou `:Perl5` pour utiliser la syntaxe des expressions régulières de Perl 5 dans un programme Perl 6 (voir § 2.9.1.3.). On retrouve alors des expressions régulières compatibles avec Perl 5.

Perl 6 ayant décidé d'être un nouveau langage et d'abandonner cette exigence de rétrocompatibilité, cela a permis de refondre assez profondément le langage des regex, de l'élargir considérablement et de le rendre plus cohérent et plus logique.

De plus, de nombreux exemples ci-dessous montreront comment il est possible de *construire des regex simples et de leur donner un nom afin de pouvoir ensuite les assembler* comme s'il s'agissait des briques d'un jeu de construction pour former des motifs de recherche de plus en plus complexes. Par exemple, une adresse IP v4 se compose de quatre octets (généralement représentés par quatre nombres entiers compris, en notation décimale, entre 0 et 255, séparés par des points. Il est très facile en Perl 6 d'écrire une première regex, que l'on nommera par exemple *octet* et qui vérifiera que l'on a bien un nombre satisfaisant les conditions voulues, puis d'assembler une nouvelle regex reconnaissant quatre de ces octets, séparés par des points (voir § [4.2.4.](#)).

Mais la véritable révolution apportée par les regex de Perl 6 est qu'elles permettent de construire des entités encore plus puissantes, *les grammaires* (voir § [3.](#)). Une grammaire est un formalisme permettant de définir une syntaxe et, donc, un langage formel. En Perl 6, elle se présente sous la forme d'un groupe de règles et de regex nommées et permet de construire progressivement un système de reconnaissance de texte beaucoup moins bien structuré que ce que l'on peut reconnaître avec les expressions régulières (même étendues) de Perl 5. Une grammaire permet donc de procéder à l'analyse lexicale et syntaxique d'un texte, par exemple du code source d'un programme informatique en vue de le compiler. Perl 6 possède sa propre grammaire de Perl 6, écrite en Perl 6.

Les expressions régulières ont parfois la réputation d'être abstraites et difficiles à comprendre. Les expressions régulières sont construites par assemblage de concepts simples qui ne sont pas plus compliqués à comprendre que les conditions `if` et les boucles `while` ou `for` du langage Perl lui-même. En fait, le véritable enjeu dans l'apprentissage des expressions régulières réside dans la compréhension de la notation très concise, voire laconique, souvent utilisée pour exprimer ces concepts. En fait, Perl 6 permet de simplifier considérablement la compréhension des expressions régulières en offrant la possibilité d'insérer des espaces, des commentaires, etc. Il est toujours possible de continuer à écrire des regex très concises et parfois un peu difficiles à déchiffrer, et c'est parfois suffisant pour des problèmes simples, mais personne n'y est obligé. La différence est flagrante dans l'exemple du chapitre [Formatage du code](#) vers la fin de ce document.

1-4 - Les opérateurs, fonctions et méthodes associés aux regex

Bien que ce ne soit pas réellement l'objet du présent tutoriel, il est utile de rappeler brièvement quelques opérateurs Perl 6 utilisant les regex.

En Perl 6, l'opérateur de base pour vérifier si une chaîne de caractères correspond à un motif est l'opérateur `~~` de reconnaissance intelligente (*smart match operator*). Par exemple :

```
say "Reconnu" if "abcdef" ~~ /ab.d/; # -> Affiche : Reconnu
```

Ici, la chaîne de caractères à analyser est « `abcdef` » et le motif de la regex `/ab.d/`. Le motif est reconnu, parce que l'on peut faire une correspondance entre les quatre atomes du motif (le point « `.` » reconnaît un caractère quelconque, donc, ici, il reconnaît « `c` ») et une partie de la chaîne de caractères ; le motif décrit en quelque sorte cette partie de la chaîne. Bien sûr, le motif `/ab.d/` aurait aussi reconnu, par exemple, les chaînes « `abwdef` », « `ab7def` » ou « `suabZda` ».

À noter que l'opérateur de reconnaissance intelligente `~~` utilisé ici pour lier une chaîne de caractères et un motif de regex sert à beaucoup d'autres choses en Perl 6 (par exemple vérifier la présence d'un élément dans un tableau, ou vérifier la compatibilité d'une variable avec un type, etc.), mais ce n'est pas l'objet du présent document (voir par exemple [Opérateur de reconnaissance intelligente](#)).

Si la chaîne de caractères à analyser est stockée dans la variable par défaut `$_`, alors la présence de l'opérateur de reconnaissance intelligente n'est pas nécessaire et la regex peut être évaluée directement en contexte booléen :

```
if / ^ab / {  
    say "La chaîne $_ commence par les lettres 'ab'";  
}
```

```
}
```

Il est possible d'utiliser la forme négative `!~` de l'opérateur de reconnaissance intelligente :

```
say "Chaîne 'ab' non trouvée" if "fedcba" !~ /ab/;
# -> Chaîne 'ab' non-trouvée
# Équivalent à :
say "Chaîne 'ab' non trouvée" unless "fedcba" ~~ /ab/;
```

Perl 6 autorise également une syntaxe de méthode orientée objet avec la méthode `.match` :

```
say "Reconnu" if "abdcef".match(/c.f/); # -> Reconnu
```

Les regex permettent également d'effectuer des substitutions :

```
my $chaîne = "abcde";
$chaîne ~~ s/bc/CB/;
say $chaîne; # -> affiche aCBde
```

Il existe également une méthode `.subst` permettant d'effectuer une substitution (mais pas « en place » comme le permet l'opérateur `s///`) :

```
my $chaîne = "abcde";
my $chaîne-modifiée = $chaîne.subst(/cd/, "DC"); # -> abDCe
```

Il est cependant possible de modifier directement la variable `$chaîne` (sans créer de nouvelle variable) en la plaçant également dans la partie gauche de l'affectation :

```
my $chaîne = "abcde";
$chaîne = $chaîne.subst(/cd/, "DC") # -> abDCe
```

Le premier argument de la méthode `.subst` peut être soit une regex, soit une chaîne de caractères.

La fonction `split` et la méthode `.split` qui divisent une chaîne de caractères en une liste de sous-chaînes en fonction d'un motif de coupure, peuvent également utiliser soit une chaîne, soit une regex :

```
say split(';', "a;b;c;d").perl; # ("a", "b", "c,d").Seq
say split(/\;/, "a;b;c;d").perl; # ("a", "b", "c,d").Seq
say split(<[;,]>, "a;b;c;d").perl; # ("a", "b", "c", "d").Seq
# Version syntaxe de méthode orientée objet :
say "a;b;c;d".split(<[;,]>).perl; # ("a", "b", "c", "d").Seq
```

La fonction `comb` et la méthode `.comb` renvoient une liste de reconnaissances (gourmandes) d'un motif sur une chaîne :

```
say join " ", comb /\d+/, "jeu du 7, 14 et 21"; # -> 7 14 21
# syntaxe de méthode :
say "3 fois 6 font 18".comb(/\d+/).join(" "); # -> 3 6 18
```

Sans être spécifiques aux regex, d'autres fonctions ou méthodes établissant une condition booléenne peuvent utiliser un motif de regex (ou autre chose, par exemple une comparaison numérique) pour définir cette condition. Ainsi, la fonction `first`, qui renvoie le premier élément d'une liste satisfaisant une condition, et la fonction `grep`, qui renvoie tous les éléments d'une liste satisfaisant une condition, peuvent s'écrire avec une regex :

```
say first /ma/, <jan fév mar avr mai>; # -> mar
say grep /ma/, <jan fév mar avr mai>; # -> (mar mai)
say <jan fév mar avr mai>.grep(/v/); # -> (fév avr)
```

La construction `given ... when` (le « switch » de Perl 6) utilise également bien souvent des regex :


```
my $var = '42';
given $var {
  when /^4/ { say "Commence par '4'"; proceed};
  when /2$/ { say "Finit par '2'"; proceed};
  when /^42$/ { say "Réponse à la Grande Question sur l'Univers" }
}
```

2 - Les regex de Perl 6

2-1 - Conventions lexicales

Perl 6 offre les constructions syntaxiques suivantes pour écrire des regex :

```
m/abc/;      # une regex immédiatement appliquée à $_
rx/abc/;     # un objet de type Regex
/abc/;       # un objet de type Regex
```

Les deux premières syntaxes peuvent utiliser d'autres délimiteurs que la barre oblique :

```
m{abc};      # ou m[abc];
rx{abc};     # ou rx!abc!;
```

À noter cependant que le caractère deux-points (« : ») ne peut pas servir de délimiteur pour des regex. Les parenthèses ordinaires (« (» et «) ») ne peuvent servir de délimiteur que si la parenthèse ouvrante est séparée par au moins un espace de l'opérateur `rx` qui le précède (c'est une règle générale de Perl 6 : une parenthèse ouvrante accolée à l'identifiant qui précède est analysée comme le début d'un appel de fonction) :

```
my $regex = rx(toto); # ERRONÉ, interprété comme appel de fonction
my $regex = rx (toto); # OK
```

D'une façon générale, les espaces blancs à l'intérieur des motifs sont ignorés par défaut, sauf en cas de l'utilisation (explicite ou implicite) de l'adverbe `:s` ou `:sigspace`, voir § 2.9.1.2. plus bas) .


```
say "Reconnu" if "abc" ~~ /a b c /; # -> "Reconnu"
```

Comme dans le reste de Perl 6, les commentaires commencent habituellement avec le caractère *dièse* (« # »), sauf si ce dièse est protégé par le caractère d'échappement « \ », et vont jusqu'à la fin de la ligne (et sauf si le dièse est utilisé comme délimiteur, auquel cas il est nettement préférable de ne pas essayer de s'en servir comme caractère de début de commentaire). Les **commentaires multilignes** sont également possibles.

```
# Commentaires unilignes :
my $regex = rx {
    abc      # chaîne littérale 'abc'
    \d       # suivie d'un chiffre
    \w       # puis d'un caractère alphanumérique.
};

# Commentaire multiligne :
my $regex = rx {
    abc \d \w #'[ chaîne littérale 'abc' suivie d'un
              chiffre puis d'un caractère
              alphanumérique quelconque.
              ]
};
say "Reconnu" if "Xyabc6QUVW" ~~ /$regex/; # reconnaît 'abc6Q'
```



La Synopse (spécification d'origine)  **S05** semble indiquer que les variables ne sont en principe pas interpolées dans une regex, ou du moins pas de la façon où l'on entend ce terme

en Perl 5. Dans la pratique, Perl 6 / Rakudo Star effectue actuellement (au moins dans la plupart des cas) l'interpolation des variables au sein des regex :

```
my $var = "ab";
say "Reconnu" if "abc" ~~ /$var c/; # -> Reconnu
say ~$/; # -> abc
```

La documentation officielle actuelle est pour l'instant muette sur ce point et le comportement exact reste (un peu) incertain et à préciser. Il semble qu'il faille comprendre que la variable n'est pas interpolée au moment de la construction du motif, mais seulement au moment de son utilisation lors de la reconnaissance proprement dite, ce qui permet d'utiliser une variable non encore initialisée dans la définition du motif :

```
my $var;
my $regex = rx/q.$var/;
# pas d'erreur bien que $var ne soit pas définie
$var = rx/abc/;
say "Vrai" if 'aaqabchg' ~~ $regex; # -> Vrai
```

2-2 - Littéraux

Le cas le plus simple de motif de reconnaissance d'une regex est une chaîne constante. Dans ce cas, reconnaître un motif consiste à rechercher le motif comme une sous-chaîne de la chaîne :

```
my $chaîne = "Esperluette est le nom parfois donné au signe &";
if $chaîne ~~ m/ perl / {
    say "$chaîne contient 'perl'"; # -> esperluette contient "perl"
}
```

Tous les caractères alphanumériques (Unicode) et le caractère souligné ou *underscore* (« `_` ») sont des reconnaissances littérales. Tous les autres caractères (signes de ponctuation, symboles, etc.) doivent être protégés par le caractère d'échappement *antislash* (« `\` ») ou être cités entre apostrophes (ou guillemets simples) :

```
/ 'deux mots' / # reconnaît 'deux mots', espace blanc compris
/ "a:b" / # reconnaît 'a:b', caractère deux-points compris
/ '#' / # reconnaît le caractère dièse (ou hash)
/moi\@gmail\.com/ # échappements pour protéger l'@ et le .
/'moi@gmail.com'/ # équivalent à : /moi\@gmail\.com/
```

Lorsqu'ils sont protégés par un caractère d'échappement, les caractères alphanumériques prennent souvent une signification particulière : par exemple, le métacaractère `\d` représente une classe de caractères pouvant signifier un chiffre quelconque (Unicode) ; de nombreux exemples seront donnés plus loin (notamment au § 2.3.1.).

Les chaînes de caractères sont explorées de gauche à droite, il suffit donc, par exemple, qu'une sous-chaîne soit égale au motif :

```
if 'abcdefg' ~~ / de / {
    say ~$/; # de -> motif reconnu
    say $/.prematch; # abc -> ce qui précède le motif reconnu
    say $/.postmatch; # fg -> ce qui suit le motif reconnu
    say $/.from; # 3 -> position du début de la reconnaissance
    say $/.to; # 5 -> position de ce qui suit la reconnaissance
};
```

Les résultats de la reconnaissance sont stockés dans la variable `$/` (représentant le *match object*, que l'on traduira dans ce document par « objet reconnu », même si le terme anglais veut parfois aussi dire « objet de type Match »), et sont également renvoyés par la reconnaissance. Le résultat est de type Match si la reconnaissance a réussi, et Nil sinon.

2-3 - Métacaractères et classes de caractères

Une *classe de caractères* est un élément de syntaxe des regex qui permet de reconnaître non plus un seul caractère déterminé, mais un caractère appartenant à tout un ensemble de caractères ayant éventuellement des traits communs (reconnaître par exemple l'un quelconque des chiffres de 0 à 9, ou l'un quelconque des caractères alphabétiques minuscules).

Le point (« . ») reconnaît tout caractère simple (sauf s'il est précédé d'un caractère d'échappement, auquel cas il reconnaît un point littéral) :

```
'perl'  ~~ /per./;      # Reconnaît toute la chaîne
'perl'  ~~ / per . /;    # Idem (espaces blancs ignorés);
'perl'  ~~ / pe.l /;     # Idem: le . reconnaît le r
'Épelle' ~~ / pe.l /;    # Idem: le . reconnaît le premier l

'perl'  ~~ /. per /      # Pas de reconnaissance:
                        # le . ne reconnaît rien avant la chaîne per
```

Contrairement à Perl 5 et à de nombreux langages dont le système d'expressions régulières dérive de Perl 5, le point reconnaît aussi toujours le caractère retour à la ligne.

2-3-1 - Caractère d'échappement et classes de caractères prédéfinies

Il existe des classes de caractères prédéfinies ayant la forme de l'antislash (ou barre oblique inverse) suivi d'une lettre, par exemple `\w`. Si la lettre est en majuscule (`\W`), c'est la négation de la classe de caractères correspondant à la même lettre en minuscule (autrement dit, `\W` reconnaît tout caractère non reconnu par `\w`) :

- **caractère alphanumérique** (lettres, chiffres et `_`) : `\w` (complément : `\W`) ; reconnaît par exemple `a`, `C`, `z`, `7` et les caractères *Unicode* 0041 A LATIN CAPITAL LETTER A, 0031 1 DIGIT ONE, 03B4 δ GREEK SMALL LETTER DELTA ou 0409 Љ CYRILLIC CAPITAL LETTER LJE) ;
- **caractère numérique** : `\d` et `\D` (chiffre unique, au sens Unicode de chiffre, pas seulement nos chiffres arabes : par exemple, U+0E53 # THAI DIGIT THREE (chiffre thaïlandais 3) est reconnu par `\d`) ;
- **espace horizontal** : `\h` et `\H` (espaces blancs, tabulations, U+00A0 NO-BREAK SPACE) ;
- **espace vertical** : `\n` et `\N` ;
- **espace (horizontal ou vertical)** : `\s` et `\S` ; par exemple, dans la chaîne 'Contient un mot commençant par m', l'expression `/ m \S+ /` reconnaît 'mot'.
- **tabulation** (U+0009) : `\t` et `\T` ;
- **espace vertical** : `\v` et `\V` (par exemple U+000A LINE FEED, U+000C CARRIAGE RETURN, etc.).

2-3-2 - Propriétés Unicode

Les classes de caractères vues ci-dessus sont pratiques pour des cas courants. L'utilisation des propriétés Unicode permet une approche plus systématique et plus fine. La syntaxe d'appel est de la forme `<:propriété>`, dans laquelle « propriété » peut être un nom court ou long de propriété Unicode. Le sens précis des propriétés Unicode elles-mêmes n'est pas défini par Perl, mais par les normes Unicode.

Voici une liste des propriétés Unicode les plus courantes :

Nom court	Nom long	Signification et remarques
L	Letter	Lettre
LC	Cased_Letter	Lettre avec sa casse (distinction capitale/minuscule)
LU	Upper_Cased_Letter ou Upper	Lettre capitale (en majuscule)
LL	Lower_Cased_Letter ou Lower	Lettre bas de casse (en minuscule)
N	Number	Nombre
Nd	Decimal_Number ou Digit	Nombre décimal (chiffre)
NI	Letter_Number	Nombre lettre
P	Punctuation ou Punct	Signe de ponctuation
Pd	Dash_Punctuation	Ponctuation de type tiret
Ps	Open_Punctuation	Ponctuation ouvrante
Pe	Close_Punctuation	Ponctuation fermante
S	Symbol	Symbole
Sm	Math_Symbol	Symbole mathématique
Sc	Currency_Symbol	Symbole monétaire (par ex. \$, £ ou €).
Z	Separator	Séparateur
Zs	Space_Separator	Séparateur espace
Zl	Line_Separator	Séparateur ligne
Zp	Paragraph_Separator	Séparateur paragraphe

Par exemple, `<:Lu>` reconnaît une seule lettre capitale (majuscule).

La négation d'une propriété Unicode est obtenue avec la forme `<:!propriété>`, par exemple `<:!Lu>` reconnaîtra tout caractère unique qui n'est pas une lettre capitale.

Il est possible de combiner plusieurs propriétés à l'aide des opérateurs infixés suivants :

Opérateur	Signification	Remarque
+	Union ensembliste	<i>ou</i> logique (<i>or</i>) entre les propriétés
	Union ensembliste	<i>ou</i> logique (<i>or</i>) entre les propriétés
&	Intersection ensembliste	<i>et</i> logique (<i>and</i>) entre les propriétés
-	Différence ensembliste	Ayant la première propriété et pas la seconde
^	Intersection symétrique ensembliste	<i>ou exclusif</i> logique (<i>XOR</i>) entre les propriétés

Par exemple, pour reconnaître soit une lettre minuscule soit un nombre, il est possible d'écrire : `<:Ll+:N>` ou `<:Ll+:Number>` ou encore `<+:Lowercase_Letter+:Number>`.

Il est également possible de grouper des catégories et des ensembles de catégories avec des parenthèses, par exemple :

```
'perl6' =~ m{\w+(<:Ll+:N>)} # 0 => #6#
```

2-3-3 - Classes de caractères énumérées et intervalles

Parfois, les métacaractères et classes de caractères prédéfinies ne suffisent pas. Il est heureusement simple de définir sa propre classe de caractères en plaçant entre `<...>` un nombre quelconque de caractères et d'intervalles de caractères (avec deux points « .. » entre les bornes de ces intervalles), avec ou sans espaces blancs :

```
"abacabadabacaba" ~~ / <[ a .. d 1 2 3 ]> / # Vrai
```

Il est possible d'utiliser à l'intérieur des `<...>` les mêmes opérateurs que pour les catégories Unicode (+, |, &, -, ^) pour combiner de multiples définitions d'intervalles ou même de les mélanger avec les catégories Unicode ci-dessus ; on peut également utiliser entre les crochets les classes de caractères définies avec les antislashes : `/ <[\d - [13579]> /`, ce qui, soit dit en passant, n'est pas la même chose que `/ <[02468]> /`, car la première reconnaît aussi les chiffres non arabes.

La négation d'une classe de caractères de ce type s'obtient avec le signe « - » après le chevron ouvrant :

```
say 'pas de guillemets' ~~ / <-[ " ]> + /;  
# reconnaît les caractères autres que "
```

Il est assez commun, pour analyser des chaînes délimitées par des guillemets, d'utiliser un motif utilisant des négations de classes de caractères :

```
say 'entre guillemets' ~~ / "'" <-[ " ]> * "'/;  
# un guillemet, suivi de non-guillemets, suivi d'un guillemet
```

2-4 - Quantificateurs

Un quantificateur permet de reconnaître non pas exactement une fois, mais plutôt un nombre fixe ou variable de fois, l'atome qui le précède. Par exemple, le quantificateur « + » cherche à reconnaître une ou plusieurs fois ce qui précède.

Les quantificateurs ont une précedence plus forte que la concaténation, si bien que `/ ab+ /` reconnaît la lettre a suivie d'une ou plusieurs fois la lettre b. La situation est inversée avec des apostrophes : `/ 'ab'+ /` reconnaît les chaînes 'ab', 'abab' 'ababab', etc.

Quantificateur	Signification	Remarques ou exemples
+	Un ou plusieurs	Reconnaît l'atome précédant une ou plusieurs fois, sans limite supérieure
*	0 ou plusieurs fois	Par exemple, pour autoriser un espace ou plusieurs espaces optionnel(s) entre a et b : <code>/ a \s* b /</code>
?	0 ou 1 fois	Par exemple pour un caractère optionnel unique
** min..max	Nombre arbitraire de fois entre min et max	<code>say Bool('a' ~~ /a ** 2..5/); #-> False</code> <code>say Bool('aaa' ~~ /a ** 2..5/); #-> True</code>
** n	Exactement n fois	<code>say Bool('aaaa' ~~ /a ** 5/); #-> False</code> <code>say Bool('aaaaa' ~~ /a ** 5/); #-> True</code>
%	Quantificateur modifié	Pour faciliter le travail avec les CSV, le modificateur %, appliqué à l'un des quantificateurs ci-dessus, permet de spécifier un séparateur qui doit être présent entre les reconnaissances répétées. Par exemple : <code>/ a+ % ', ' /</code> reconnaît 'a,a' ou 'a,a,a', etc., mais ni 'a,', ni 'a,a,'.

2-4-1 - Avidité et frugalité des quantificateurs

Par défaut, les quantificateurs `+` et `*` sont *avides* ou *gourmands*, c'est-à-dire qu'ils cherchent la reconnaissance la plus longue possible dans la chaîne. Par exemple :

```
say ~$/ if 'aabaababa' ~~ /.+ b /;      # -> aabaabab
```

Ici, la sous-regex `.+` recherche la plus longue chaîne possible de caractères quelconques permettant encore de reconnaître la suite de la regex, en l'occurrence l'atome `b`, ce qui peut être le but recherché. Mais il arrive assez fréquemment que cela soit une erreur de débutant et que l'objectif soit plutôt de reconnaître « des caractères quelconques jusqu'au premier `b` ». Dans ce cas, on préférera utiliser un quantificateur *non gourmand* (ou « frugal »), obtenu en postfixant le quantificateur d'origine avec un point d'interrogation, ce qui donne soit `+`, soit `*`. Par exemple :

```
say ~$/ if 'aabaababa' ~~ /.+? b /;      # -> aab
```

2-5 - Alternatives (reconnaître ceci ou cela)

Pour reconnaître une possibilité parmi plusieurs, il faut les séparer par « `||` » ; la première reconnaissance trouvée (de gauche à droite) l'emporte. Par exemple les fichiers d'initialisation (du style `config.ini`) ont souvent la forme suivante :

```
[section]
clef = valeur
```

Quand on lit une ligne d'un fichier de ce genre, ce peut être soit une section, soit une paire clef-valeur. En première approche, la regex pour lire ce type de fichier pourrait être :

```
/ '[' \w+ ']' || \S+ \s* '=' \s* \S* /
```

C'est-à-dire :

- soit un mot entre crochets ;
- soit une chaîne composée de caractères autres que des espaces blancs, suivie de 0 ou plusieurs espaces, suivis du signe égal « `=` », suivi à nouveau d'espaces optionnels, suivis d'une autre chaîne composée de caractères autres que des espaces blancs.

Il existe une autre forme d'alternative, utilisant le séparateur « `|` » (au lieu de « `||` »). L'idée est la même, mais c'est la reconnaissance la plus longue (en non plus la première) qui est retenue (à condition de commencer au même emplacement), ce qui implique qu'elles doivent toutes être testées (éventuellement en parallèle) et comparées.

L'exemple ci-dessous illustre la différence entre les deux opérateurs :

```
my $chaîne = "abcdef";
say ~$/ if $chaîne ~~ /ab || abcde/;      # -> ab
say ~$/ if $chaîne ~~ /ab | abcde/;      # -> abcde
say ~$/ if $chaîne ~~ /ab | bcde/;      # -> ab
# dans ce dernier cas, la reconnaissance la plus à gauche l'emporte
```

La règle de la reconnaissance la plus longue est particulièrement utile pour procéder à l'analyse lexicale d'une chaîne de caractères appartenant par exemple à du code Perl. Ainsi, c'est elle qui permet à la grammaire Perl de reconnaître des identifiants (de variables, de fonctions, etc.) contenant des tirets (à condition qu'ils soient suivis d'un caractère alphabétique) : Perl est capable de reconnaître un tel identifiant (et de le distinguer par exemple d'un opérateur arithmétique moins placé entre deux symboles distincts) grâce à cette règle de la reconnaissance la plus longue.

2-5-1 - Conjonction (reconnaître ceci et cela)

Une alternative établit un *ou logique* (une disjonction) entre les termes de l'alternative séparés par les opérateurs « | » ou « || ».

Il existe également des opérateurs « & » et « && » établissant un *et logique* (une conjonction) entre les termes qu'ils séparent. La regex ne réussit que si les deux termes sont reconnus et s'ils reconnaissent la même sous-chaîne (même début et même fin de reconnaissance) :

```
my $regex = rx/a..d & .bcd/;
say "Reconnu" if "XZabcdZ" ~~ /$regex/; # -> Reconnu
say ~$/; # -> abcd
```

2-6 - Ancres

Le moteur de regex essaie de trouver une correspondance dans une chaîne en cherchant de gauche à droite.

```
say so 'Saperlipopette' ~~ /perl/; # True
#      ^^^^
# (so renvoie une évaluation booléenne, donc, en fait True ou False)
```

Mais ce n'est pas toujours ce que l'on désire. Par exemple, on peut vouloir reconnaître toute la chaîne, ou toute une ligne, ou un ou plusieurs mots entiers, ou attacher de l'importance à l'*endroit* de la chaîne où la reconnaissance se produit (par exemple uniquement en début de chaîne). Les ancres (et les assertions) permettent de spécifier où la reconnaissance aura lieu.

Il faut que les ancres d'une regex soient reconnues pour que l'ensemble de la regex le soit, mais les ancres *ne consomment pas* de caractère dans la chaîne.

Ancre	Signification	Remarques ou exemples
^	Début de la chaîne	'Saperlipopette' ~~ /perl/; # Vrai 'Saperlipopette' ~~ /^ perl/; # Faux 'perles fines' ~~ /^ perl/; # Vrai
^^	Début de ligne	^^ reconnaît le début de la chaîne ou ce qui suit un caractère de retour à la ligne
\$\$	Fin de ligne	\$\$ reconnaît la fin de la chaîne ou un caractère suivi d'un retour à la ligne
<<	Limite à gauche de mot	Autrement dit, un début de mot. Ou, plus précisément : la limite entre un caractère non-mot à gauche et un caractère mot à droite.
>>	Limite à droite de mot	Ou fin de mot. Par exemple : 'Carpe diem' ~~ /arpe >>/ ; # vrai 'Carpe diem' ~~ /die >>/ ; # faux
\$	Fin de chaîne	'Carpe diem' ~~ /arpe \$/ ; # faux

'Carpe diem' ~~ /diem \$/ ;
vrai

2-7 - Regroupements et captures

2-7-1 - Regroupements

Dans du code Perl 6 ordinaire (hors regex), les parenthèses permettent de regrouper des expressions, souvent pour modifier la priorité d'exécution :

```
say 1 + 4 * 2;      # 9, analysé comme : 1 + (4 * 2)
say (1 + 4) * 2;    # 10
```

On peut utiliser la même idée pour regrouper des éléments d'une regex :

```
/ a || b c /      # reconnaît 'a' ou 'bc'
/ ( a || b ) c /  # reconnaît 'ac' ou 'bc'
```

La même technique de regroupement peut s'appliquer aux quantificateurs :

```
/ a b+ /          # Reconnaît un 'a' suivi d'un ou plusieurs 'b'
/ (a b)+ /        # Reconnaît une ou plusieurs séquences 'ab'
/ (a || b)+ /      # Reconnaît une séquence quelconque de 'a' et
                  # de 'b' longue d'au moins un caractère
```

2-7-2 - Captures

Les parenthèses ne servent pas seulement à regrouper, elles servent aussi à capturer, c'est-à-dire qu'elles stockent la partie reconnue entre parenthèses dans une variable réutilisable ensuite, ainsi que sous la forme d'un élément de l'objet reconnu.

```
my $str = 'nombre 42';
if $str ~~ /nombre ' (\d+) / {
    say "Le nombre est $0";      # Le nombre est 42
    # ou
    say "Le nombre est $/[0]";   # Le nombre est 42
}
```

S'il y a plusieurs paires de parenthèses, elles sont numérotées de gauche à droite, en partant de zéro (\$0, \$1, \$2, etc.), contrairement à Perl 5 où les captures commencent à \$1.

```
if 'abc' ~~ /(a) b (c)/ {
    say "0: $0; 1: $1";          # 0: a; 1: c
}
```

Les variables \$0, \$1, etc. sont en fait des raccourcis. Ces captures sont canoniquement disponibles dans l'objet reconnu \$/ en utilisant celui-ci sous la forme d'une liste, si bien que \$0 est en fait du sucre syntaxique pour \$/[0], \$1 pour \$/[1] et ainsi de suite.

Forcer un contexte de liste à l'objet reconnu permet accéder facilement à tous les éléments :

```
if 'abcdef' ~~ /(a) b (c) (d) e (f)/ {
    say $/.list.join: ', ' # a, c, d, f
}
```


2-7-3 - Regroupements sans capture

Les parenthèses assurent une double fonction : elles regroupent des éléments à l'intérieur de la regex et elles capturent ce qui a été reconnu dans la sous-regex entre les parenthèses.

Pour ne conserver que le comportement de regroupement (sans capturer), on peut utiliser des crochets au lieu de parenthèses :

```
if 'abc' ~~ / [a|b] (c) / {
    say ~$0;          # c
}
```

Si l'on n'a pas besoin de capture, utiliser des groupes non capturants présente trois avantages : l'intention du développeur est plus claire, il est plus facile de compter les groupes capturants dont on a besoin, et c'est un peu plus rapide.

2-7-4 - Numérotation des captures

Il a été dit plus haut que les captures sont numérotées de gauche à droite. C'est vrai en principe, mais c'est un peu trop simplifié. Les règles qui suivent sont énumérées pour rendre ce document aussi complet que possible, mais si vous vous retrouvez à devoir les appliquer régulièrement, alors il est peut-être préférable d'envisager des **Captures nommées** (§ 2.7.5.) ou même des **Sous-règles ou règles nommées** (§ 2.8.).

Les alternatives réinitialisent à 0 la numérotation des captures :

```
/ (x) (y) || (a) (.) (.) /
# $0 $1      $0 $1 $2
```

Par exemple :

```
if 'abc' ~~ / (x) (y) || (a) (.) (.) / {
    #          $0 $1 $2
    say ~$1;    # b
}
```

Si les différentes options (plus ou moins complexes) d'une alternative ont des nombres de captures différentes, c'est celle ayant le plus grand nombre de captures qui détermine (logiquement) l'indice de la capture suivante.

```
$chaîne = 'abcd';
if $chaîne / a [ b (.) || (x) (y) ] (.) / {
    #          $0 $0 $1 $2
    say ~$2;    # d
}
```

Les captures peuvent être imbriquées, auquel cas elles sont numérotées par niveau.

```
if 'abc' ~~ / ( a (.) (.) ) / {
    say "Extérieur: $0";          # Extérieur: abc
    say "Intérieur: $0[0] et $0[1]"; # Intérieur: b et c
}
```

2-7-5 - Captures nommées

Au lieu de numéroter les captures, il est possible de leur donner des noms. La façon générique (et un peu bavarde) de nommer des captures est la suivante :

```
if 'abc' ~~ / $<mon_nom> = [ \w+ ] / {
```

```
say ~$<mon_nom>      # abc
}
```

L'accès à une capture nommée, `$<mon_nom>`, est en fait un raccourci pour accéder à l'objet reconnu sous la forme d'un hachage, autrement dit : `$/{'mon_nom'}` ou `$/<mon_nom>`.

Forcer l'objet reconnu dans un contexte de hachage donne un moyen d'accès simple à toutes les captures nommées :

```
if 'décompte=23' ~~ / $<variable>=\w+ '=' $<valeur>=\w+ / {
  my %h = $/.hash;
  say %h.keys.sort.join: ', ';      # valeur, variable
  say %h.values.sort.join: ', ';    # 23, décompte
  for %h.kv -> $k, $v {
    say "Trouvé valeur '$v' avec la clef '$k'";
    # Affiche ces deux lignes :
    #   Trouvé valeur 'décompte' avec la clef 'variable'
    #   Trouvé valeur '23' avec la clef 'valeur'
  }
}
```

La section suivante (**Sous-règles ou règles nommées**) offre un moyen souvent plus pratique d'accéder aux captures nommées.

2-8 - Sous-règles ou règles nommées

Il est possible de mettre des morceaux de regex dans des règles nommées, de même que l'on peut mettre des fragments de code dans une fonction (ou *subroutine*) ou une méthode.

```
my regex ligne { \N*\n }
if "abc\ndef" ~~ /<ligne> def/ {
  say "Première ligne: ", $<ligne>.chomp;      # Première ligne: abc
}
```

Une regex nommée peut se déclarer avec la syntaxe `my regex nom_regex {corps de la regex }`, et appelée ensuite avec `<nom_regex>`. En outre, invoquer une regex nommée crée *ipso facto* une capture nommée portant le même nom (`$<ligne>` dans l'exemple ci-dessus).

Il est toutefois possible, si on le désire, de donner un autre nom à la capture, avec la syntaxe d'appel suivante `<nom_capture=nom_regex>`. Si l'on n'a pas besoin de capture, préfixer le nom de la regex d'un point, `<.nom_regex>`, supprimera la capture.

Voici un extrait de code un peu plus complet (mais cependant encore assez limité) pour analyser le fichier de type `config.ini` du § 2.5. :

```
my regex entête { \s* '[' (\w+) ']' \h* \n+ }
my regex identifiant { \w+ }
my regex kvpair { \s* <clef=identifiant> '=' <val=identifiant> \n+ }
my regex section {
  <entête>
  <kvpair>*
}

# Exemple de fichier ini dans un document "ici même" :
my $contenu = q:to/FIN_INI/;
[passwords]
jean=mdp1
anne=plusfiab1e123
[quotas]
jean=123
anne=42
FIN_INI
```

```
my %config;
if $contenu ~~ /<section>*/ {
    for $<section>.list -> $section {
        my %section;
        for $section<kvpair>.list -> $p {
            say $p<val>;
            %section{ $p<key> } = ~$p<val>;
        }
        %config{ $section<header>[0] } = %section;
    }
}
say %config.perl;
# ("passwords" => {"jean" => "mdp1", "joy" => "plusfiabile123"},
#   "quotas" => {"jean" => "123", "anne" => "42"}).hash
```

Les regex nommées peuvent être regroupées en grammaires (voir § 3.) et il est souvent souhaitable de le faire (l'objectif des regex nommées est précisément de construire des grammaires).

Il existe des sous-règles prédéfinies, correspondant plus ou moins aux classes de caractères vues antérieurement, par exemple :

- `ident` : un identifiant ;
- `upper` : un seul caractère capital ;
- `lower` : un seul caractère minuscule ;
- `alpha` : un seul caractère alphabétique ou un caractère souligné « `_` » (pour un caractère alphabétique Unicode sans le caractère souligné, utiliser `<:alpha>` ;
- `digit` : un seul chiffre ;
- `xdigit` : un seul chiffre hexadécimal ;
- `print` : un seul caractère imprimable ;
- `punct` : un seul caractère ponctuation ;
- `alnum` : un seul caractère alphanumérique (équivalent à `<+alpha+digit>`) ;
- `wb` : limite de mot, assertion de longueur nulle ;
- `space` : un seul espace blanc (même chose que `\s`) ;
- `before motif` : assertion avant de longueur nulle, c'est-à-dire vérifie si l'on est à une position où le motif est reconnu et renvoie un objet reconnu de taille nulle en cas de succès (il existe de même une sous-règle `after motif` vers l'arrière). Voir aussi § 2.10.).

Une liste plus complète se trouve dans la *Synopse* 🇬🇧 **S05**.

Comme on le verra au chapitre suivant, une regex nommée ou sous-règle peut également être définie avec les mots-clés `token` ou `rule` (en lieu et place de `regex`), ce qui modifie la façon dont le moteur de regex analysera la chaîne en utilisant implicitement les adverbes `:ratchet` et/ou (selon le cas) `:sigspace` (voir les sections 2.9.1.1. et 2.9.1.2. ci-dessous). La syntaxe de la déclaration est alors la suivante :

```
my token nom-de-règle { ... }
# ou :
my rule nom-de-règle { ... }
```

2-9 - Adverbes

Les *adverbes* (qui correspondent à ce que l'on appelait *modificateurs* en Perl 5 et dans les packages de regex issus de Perl 5 employés dans d'autres langages comme *PCRE*) modifient la façon dont fonctionnent les regex et permettent des raccourcis très pratiques pour certaines tâches répétitives.

Il y a deux sortes d'adverbes : les *adverbes de regex* s'appliquent là où la regex est définie et les *adverbes de reconnaissance* là où le motif reconnaît une chaîne. La distinction est parfois un peu floue, car la reconnaissance et la définition sont souvent textuellement rapprochées, mais utiliser la syntaxe de méthode de la reconnaissance aide à clarifier la différence.

'abc' ~~ /. / équivaut à peu près à 'abc'.match(/./) et peut même s'écrire plus clairement en deux lignes distinctes :

```
my $regex = /. /;           # définition, deux caractères quelconques
if 'abc'.match($regex) {    # reconnaissance
    say "'abc' a au moins deux caractères";
}
```

Les adverbes de regex comme :i (ignorer la casse, c'est-à-dire la distinction entre lettres capitales et minuscules) vont dans la définition alors que des adverbes de reconnaissance comme :overlap (chevauchement) sont ajoutés à l'appel de la reconnaissance :

```
my $regex = /:i . a/;
for 'baA'.match($regex, :overlap) -> $m {
    say ~$m;
}
# Affiche :
#      ba
#      aA
```

2-9-1 - Adverbes de regex

Les adverbes qui apparaissent au moment de la déclaration d'une regex font partie intégrante de la regex et influent la façon dont le compilateur Perl 6 traduit la regex en code binaire.

Par exemple, l'adverbe :ignorecase ou :i (ignorer la casse) dit au compilateur d'ignorer les distinctions entre lettres capitales et minuscules. Ainsi, 'a' ~~ /A/ est faux, alors que 'a' ~~ /:i A/ est reconnu avec succès. (Il existe une variante :ii ou :samecase qui, dans le cas des substitutions, assure que la chaîne de remplacement sera dans la même casse que la chaîne reconnue.)

Les adverbes de regex peuvent être placés avant ou à l'intérieur d'une déclaration de regex, et n'affectent que la partie de la regex qui vient ensuite, lexicalement.

Ces deux regex sont équivalentes :

```
my $rx1 = rx:i/a/;          # avant
my $rx2 = rx/:i a/;         # à l'intérieur
```

Mais celles-ci ne le sont pas :

```
my $rx3 = rx/a :i b/;       # insensible à la casse seulement pour b
my $rx4 = rx/:i a b/;       # complètement insensible à la casse
```

Les crochets et les parenthèses limitent la portée d'un adverbe :

```
/ (:i a b) c /              # reconnaît 'ABc' mais pas 'ABC'
/ [:i a b] c /              # reconnaît 'ABc' mais pas 'ABC'
```

2-9-1-1 - L'adverbe « ratchet » : pas de retour arrière

L'adverbe :ratchet ou :r ordonne au moteur de regex de ne pas revenir en arrière (*backtrack*). Le mot anglais *ratchet* désigne un cliquet antiretour (comme dans une clef à cliquet), un système mécanique empêchant un dispositif de revenir en arrière (et le forçant donc, implicitement, à aller de l'avant).

Sans cet adverbe, différentes parties d'une regex vont essayer différentes façons de reconnaître une chaîne afin de permettre à d'autres parties de la regex de correspondre. Par exemple, avec la regex 'abc' ~~ /\w+ ./, la partie \w+ commence par consommer toute la chaîne, abc, puis échoue sur le « . » qui suit. Il y a alors un retour arrière (ou retour sur trace), c'est-à-dire que \w+ abandonne le dernier caractère et ne reconnaît que ab, ce qui permet au « . »

de reconnaître avec succès `c`. Ce processus consistant à abandonner un caractère (ou plusieurs) pour recommencer un nouvel essai de reconnaissance s'appelle retour arrière (parfois retour sur trace) ou *backtracking*.

```
say so 'abc' ~~ / \w+ . /;      # Vrai
say so 'abc' ~~ / :r \w+ . /;  # Faux
```

L'utilisation d'un tel « cliquet » (de l'adverbe *ratchet*) peut être une optimisation, car les retours arrière sont souvent coûteux. Mais l'intérêt est surtout que la reconnaissance sans retour arrière correspond étroitement à la façon dont les humains analysent un texte qu'ils lisent. Avec les regex `my regex identifiant { \w+ }` et `my regex keyword { if | else | endif }`, on attend intuitivement que l'identifiant absorbe un mot complet et n'ait pas besoin de restituer la fin de ce mot pour satisfaire la règle suivante. Par exemple, personne ne s'attend à ce que le mot `motif` soit analysé comme l'identifiant `mot` suivi du mot-clé `if` ; on attend plutôt que `motif` soit analysé comme un identifiant et, si l'analyseur attend le mot `if` à sa suite, qu'il échoue plutôt que d'analyser la donnée en entrée différemment de ce que l'on attend.

On peut considérer que le retour arrière est le comportement généralement recherché pour une analyse de bas niveau, caractère par caractère, d'une chaîne de caractères, mais que la recherche avec cliquet (*ratchet*) correspond généralement mieux à ce que l'on désire faire pour l'analyse lexicale ou syntaxique d'un texte structuré.

Comme le comportement à cliquet est si souvent souhaitable dans les analyseurs lexicaux (*lexers*) et syntaxiques (*parsers*), il existe un raccourci pour une regex à cliquet : définir la sous-règle en utilisant le mot-clé *token* au lieu de *regex* :

```
my token truc { .... }
# raccourci pour :
my regex truc { :r ... }
```

Une sous-règle utilisant le mot-clé *rule* (voir § 2.9.1.2.) aura également l'effet d'empêcher le retour arrière.

L'existence des règles de type *token* et *rule* fait que l'adverbe `:ratchet` est en fait assez rarement utilisé de façon explicite : on préfère généralement l'utiliser *implicitement* en définissant la sous-règle en tant que *token* ou *rule*.

2-9-1-2 - L'adverbe « sigspace » (espaces blancs significatifs)

L'adverbe `:sigspace` ou `:s` rend les espaces blancs significatifs dans une regex (ils ne sont plus ignorés comme dans les exemples jusqu'ici) :

```
say so "J'ai utilisé Photoshop@" ~~ m:i/ photo shop /; # Vrai
say so "J'ai utilisé photo shop"  ~~ m:i:s/ photo shop /; # Vrai
say so "J'ai utilisé Photoshop@"  ~~ m:i:s/ photo shop /; # Faux
```

`m:s/ photo shop /` se comporte comme si l'on avait écrit `m/ photo <.ws> shop <.ws> /`. Par défaut, `<.ws>` assure que les mots sont séparés, si bien que `'a b'` sera reconnu par `<.ws>`, mais pas `'ab'`.

Un espace dans une regex se transforme ou non en `<.ws>` selon ce qui précède l'espace. Dans l'exemple ci-dessus, l'espace au début de la regex ne se transforme pas en `<.ws>`, mais l'espace après les caractères le fait. D'une façon générale, la règle est que si un terme peut reconnaître quelque chose, alors un espace suivant ce terme est converti en `<.ws>`.

En outre, s'il y a un espace après un terme, mais avant un quantificateur (`+`, `*`, ou `?`), `<.ws>` sera reconnu après chaque reconnaissance de ce terme, si bien que `toto +` devient `[toto <.ws>]+`. En revanche, un espace après un quantificateur se comporte comme un espace significatif normal, par exemple `"toto+ "` devient `toto+ <.ws>`.

En définitive, ce code :

```
rx :s {
  ^^
  {
```

```

    say "Pas de sigspace après ceci";
}
<.assertion_puis_ws>
caractères_puis_ws+
caractères_séparés_par_ws *
[
| des "trucs" .. .
| $$
]
:my $toto = "pas de ws après ceci";
$toto
}

```

devient:

```

rx {
  ^^ <.ws>
  {
    say "Pas d'espace après ceci";
  }
  <.assertion_puis_ws> <.ws>
  caractères_puis_ws++ <.ws>
  [ caractères_séparés_par_ws <.ws>]* <.ws>
  [
  | des <.ws> "trucs" <.ws> .. <.ws> . <.ws>
  | $$ <.ws>
  ] <.ws>
  :my $toto = "pas de ws après ceci";
  $toto <.ws>
}

```

De même qu'une regex déclarée avec le mot-clef *token* implique l'adverbe `:ratchet`, une regex déclarée avec le mot-clef *rule* implique à la fois les adverbes `:ratchet` et `:sigspace`.

Les grammaires offrent un moyen simple de redéfinir ce que `<.ws>` reconnaîtra :

```

grammar Demo {
  token ws {
    <!ww> # reconnaissance si pas à l'intérieur d'un mot
    \h*   # reconnaît seulement les espaces horizontaux
  }
  rule TOP { # appelée par Demo.parse;
    a b ' '
  }
}

say so Demo.parse("ab.");      # Faux (espace requis entre a et b)
say so Demo.parse("a b.");     # Vrai
say so Demo.parse("a\tb .");  # Vrai
say so Demo.parse("a\tb\n.");  # Faux (\n est un espace vertical)

```

Lors de l'analyse de formats de fichiers dans lesquels certains types d'espaces blancs (par exemple les espaces verticaux) sont significatifs, il est souvent souhaitable de redéfinir `ws`.

2-9-1-3 - L'adverbe « Perl5 » pour retrouver la syntaxe de Perl 5

Les adverbes `:P5` et `:Perl5` permettent d'utiliser la syntaxe des expressions régulières de Perl 5 dans un programme Perl 6 :

```

next if $ligne =~ m/[aeiou]/ ; # Classe de caractères Perl 5
next if $ligne =~ m:P5/[aeiou]/ ; # Perl 6, avec l'adverbe P5
next if $ligne =~ m:Perl5/[aeiou]/ ; # Perl 6, avec l'adverbe Perl5
next if $ligne =~ m/ <[aeiou]> / ; # Perl 6, nouvelle syntaxe des
                                   # classes de caractères

```

Cela peut rendre service pour des migrations de programmes ou aux personnes non encore habituées aux regex de Perl 6, mais nous ne nous attarderons pas plus sur cette possibilité dans ce tutoriel dont l'objet est de présenter la syntaxe des regex Perl 6.

2-9-2 - Les adverbes de reconnaissance

Contrairement aux adverbes de regex (§ 2.9.1.), qui sont liés à la déclaration d'une regex, les adverbes de reconnaissance n'ont un sens qu'au moment où l'on veut faire correspondre une chaîne et une regex.

Ils ne peuvent jamais figurer à l'intérieur d'une regex, mais seulement à l'extérieur de celle-ci, soit dans le cadre d'une reconnaissance `m/.../`, soit comme argument d'une méthode de reconnaissance.

2-9-2-1 - Adverbe « continue » (position de départ de la reconnaissance)

L'adverbe `:continue` ou `:c` prend un argument : la position à laquelle la regex doit commencer sa recherche. Par défaut, la recherche commence au début de la chaîne, mais l'adverbe `:c` modifie ce comportement. Si aucune position n'est spécifiée, la recherche commencera en position 0, sauf si l'objet reconnu `$\` est défini, auquel cas la recherche commencera à la position `$/to` (position suivant la fin de la reconnaissance précédente).

```
given 'alxa2' {
  say ~m/a./;      # a1
  say ~m:c(2)/a./;  # a2 (la recherche commence à la position du x)
}
```

2-9-2-2 - Adverbe « exhaustive » (toutes les reconnaissances)

Pour trouver toutes les reconnaissances possibles d'une regex - y compris celles qui se chevauchent - et plusieurs reconnaissances qui commencent à la même position, il faut utiliser l'adverbe `:exhaustive` ou `:ex`.

```
given 'abracadabra' {
  for m:exhaustive/ a .* a / -> $match {
    say ' ' x $match.from, ~$match;
  }
}
```

Le code ci-dessus affiche les résultats suivants :

```
abracadabra
abracada
abraca
abra
  acadabra
  acada
  aca
    adabra
    ada
      abra
```

2-9-2-3 - Adverbe « global » (chaque reconnaissance)

Au lieu de rechercher une seule correspondance et de retourner un objet reconnu, il est possible de rechercher chaque reconnaissance sans chevauchement et de renvoyer le résultat sous forme d'une liste, en utilisant l'adverbe `:global` ou `:g` :

```
given 'Trois mots ici' {
  my @matches = m:global/\w+/;
  say @matches.elems;      # 3
}
```

```
say ~@matches[2];      # ici
}
```

2-9-2-4 - Adverbe « pos »

L'adverbe `:pos` ou `:p` ancre la reconnaissance à une position spécifique de la chaîne :

```
given 'abcdef' {
  my $trouvé = m:pos(2)/.*/;
  say $trouvé.from;      # 2
  say ~$trouvé;         # cdef
}
```

2-9-2-5 - Adverbe « overlap » (avec chevauchement)

Pour obtenir plusieurs reconnaissances, y compris des reconnaissances qui se chevauchent, mais seulement une seule (la plus longue) pour chaque position de départ, il est possible d'utiliser l'adverbe `:overlap` ou `:ov` :

```
given 'abracadabra' {
  for m:overlap/ a .* a / -> $match {
    say ' ' x $match.from, ~$match;
  }
}
```

Ce qui affiche :

```
abracadabra
 acadabra
  adabra
   abra
```

2-10 - Regarder devant et derrière (assertions)

Les assertions permettent de rechercher des correspondances vers l'avant ou vers l'arrière, mais sans consommer la chaîne cible (en restant à la même position), comme le font les ancres.

2-10-1 - Assertions avant

Pour vérifier si un motif apparaît avant un autre motif, on peut utiliser l'*assertion avant* `before`, de la forme suivante : `<?before motif>`.

Ainsi, pour rechercher la chaîne `toto` immédiatement suivie de la chaîne `titi`, on peut utiliser la regex `rx{ toto <?before titi> }`, par exemple comme suit :

```
say "tototiti" ~~ rx{ toto <?before titi> };      # -> toto
```

Si l'on souhaite au contraire rechercher un motif qui ne soit pas immédiatement suivi par un autre motif, il faut utiliser une *assertion avant négative*, de la forme suivante : `< !before motif>`.

Par exemple, pour rechercher la chaîne `toto` non immédiatement suivie de la chaîne `titi`, on peut utiliser la regex `rx{ toto < !before titi> }`.

2-10-2 - Assertions arrière

Pour vérifier si un motif apparaît après un autre motif, on peut utiliser l'*assertion arrière* `after`, de la forme suivante : `<?after motif>`.

Par exemple, pour rechercher la chaîne `toto` immédiatement précédée de la chaîne `titi`, on peut utiliser la regex `rx{ toto <?after titi> }` comme suit :

```
say "tittitoto" ~~ rx{ toto <?after titi> }; # -> toto
```

Si l'on souhaite au contraire rechercher un motif qui ne soit pas immédiatement précédé par un autre motif, il faut utiliser une *assertion arrière négative*, de la forme suivante : `<!after motif>`.

Par exemple, pour rechercher la chaîne `toto` non immédiatement suivie de la chaîne `titi`, on peut utiliser la regex `rx{ toto <!after titi> }`.

3 - Grammaires

Les grammaires sont un outil puissant pour décomposer un texte en éléments individuels et, souvent, renvoyer les structures de données qui ont été créées en interprétant ce texte.

Par exemple, un programme Perl 6 est interprété et exécuté en utilisant une grammaire Perl 6.

Un exemple de portée plus pratique pour un utilisateur courant de Perl 6 est le module **JSON::Tiny** de Perl 6, qui peut désérialiser n'importe quel fichier JSON valide. Le code qui effectue cette désérialisation est écrit en moins de 100 lignes de code simple et facile à étendre. Nous décrivons plus bas (§ 3.6.) la construction détaillée d'une telle grammaire capable d'analyser du JSON.

Si vous n'aimiez pas la grammaire à l'école, ne partez pas en courant. Les grammaires Perl 6 sont en fait un simple moyen de regrouper des regex, de même que les classes permettent de regrouper des méthodes de code ordinaire (et l'analogie va beaucoup plus loin qu'on pourrait le croire de prime abord, comme on le verra plus loin).

3-1 - Les « briques » de construction d'une grammaire

Une grosse partie de ce qu'il y a besoin de savoir et de comprendre pour écrire une grammaire a déjà été vue ci-dessus, en particulier au chapitre 2.8. **Sous-règles ou règles nommées**, ainsi qu'aux sous-chapitres 2.9.1.1. et 2.9.1.2. ci-dessus. Ces sous-règles ou règles nommées constituent les briques élémentaires d'une grammaire. En fait, l'un des principaux buts d'une grammaire est de regrouper des sous-règles ou regex nommées (de types `regex`, `token` et `rule`) dans un espace de noms bien délimité afin d'éviter les collisions de noms d'identifiants avec d'autres regex ailleurs dans le code.

Rappelons que les `regex`, `token` et `rule` sont des entités très semblables servant à déclarer une regex nommée sous une forme ressemblant à la définition d'une fonction ou d'une méthode. Dans ce chapitre, nous les appellerons désormais collectivement *règles*, indépendamment du mot-clef utilisé pour les déclarer. On les déclare de la façon suivante :

```
regex ma_regex { ... } # regex ordinaire
token mon_token { ... } # regex avec adverbe :ratchet implicite
rule ma_règle { ... } # regex avec :ratchet et :sigspace implicites
```

Pour mémoire :

- Le mot-clef `regex` signale une regex ordinaire ;

- Le mot-clef `token` implique l'adverbe `:ratchet` (« cliquet ») implicite, c'est-à-dire que ce genre de règle ne fait pas de retour arrière (pas de *backtracking*) ;
- Le mot-clef `rule` indique implicitement les adverbes `:ratchet` (pas de retour arrière) et `:sigspace` (les espaces du motif ne sont pas ignorés).

Voici un exemple dans lequel on définit une première règle `chiffres`, et l'utilise pour en définir une seconde, `décimal` :

```
my regex chiffres { \d+ }
my regex décimal { <chiffres> \. <chiffres> }
say so " Cet objet coûte 13.45 euros" ~~ /<décimal>; # -> True
# (so renvoie une évaluation booléenne, donc, en fait True ou False)
say ~$/; # -> 13.45
```

On voit ci-dessus qu'une règle peut appeler une autre règle, de même qu'une fonction peut appeler une autre fonction. Une règle peut aussi s'appeler elle-même récursivement. Ce mécanisme capital est le cœur de la puissance des regex de Perl 6 et de leur capacité à créer des grammaires.

Une règle de type `token` ne fait pas de retour arrière :

```
my token lettres { abc .+ g };
say "abcdefg" ~~ /<lettres>/ ?? "Réussit" !! "Échoue"; # -> Échoue
```

Cette règle échoue ici parce que le sous-motif `.+`, avec son quantificateur gourmand « `+` », consomme toute la fin de la chaîne, y compris la dernière lettre « `g` », si bien que la règle ne peut plus reconnaître le « `g` » final déjà consommé : comme elle n'est pas autorisée à revenir en arrière pour abandonner le « `g` » et le laisser à la disposition du dernier atome du motif afin de parvenir à réussir, elle échoue.

Avec un quantificateur non gourmand (ou *frugal*), la règle `lettres` modifiée réussit :

```
my token lettres { abc .+? g };
say "abcdefg" ~~ /<lettres>/ ?? "Réussit" !! "Échoue"; # -> Réussit
```

Avec une définition identique, une règle de type `rule` échoue :

```
my rule lettres { abc .+? g };
say "abcdefg" ~~ /<lettres>/ ?? "Réussit" !! "Échoue"; # -> Échoue
```

parce que les espaces présents dans le motif ne sont plus ignorés (et qu'on ne les retrouve pas dans la chaîne).

Les noms des trois types de règles nommées, *regex*, *token* et *rule*, reflètent l'idée que, d'une façon très générale et assez floue, dans une grammaire, une *regex* fait à peu près ce que l'on attend usuellement d'une expression régulière (analyse de bas niveau, caractère par caractère), alors qu'une *token* va souvent servir plutôt à l'analyse *lexicale* (division du texte en entrée en « mots » ou lexèmes individuels) et qu'une *rule* sera plus souvent destinée à l'analyse *syntactique* (analyse du rapport entre les lexèmes, éventuellement en fonction du contexte). Cette répartition théorique des tâches est sujette à de très nombreuses exceptions et ne doit surtout pas être suivie de façon dogmatique, mais il n'est pas inutile de conserver cette idée à l'esprit lors de la décision de choix entre les différents types de règles.

3-2 - Créer une grammaire

Une grammaire crée un espace de noms propre et s'introduit avec le mot-clef `grammar`.

3-2-1 - Syntaxe de définition d'une grammaire

De même qu'une classe peut regrouper des actions ou méthodes nommées :

```
class Identité {
```

```

method nom      { "Nom = $!nom" }
method âge      { "Âge = $!âge" }
method adresse  { "Adr = $!adresse" }

method desc {
    print &.nom(),      "\n",
          &.âge(),      "\n",
          &.adresse(),  "\n";
}

# etc.
}

```

une grammaire regroupe des règles nommées :

```

grammar Identité {
    rule nom      { Nom '=' (\N+) } # chaîne de caractères quelconques
                                   # autres que des retours à la ligne
    rule adresse  { Adr '=' (\N+) } # idem
    rule âge      { Age '=' (\d+) } # des chiffres

    rule desc {
        <nom>      \n
        <âge>      \n
        <adresse> \n
    }

    # etc.
}

```

La règle `desc` est ici définie en utilisant d'autres règles (`nom`, `âge` et `adresse`) définies par ailleurs. Cela permet de construire progressivement des niveaux d'abstraction de plus en plus élevés.

Ceci crée un objet de type `Grammar` dont le type dénote le langage en cours d'analyse et dont on peut dériver d'autres grammaires sous la forme de langages étendus.

Le nouvel objet est ensuite passé comme invoquant de la méthode `TOP` (`regex`, `token` ou `rule`). Cette méthode `TOP` par défaut peut être remplacée par une autre passée avec le paramètre nommé `:rule`, ce qui peut être utile notamment pour tester une grammaire.

À remarquer ici qu'il n'est plus nécessaire de déclarer les règles avec l'opérateur `my` comme cela avait été fait jusqu'à présent, parce la grammaire crée l'espace de noms et la portée lexicale nécessaires.

3-2-2 - Héritage de grammaires

Une grammaire peut hériter d'une autre grammaire (parente) de la même façon qu'une classe peut hériter d'une autre classe :

```

grammar Message {
    rule texte    { <salutation> $<corps>=<ligne>+? <fin> }
    rule salutation { [Salut|Bonjour] $<dest>=<S+? '&', ' ' > }
    rule fin      { [à|'@'] plus '<auteur>=<.+ >' }
    token ligne   { \N* \n }
}

grammar MessageFormel is Message {
    rule salutation { [Cher|Chère] $<dest>=<S+? '&quot;,&quot; ' >' }
    rule fin { Bien cordialement '&quot;,&quot; $<auteur>=<.+ >' }
}

```

Ici, la grammaire `MessageFormel` hérite de la grammaire parente `Message`. Comme pour les méthodes d'une classe, les règles sont héritées de la grammaire parente (et polymorphiques), il n'y a donc pas besoin de définir à nouveau les règles `texte` et `ligne` qui ne changent pas. On ne surcharge que les règles qui changent.

Toutes les grammaires dérivent de la classe `Grammar`, qui fournit entre autres les méthodes communes à toutes les grammaires, comme `.parse` et `.parsefile`, décrites ci-dessous.

La capacité des grammaires d'hériter d'autres grammaires est un instrument extrêmement puissant et un facteur essentiel permettant l'extensibilité du langage Perl 6 (voir § 3.8.).

3-3 - Utiliser une grammaire

Il est possible d'analyser une chaîne de caractères avec une grammaire en appelant la méthode `.parse` sur cette grammaire et en passant optionnellement en paramètre un *objet d'actions* (voir § 3.4. ci-dessous). De même, la méthode `.parsefile` permet d'analyser un fichier.

```
MaGrammaire.parse($chaîne, :actions($objet-action))
MaGrammar.parsefile($nom-fic, :actions($objet-action))
```

Les méthodes `.parse` et `.parsefile` sont ancrées au début et à la fin du texte, et échouent si la fin du texte n'est pas atteinte.

En principe, il ne faut utiliser une grammaire que pour effectuer l'analyse lexicale et syntaxique proprement dite du texte. Pour extraire des données complexes, il est recommandé d'utiliser un objet d'actions en conjonction avec la grammaire.

3-4 - Les classes et objets d'actions

3-4-1 - Exécuter du code lors d'une reconnaissance

Quand une grammaire analyse avec succès un texte, elle renvoie un arbre syntaxique d'objets reconnus. Plus cet arbre est profond (et il le devient souvent très rapidement), et plus il y a de branches dans cet arbre, plus il devient difficile d'explorer cet arbre pour y trouver l'information que l'on recherche.

Pour éviter de devoir procéder à cette exploration de l'arbre des reconnaissances, on peut fournir un *objet d'action*. Après chaque analyse réussie d'une règle nommée de la grammaire, celle-ci cherche à invoquer une méthode de cet objet d'action portant le même nom que la règle, en lui fournissant en argument positionnel l'objet reconnu qui vient d'être créé. Cette méthode, si elle existe, peut notamment servir à construire un arbre syntaxique abstrait (*Abstract Syntax Tree* ou AST) ou à faire toutes sortes d'autres choses dont on pourrait avoir besoin pour la suite. Si cette méthode n'existe pas, cette étape est simplement ignorée.

Voici un exemple minimaliste et assez artificiel d'une grammaire et d'actions travaillant de concert :

```
use v6;

grammar GrammaireTest {
    token TOP { ^ \d+ $ }
}

class ActionsTest {
    method TOP($/) {
        $/.make(2 + ~$/.);
    }
}

my $actions = ActionsTest.new;
my $reconnu = GrammaireTest.parse('40', :$actions);
say $reconnu;           # -> #40#
say $reconnu.made;      # -> 42
```

L'objet `$actions` de la classe `Actionstest` est instancié puis passé en argument lors de l'appel de la méthode `.parse`. Quand la règle `TOP` reconnaît l'argument, la grammaire appelle automatiquement la méthode `TOP` en lui passant l'objet reconnu en argument.

La méthode `make` de la classe `Match` alimente la structure `$/made` (l'utilisateur décide son contenu, mais ce sera souvent un arbre syntaxique abstrait) avec son argument.

Pour bien montrer que l'argument est un objet reconnu, l'exemple utilise `$/` comme nom de paramètre passé à la méthode `action`, mais ce n'est qu'une convention bien pratique, il n'y a rien d'intrinsèquement indispensable ici. Passer `$reconnu` aurait fonctionné tout aussi bien. (À noter cependant qu'utiliser `$/` donne le léger avantage de rendre disponible le raccourci `$<capture>` au lieu de `$/<capture>`.)

Voici un exemple un peu plus construit :

```
use v6;

grammar PairesClésValeurs {
    token TOP {
        [<paire> \n+]*
    }
    token ws { \h* }

    rule paire {
        <clé=.identifiant> '=' <valeur=.identifiant>
    }
    token identifiant {
        \w+
    }
}

class PairesClésValeursActions {
    method identifiant($/) { $/.make: ~$/ }
    method paire ($/) { $/.make: $<clé>.made => $<valeur>.made }
    method TOP ($/) { $/.make: $<paire>».made }
}

my $res = PairesClésValeurs.parse(q:to/EOI/, :actions(PairesClésValeursActions)).made;
phase=b
points=42
Perl=6
EOI

for @$res -> $p {
    say "Clef : $p.clé()\tValeur : $p.valeur()";
}
```

Ce qui affiche le résultat suivant :

```
Clef : phase      Valeur : b
Clef : points     Valeur : 42
Clef : Perl       Valeur : 6
```

La règle `paire`, qui analyse une paire d'identifiants séparés par le caractère égal, fournit un alias aux deux appels de la règle `identifiant` afin de distinguer les noms des captures et de les mettre à disposition plus facilement et plus intuitivement. La méthode-action correspondante construit un objet de type `Pair` et utilise la propriété `.made` des sous-objets reconnus. Ainsi (et de même que la méthode-action `TOP`), elle exploite le fait que les méthodes-actions des sous-reconnaisances sont appelées avant celles correspondant à la regex englobante. Les méthodes-actions sont ainsi appelées dans l'ordre voulu.

La méthode-action `TOP` se contente de collecter tous les objets qui ont été générés par les reconnaissances multiples de la règle `paire` et les renvoie sous la forme d'une liste.

À noter également que `PairesClésValeursActions` est ici passé sous la forme d'un objet type (et non d'une instance d'un objet) à la méthode `.parse`, ce qui est possible parce qu'aucune des méthodes-actions n'utilise d'attributs (qui ne seraient disponibles que dans une instance dûment initialisée).

Il existe des cas dans lesquels il peut être souhaitable de conserver des états dans des attributs. Dans ce cas, c'est un objet instancié qu'il faut passer à la méthode `.parse`.

Les méthodes-actions peuvent aussi servir à déboguer des grammaires récalcitrantes, par exemple en affichant des états intermédiaires de l'analyse syntaxique. Mais le rôle essentiel des méthodes-actions est en fait de transformer une simple grammaire en un réel outil d'analyse syntaxique produisant un arbre syntaxique abstrait susceptible d'être utilisé par l'interpréteur Perl (ou d'autres outils) pour générer par exemple du code exécutable.

3-4-2 - Autres façons d'exécuter du code dans une grammaire

Dans les exemples ci-dessus, les méthodes-actions sont définies dans une classe d'actions distincte de la grammaire proprement dite, et c'est généralement la voie à suivre dans toute grammaire un tant soit peu étoffée.

Pour des cas simples, il est toutefois également possible de définir des méthodes au sein même de la grammaire :

```
grammar toto {
  regex titi { <.configurer> blah blah }
  method configurer {
    # faire quelque chose ici
  }
}
```

Oui, les grammaires peuvent définir des méthodes (et elles peuvent même utiliser des rôles), ce sont vraiment des classes...

Il est également possible d'exécuter du code au sein même d'une règle en l'insérant entre des accolades :

```
grammar toto {
  regex titi { blah blah { say "Je suis arrivé ici" } blah blah }
}
```

Si la portion du motif qui précède le bloc de code est reconnue, alors ce bloc est immédiatement exécuté.

Ceci fonctionne à la vérité dans une regex quelconque (règle nommée ou non, même en-dehors d'une grammaire) et cela peut par exemple aider à déboguer une regex (*voir aussi* § [4.5. Déboguer des regex ou des grammaires Perl 6](#)) :

```
my regex lettres { a b {say "reconnu ab"} c d}
say so "abc" ~~ /<lettres>/; # -> reconnu ab / False
# la regex ci-dessus a reconnu ab et l'affiche et n'a échoué qu'ensuite
say so "abcde" ~~ /<lettres>/; # -> reconnu ab / True - ici, succès
```

Attention, cependant : si la regex effectue des retours arrière, le bloc de code risque d'être exécuté plusieurs fois :

```
say so "aaa" ~~ / a { say "Bonjour" } b /;
```

Ce qui affiche :

```
Bonjour
Bonjour
Bonjour
False
```

3-5 - Une grammaire pour valider des noms de modules Perl

L'objectif de cet exemple est de valider un nom de module Perl.

Le nom d'un module Perl peut se décomposer en identifiants séparés par des paires de caractères deux-points : « :: », par exemple **List::Util** ou **List::MoreUtils** (les exemples de noms de modules fournis ici sont des modules Perl 5). Un identifiant doit commencer par un caractère alphabétique (a-z) ou un caractère souligné, suivi de zéro, un ou plusieurs caractères alphanumériques.

Rien de bien complexe jusqu'ici, mais ceci se complique quelque peu du fait que certains modules ont un seul identifiant (**Memoize**), et donc pas de caractères deux-points, et que d'autres peuvent avoir des noms « à rallonge » : **Regexp::Common::Email::Address**.

N'est-ce pas un bon candidat pour une grammaire ?

3-5-1 - La grammaire de validation

Il suffit, par exemple, de définir une règle identifiant garantissant les règles de nommage ci-dessus et une règle séparateur, et de les combiner adéquatement dans une grammaire.

```
grammar Valide-Nom-Module {
  token TOP { ^ <identifiant> [ <séparateur> <identifiant> ]* $ }
  token identifiant {
    <[A..Za..z_]>      # 'mot' commençant par un caractère
                      # alphabétique ou un caractère souligné
    <[A..Za..z0..9]>*> # 0 ou plusieurs caractères alphanumériques
  }
  rule séparateur { '::' } # paire de caractères deux-points
}
```

On peut maintenant tester cette grammaire avec quelques noms de modules valides ou non :

```
for <Super:Nouveau::Module Super.Nouveau.Module
  Super::6ouveau::Module Super::Nouveau::Module> -> $nom {
  my $reconnu = Valide-Nom-Module.parse($nom);
  say "nom\t", $reconnu ?? $reconnu !! "Nom de module invalide";
}
```

Ce qui affiche :

```
Super:Nouveau::Module  Nom de module invalide
Super.Nouveau.Module   Nom de module invalide
Super::6ouveau::Module Nom de module invalide
Super::Nouveau::Module #Super::Nouveau::Module#
identifiant => #Super#
séparateur => #::#
identifiant => #Nouveau#
séparateur => #::#
identifiant => #Module#
```

Seul le nom de module valide a été reconnu, les trois autres ont été rejetés à juste titre.

Parfois, les noms de modules sont résumés en remplaçant les paires de caractères deux-points par des tirets. Par exemple, le nom officiel est **Regexp::Common::Email::Address** et peut aussi s'écrire **Regexp-Common-Email-Address**. Si l'on désire valider cette seconde écriture, il suffit de modifier le séparateur pour qu'il autorise également un tiret :

```
rule séparateur { '::' || '-' } # deux car. deux-points ou tiret
```

En testant la grammaire avec le nom « Super-Nouveau-Module », on obtient :

```
Super-Nouveau-Module #Super-Nouveau-Module#
identifiant => #Super#
séparateur => #-#
identifiant => #Nouveau#
séparateur => #-#
identifiant => #Module#
```

Il a suffi de modifier la règle `séparateur` pour que la modification se propage à toute la grammaire, jusqu'à la règle `TOP`.

3-5-2 - Ajout d'un objet d'actions

La grammaire ci-dessus peut déterminer si un nom de module Perl est valide ou non.

On désire maintenant ajouter un avertissement si le nom du module est trop long (plus de 5 identifiants) ; dans ce cas, le nom du module sera toujours valide, mais l'on pourra peut-être conseiller à l'auteur du module d'essayer de choisir un nom plus court.

Il suffit par exemple d'ajouter une classe d'actions `Valide-Nom-Module-Actions` définie comme suit :

```
class Valide-Nom-Module-Actions {
  method TOP($/) {
    if $<identifiant>.elems > 5 {
      warn "Nom de module très long! Peut-être le réduire ?\n"
    }
  }
}
```

La définition de la classe n'a rien de particulier, c'est une classe Perl 6 ordinaire. La particularité importante est que la seule méthode définie ici a le même nom qu'une des règles de la grammaire (en l'occurrence, la règle d'entrée dans la grammaire, `TOP`). L'avertissement sera envoyé si le nombre d'identifiants dépasse 5, mais cela n'empêchera pas de valider le nom du module.

La syntaxe d'appel de la grammaire est modifiée comme suit :

```
my $reconnu = Valide-Nom-Module.parse($nom, :actions(Valide-Nom-Module-Actions));
```

Le résultat est le même que précédemment si l'on appelle la grammaire avec le nom de module « `Super::Nouveau::Module` » (ou « `Super-Nouveau-Module` »), ce qui est rassurant.

Mais avec un nom de module à la Mary Poppins :

```
my $nom = "Mon::Module::Super::Cali::Fragi::Listi::Cexpi::Delilicieux";
my $reconnu = Valide-Nom-Module.parse($nom, :actions(Valide-Nom-Module-Actions));
say $reconnu if $reconnu;
```

on obtient l'avertissement :

```
> perl6 grammaire_nom_module.pl
Nom de module très long! Peut-être le réduire ?
  in method TOP at perl6_grammaire_module.pl:15
#Mon::Module::Super::Cali::Fragi::Listi::Cexpi::Delilicieux#
identifiant => #Mon#
séparateur => #::#
identifiant => #Module#
séparateur => #::#
identifiant => #Super#
(...)
identifiant => #Delilicieux#
```



suivi de l'affichage de l'objet reconnu.

À noter que l'appel de la grammaire passe ci-dessus en paramètre directement le nom de la classe d'actions. Comme cela a été indiqué au § 3.4.1., il est possible, au besoin, de passer un objet instancié de cette classe :

```
my $nom = "Mon::Module::Super::Cali::Fragi::Listi::Cexpi::Delilicieux";
my $actions = Valide-Nom-Module-Actions.new();
my $reconnu = Valide-Nom-Module.parse($nom, :actions($actions));
```

Ce qui affiche :

```
perl6 grammaire_nom_module.pl
Nom de module très long! Peut-être le réduire ?
in method TOP at perl6_grammaire_module.pl:15
```

Cet exemple de validation d'un nom de module est très librement inspiré d'une idée provenant de l'article  **How to create a grammar in Perl 6** de David Farrell.

3-6 - Une grammaire pour analyser du JSON

Le JSON (*JavaScript Object Notation*) est un format de données textuelles dérivé de la notation des objets du langage JavaScript. Il est devenu (avec d'autres comme XML et YAML) un standard communément utilisé pour sérialiser des structures de données, ce qui permet par exemple de les échanger entre des plates-formes ou des langages différents, ou de les stocker de façon persistante (dans un fichier, par exemple).

3-6-1 - Structure d'un document JSON

Le format **JSON** a l'avantage d'être simple. Un document JSON comprend deux types d'entités structurales :

- des objets ou listes de paires nom/valeur (ce qui correspond essentiellement des objets JavaScript ou des tables de hachage Perl) ;
- des listes ordonnées de valeurs (des tableaux).

Les valeurs elles-mêmes peuvent être soit (récursivement) des objets ou des tableaux tels que décrits ci-dessus, soit des données génériques élémentaires : booléen (*true* ou *false*), nombre, chaîne ou *null* (valeur vide ou absence de valeur définie).

Une chaîne est une séquence contenant un nombre entier (éventuellement nul) de caractères de type unicode. Un nombre a le format d'un décimal signé qui peut contenir une partie fractionnaire ou élevé à la puissance (notation E). JSON ne fait pas de distinction entre un entier et un flottant.

3-6-2 - Exemple de document JSON

Dans le cadre de ce tutoriel, on utilisera le document JSON suivant pour vérifier le fonctionnement de la grammaire à écrire :

```
{
  "prénom": "Martine",
  "nom": "Unetelle",
  "enVie": true,
  "âge": 28,
  "sexe": "F",
  "adresse": {
    "NumRue": "21 rue Pasteur",
    "Ville": "Lyon",
    "CodePostal": "F-69000"
  },
}
```

```
"NumérosTéléphone": [
  {
    "type": "domicile",
    "num": "04 05 06 07 08"
  },
  {
    "type": "professionnel",
    "num": "04 08 07 06 05"
  },
  {
    "type": "mobile",
    "num": "06 12 34 56 78"
  }
],
"enfants": [],
"Conjoint": null,
"Profession": "sage-femme",
"Hobbies": ["GRS", "surf", "peinture"]
}
```

Il a été vérifié avec les outils disponibles que ce document respecte bien la norme JSON et l'on peut constater qu'il contient des exemples de tous les types de données décrits plus haut.

3-6-3 - Écrire pas à pas les éléments de la grammaire JSON

3-6-3-1 - Les nombres

L'exemple de document JSON choisi ne comprend qu'un nombre entier, mais la description de l'entité nombre (§ 3.6.1.) montre qu'il faut pouvoir reconnaître des nombres ayant par exemple les formats suivants : « 17 », « -138.27 », « 1.2e-3 », etc.

Ceci peut se traduire par la règle suivante :

```
token nombre {
  [\+|\-]?      # signe optionnel
  [<[0..9]>+ ]   # chiffres optionnels (partie entière)
  [ \. <[0..9]>+ ]? # séparateur décimal et partie fractionnaire
  [<[eE]> [\+|\-]? <[0..9]>+ ]? # exposant optionnel
}
```

3-6-3-2 - Les chaînes de caractères

On peut imaginer de très nombreuses façons de définir une chaîne par un motif. Pour l'exemple de document JSON choisi, la règle suivante sera suffisante :

```
token chaîne {
  \" <[ \w \s - ]>+ \" # caractères alphanumériques, espaces et
                        # tirets, le tout entre des guillemets
}
```

Pour un vrai analyseur JSON, on pourrait préférer une règle utilisant une classe de caractères négative excluant tout ce qui ne peut pas appartenir à une chaîne, par exemple :

```
token chaîne {
  \" <-[\\n \" \\t]>+ \" # tous caractères sauf guillemets, retours à
                        # la ligne et tabulations, entre guillemets
}
```

Il faudrait se pencher sur les détails de la norme JSON pour affiner précisément ce qui est acceptable ou non dans une chaîne JSON. La première règle citée ci-dessus suffira pour l'exemple choisi.

3-6-3-3 - Les objets JSON

Les objets JSON sont des listes de paires clef/valeur. Les listes sont encadrées par des accolades et les paires sont séparées entre elles par des virgules. Une paire clef/valeur est un identifiant (une chaîne, règle déjà définie ci-dessus), suivi d'un caractère deux-points, suivi d'une valeur. Cela peut se traduire ainsi :

```
rule objet      { '{' <listepaires> '}' }
rule listepaires { <paire> * % \,   }
rule paire      { <chaîne> ':' <valeur> }
```

Pour la compréhension de { <paire> * % \, }, il est bon de rappeler que le modificateur « % » appliqué à un quantificateur, permet de spécifier un séparateur qui doit être présent entre les reconnaissances répétées (voir § 2.4.). Ce modificateur permet ici de spécifier aisément que les paires sont séparées par des virgules. Sans ce modificateur, il aurait fallu écrire, par exemple :

```
rule listepaires { \s* | [<paire> [',' <paire>]* ] }
```

ce qui est un peu moins commode et un peu moins lisible.

3-6-3-4 - Les tableaux JSON

Les tableaux sont des listes de valeurs. Les listes sont encadrées par des crochets et les valeurs séparées par des virgules :

```
rule tableau      { '[' <listeTableaux> ']' }
rule listeTableaux { <valeur> * % [ \, ] }
```

3-6-3-5 - Les valeurs

Les valeurs sont soit des objets, soit des tableaux, soit des chaînes, soit des nombres, soit des booléens, soit la valeur *null*.

```
token valeur { | <objet> | <tableau> | <chaîne> | <nombre>
               | true   | false   | null
            }
```

Cette syntaxe fonctionne très bien, mais l'on pourrait pour mémoire envisager une notation syntaxique un peu avancée qui n'a pas encore été abordée dans le présent document, les règles de type *proto* examinées plus loin au § 3.8.3.:

```
proto token valeur {*};
token valeur:sym<nombre> {
  [ '-' | '+' ]?
  [ <[0..9]>* ]
  [ \. <[0..9]>+ ]?
  [ <[eE]> [ \+ | \- ]? <[0..9]>+ ]?
}
token chaîne {
  \" <[ \w \s - ]>+ \"
}
token valeur:sym<true>   { <sym>   };
token valeur:sym<false> { <sym>   };
token valeur:sym<null>  { <sym>   };
token valeur:sym<objet> { <objet> };
token valeur:sym<tableau> { <tableau> };
token valeur:sym<chaîne> { <chaîne> }
```

Cette notation beaucoup moins concise présente l'avantage de faciliter l'extension d'une grammaire, mais l'intérêt est ici assez limité dans la mesure où la norme JSON est stricte et relativement immuable. La première règle citée ci-dessus suffira pour l'exemple choisi.

3-6-4 - La grammaire JSON

Il est maintenant possible d'assembler les différents éléments décrits ci-dessus et d'ajouter la règle **TOP** pour écrire l'ensemble de la grammaire :

```
grammar JSON-Grammaire {
  token TOP      { ^ \s* [ <objet> | <tableau> ] \s* $ }
  rule objet     { '{' <listepaires> '}' }
  rule listepaires { <paires> * % \, }
  rule paire     { <chaîne> ':' <valeur> }
  rule tableau   { '[' <listeTableaux> ']' }
  rule listeTableaux { <valeur> * % [ \, ] }
  token chaîne   { \" ([ \w \s - ]>+) \" }
  token nombre {
    [ \+ | \- ]?
    [ <[0..9]>* ]
    [ \. <[0..9]>+ ]?
    [ <[eE]> [ \+ | \- ]? <[0..9]>+ ]?
  }
  token valeur {
    | <objet> | <tableau> | <chaîne> | <nombre>
    | true   | false    | null
  }
}
```

Pour tester, il suffit d'appeler la grammaire :

```
my $reconnu = JSON-Grammaire.parse($chaîne_json);
say ~$reconnu if $reconnu;
```

Ce qui affiche un objet reconnu contenant l'ensemble du JSON d'origine :

```
$ perl6 json_grammaire.pl
{
  "prénom": "Martine",
  "nom": "Unetelle",
  "enVie": true,
  "âge": 28,

  # [ ... affichage abrégé pour des raisons de place ]

  "Profession": "sage-femme",
  "Hobbies": ["GRS", "surf", "peinture"]
}
```

Le document JSON est bien reconnu intégralement. Cette grammaire JSON fonctionne parfaitement sur le document JSON choisi en exemple et tient en moins de 20 lignes. Le lecteur est encouragé à la tester. Il pourra également vérifier qu'en introduisant des erreurs dans le document JSON (par exemple, supprimer une virgule entre deux valeurs d'une liste), la reconnaissance n'a plus lieu.

On pourrait objecter que cette grammaire ne couvre qu'un sous-ensemble de JSON. Ce n'est pourtant pas vraiment le cas. Il n'est certes pas recommandé d'utiliser cette grammaire dans un environnement de production pour analyser du JSON quelconque, puisqu'elle a été réalisée à titre d'exemple pédagogique, mais sans vérifier les détails les plus fins de la norme JSON, mais la grammaire ci-dessus est néanmoins presque complète.

La grammaire du module **JSON::Tiny** de Perl 6, qui peut analyser n'importe quel fichier JSON valide, n'est pas beaucoup plus longue puisqu'elle tient en environ 35 lignes.

3-6-5 - Ajouter des actions

La grammaire JSON fonctionne, mais si l'on imprime l'arbre des objets reconnus, on obtient pour le document JSON utilisé en exemple près de 300 lignes de texte, car il fournit tout le détail de tout ce qui a été reconnu, pas à pas et sous-motif par sous-motif. Cela peut être très utile pour comprendre ce que fait la grammaire (par exemple en cas de dysfonctionnement), mais explorer cet arbre pour extraire les données peut s'avérer assez pénible.

On peut ajouter une classe d'actions permettant de construire un arbre syntaxique abstrait (AST). Le code de la classe ci-dessous est partiellement inspiré de la classe d'actions du module **JSON::Tiny** :

```
class JSON-actions {
  method TOP($/) {
    make $/.values.[0].made;
  };
  method objet($/) {
    make $<listepaires>.made.hash.item;
  }
  method listepaires($/) {
    make $<paires>>.made.flat;
  }
  method paire($/) {
    make $<chaîne>.made => $<valeur>.made;
  }
  method tableau($/) {
    make $<listeTableaux>.made.item;
  }
  method listeTableaux($/) {
    make [$<valeur>.map(*.made)];
  }
  method chaîne($/) { make ~$0 }
  method nombre($/) { make +$/.Str; }
  method valeur($/) {
    given ~$/ {
      when "true"   {make Bool::True;}
      when "false"  {make Bool::False;}
      when "null"   {make Any;} # équivalent d'undef en Perl5
      default       {make $<val>.made}
    }
  }
}
```

Cette classe d'actions utilise une regex nommée `$<val>` dans la méthode `valeur`, ce qui nous conduit à modifier légèrement la règle `valeur` de la grammaire :

```
token valeur {
  | <val=objet> | <val=tableau> | <val=chaîne> | <val=nombre>
  | true       | false        | null
}
```

L'appel de la grammaire peut maintenant se faire comme suit :

```
my $j-actions = JSON-actions.new();
my $reconnu = JSON-Grammaire.parse($chaîne_json, :actions($j-actions));
say $reconnu.made if $reconnu;
```

L'objet `$reconnu.made` contient maintenant un arbre syntaxique abstrait (AST), une structure de données Perl 6 qu'il est désormais facile d'explorer et d'exploiter. L'affichage de cet arbre ci-dessous a été légèrement reformaté à des seules fins de meilleure lisibilité :

```
{
  Conjoint => (Any),
  Hobbies => [GRS surf peinture],
  NumérosTéléphone => [
    {num => 04 05 06 07 08, type => domicile}
```

```

        {num => 04 08 07 06 05, type => professionnel}
        {num => 06 12 34 56 78, type => mobile}
    ],
    Profession => sage-femme,
    adresse => {
        CodePostal => F-69000
        Complément_Adr => (Any),
        NumRue => 21 rue Pasteur,
        Ville => Lyon
    },
    enVie => True,
    enfants => [0],
    nom => Unetelle,
    prénom => Martine,
    sexe => F,
    âge => 28
}

```

Il est maintenant facile d'accéder aux valeurs individuelles, par exemple :

```
say $reconnu.made<adresse><CodePostal Ville>;      # -> (F-69000 Lyon)
```

3-7 - Une grammaire pour analyser du (pseudo) XML

Un  **chapitre consacré aux grammaires** de l'article  **De Perl 5 à Perl 6 : approfondissements** décrit pas à pas l'écriture d'une grammaire pour analyser un texte respectant un sous-ensemble de la norme XML.

Ce sous-ensemble est défini par la suite de tests suivante :

```

my @tests = (
    [1, 'abc' ], # 1
    [1, '<a></a>' ], # 2
    [1, '..<ab>foo</ab>dd' ], # 3
    [1, '<a><b>c</b></a>' ], # 4
    [1, '<a href="foo"><b>c</b></a>' ], # 5
    [1, '<a empty="" ><b>c</b></a>' ], # 6
    [1, '<a><b>c</b><c></c></a>' ], # 7
    [0, '<' ], # 8
    [0, '<a>b</b>' ], # 9
    [0, '<a>b</a>' ], # 10
    [0, '<a>b</a href="">' ], # 11
    [1, '<a/>' ], # 12
    [1, '<a />' ], # 13
);

```

dans laquelle les chaînes associées à un premier champ égal à 1 sont considérées comme des chaînes XML bien formées, et celles dont le premier champ est à 0 sont considérées comme mal formées.

Le lecteur intéressé est vivement encouragé à aller lire ce chapitre, dont seule la grammaire finale sera citée ici :

```

grammar XML {
    token TOP { ^ <xml> $ };
    token xml { <text> [ <tag> <text> ]* };
    token text { <-[<>&]>* };
    rule tag {
        '<'(\w+) <attributes>*
        [
            | '>' # a single tag
            | '><xml></' $0 '>' # an opening and a closing tag
        ]
    };
    token attributes { \w+ '=' <-["<>"]>* ' ' };
};

```

3-8 - Grammaires : concepts avancés et perspectives

3-8-1 - Les règles paramétrées

Une règle (regex nommée de type regex, token ou rule) peut être définie avec des paramètres et donc appelée avec des arguments :

```
my token date($mois) { \d\d? \s+ $mois \s+ \d**4 }
say "En juin" if "13 juin 2015" ~~ /<date("juin")>/; # -> En juin
say "En mai" if "13 juin 2015" ~~ /<date("mai")>/; # (échec)

my $date = "13 $_ 2015" and say $date ~~ /<date("mai")>/
?? "Date $date en $_"
!! "Date $date pas en $_"
for <mars avril mai juin>;
# affiche :
# Date 13 mars 2015 pas en mars
# Date 13 avril 2015 pas en avril
# Date 13 mai 2015 en mai
# Date 13 juin 2015 pas en juin
```

Voici un exemple un peu gratuit de constructions un peu plus sophistiquées avec des règles paramétrées :

```
my token mots { <[ \w \s \-]>+ }
my token entre-crochets { <start("[")> <ident> <end("]")> }
my token entre-paren { <start("(")> <mots> <end(")")> }
my token entre-chevrons { <start("<")> <ident> <end(">")> }
my token start($début) { $début }
my token end($fin) { $fin }

say ~$<entre-crochets> if "[Capitaine_Crochet]" ~~ /<entre-crochets>/;
# Affiche : [Capitaine_Crochet]

say $<entre-paren> if "[par-parenthèse]" ~~ /<entre-paren>/; # -> ()
# (échec puisqu'il y a des crochets et non des parenthèses)

say $<entre-paren> if "(par-parenthèse)" ~~ /<entre-paren>/;
# Affiche :
# # (par-parenthèse) #
# start => # ( #
# mots => # par-parenthèse #
# end => # ) #
```

3-8-2 - Règles récursives et variables dynamiques

Les règles des grammaires sont par nature très souvent récursives. Par exemple, pour reconnaître des parenthèses imbriquées, on peut vouloir écrire une grammaire récursive de ce style :

```
grammar G { rule TOP { '(' ~ ')' [ [ $<int>=<int>+ ]+ | <TOP> ] + } }
say G.parse("(22 (43 45))");
# Affiche :

# # (22 (43 45)) #
# int => #22#
# TOP => # (43 45) #
# int => #43#
# int => #45#
```

Ici, la règle TOP s'appelle récursivement pour analyser les parenthèses imbriquées. Elle analysera tout aussi facilement une chaîne ayant un niveau d'imbrication plus élevé, comme "(22 (43 (46 45 (41))))".

Cela peut devenir plus délicat quand il faut passer un contexte à la règle appelée récursivement.

On peut en principe utiliser des règles paramétrées (voir § 3.8.1. ci-dessus) pour passer le contexte, mais cela risque de rapidement devenir très verbeux. Il est également en principe possible d'utiliser des variables globales, mais les variables globales sont généralement considérées comme une mauvaise pratique (hors éventuellement certains cas particuliers comme les variables d'environnement, qui sont par nature globales), parce que les variables globales sont souvent contraires aux principes de la programmation structurée, peu claires, peu robustes et dangereuses, et cela ne fonctionne généralement pas avec des threads.

Qu'il s'agisse de grammaires ou de simples appels de fonctions ou de méthodes ordinaires, dès lors que l'on utilise un mécanisme d'appel récursif, la syntaxe peut devenir rébarbative quand il faut gérer plusieurs variables à passer en argument à l'appel de la fonction ou en valeur de retour d'une fonction.

Les variables dynamiques (utilisant le twigil `*`) permettent de résoudre le problème. Elles sont déclarées lexicalement, mais elles sont non seulement cherchées dans les portées *lexicales* englobantes, mais aussi dans les portées *dynamiques* englobantes.

Considérons par exemple une simple fonction récursive de calcul de la factorielle d'un nombre, que l'on peut écrire ainsi :

```
sub fact($n){return 1 if $n <= 1; return $n*fact($n-1)}
sub callfact(Int $n){
    die "Factorielle non définie pour nombre négatif" if $n < 0;
    say "Factorielle = ", fact($n);
}
callfact(5);          # -> 120
```

Supposons que nous voulions aussi calculer au passage la somme des nombres entiers inférieurs au nombre considéré. Le passage de paramètres et de valeurs de retour multiples peut devenir assez vite malcommode et peu lisible. Voici une syntaxe possible stockant la somme partielle dans une variable dynamique :

```
sub fact($n){*$sum += $n; return 1 if $n <= 1; return $n*fact($n-1);}
sub callfact(Int $n){
    die "Factorielle non définie pour nombre négatif" if $n < 0;
    my $*sum = 0;
    say "Fact = ", fact($n); say "Somme = ", $*sum;
}
callfact(5);          # -> Fact = 120 \n Somme = 15
```

La variable dynamique `$*sum` est déclarée et initialisée à 0 dans la fonction appelante `callfact`. Elle est lexicalement locale à la fonction `callfact`, mais elle est visible et modifiable dans la fonction `fact`, parce que `fact` est appelée par `callfact` et réside donc dans la portée dynamique de `callfact`. Cette variable devient en quelque sorte une variable globale à ces deux fonctions (et le cas échéant aux fonctions qu'elles sont susceptibles d'appeler), ce qui dispense de devoir passer des paramètres ou des valeurs de retour entre elles, mais elle demeure une variable lexicale « privée » en ce sens qu'elle est invisible dans le reste du programme.

Une variable dynamique n'est pas nécessairement scalaire, on peut définir de même des tableaux (notés, par exemple, `@*tableau`) ou des hachages dynamiques.

Dès qu'une grammaire devient un peu complexe, les variables dynamiques permettent de simplifier assez nettement la mise en œuvre.

La grammaire **STD.pm** de Perl 6 utilise assez abondamment des variables dynamiques, en particulier pour définir des contextes d'analyse lexicale ou syntaxique qui seront transmis aux fonctions, méthodes ou règles appelées, sans avoir besoin de construire une véritable usine à gaz de passages de paramètres et de valeurs de retour. Par exemple, il existe un hachage dynamique `%*LANG` qui définit les différents « sous-langages » de Perl 6 (Perl 6 de base, regex, regex de Perl 5, etc.) :

```
%*LANG<MAIN>    = ::STD::P6 ;
%*LANG<Q>       = ::STD::Q ;
%*LANG<Quasi>   = ::STD::Quasi ;
```



```
%*LANG<Regex> = ::STD::Regex ;
%*LANG<P5> = ::STD5 ;
%*LANG<P5Regex> = ::STD5::Regex ;
```

3-8-3 - Les règles nommées de type proto

De même qu'il est possible, grâce au mot-clef `multi`, d'écrire plusieurs fonctions ou méthodes *multiples* ayant le même nom, mais que le compilateur pourra distinguer grâce à des signatures différentes, on peut écrire, à l'aide du mot-clef `proto`, des règles ayant le même nom, mais s'appliquant à des entités différentes.

Par exemple, la grammaire Perl 6 actuelle utilise la construction suivante pour définir les sigils :

```
proto token sigil {*}
# ...
token sigil:sym<$> { <sym> }
token sigil:sym<@> { <sym> }
token sigil:sym<%> { <sym> }
token sigil:sym<&> { <sym> }
# ... une petite dizaine d'autres définitions de sigils
```

Ceci crée un sigil nommé de groupe (proto) et (dans cet exemple) quatre règles appartenant à ce groupe (elles appartiennent à ce groupe parce qu'elles portent le même nom) recevant en paramètre l'identifiant `sym`. La première de ces règles affecte `sym` à `$`, puis reconnaît ce symbole dans le corps de la règle (avec la notation `<sym>`). La deuxième règle fait de même avec `@`, et ainsi de suite.

Si une grammaire appelle la règle `<sigil>`, on obtient une liste de ces cinq règles, avec un *ou logique* entre elles. Cela revient à peu près au même que si l'on avait écrit la règle comme suit :

```
token sigil { '$' | '@' | '%' | '&' }
```

mais il devient beaucoup plus facile, avec ces *protorègles* ou *protoregex*, d'étendre la grammaire, comme on le verra ci-dessous (§ 3.8.5.).

Un autre exemple d'utilisation possible de règles de type proto a été donné pour mémoire au § 3.6.3.5.

3-8-4 - Héritage et grammaires mutables

La possibilité d'hériter d'une grammaire offre une puissance d'expression insoupçonnée et des perspectives immenses : il est possible, par exemple dans le cadre d'un module, d'écrire une « sous-grammaire » ou grammaire fille de la grammaire de Perl 6 afin de surcharger un opérateur, d'ajouter une fonctionnalité ou même de modifier un élément de la syntaxe, et de faire tourner un programme Perl avec le même compilateur Perl 6 utilisant cette syntaxe localement modifiée.

C'est grâce à ce mécanisme sous-jacent que la syntaxe de Perl 6 est dynamique et, par exemple, qu'il est facile de définir ses propres opérateurs (voir le chapitre [Créer ses propres opérateurs](#) du tutoriel **De Perl à Perl 6 - Partie 2 : les nouveautés**). Il n'est cependant nullement besoin de maîtriser l'héritage et la mutabilité des grammaires pour créer ses propres opérateurs, car le langage offre un mécanisme simple de plus haut niveau pour le faire, comme le montre cette définition de l'opérateur *factorielle* « `!` » :

```
multi sub postfix:<!>(Int $x) {
  my $factorielle = 1;
  $factorielle *= $_ for 2..$x;
  return $factorielle;
}
say 5!;           # -> imprime 120
```

Les macros de Perl 6 (sortes de fonctions exécutées à la compilation) utilisent également la mutabilité de la grammaire Perl 6 comme mécanisme sous-jacent.

3-8-5 - Modifications de la grammaire et extensibilité du langage

Effectuer une modification de bas niveau de la grammaire pour étendre le langage peut paraître relever de la magie blanche, mais est en fait moins mystérieux qu'il n'y paraît.

En reprenant les règles de type proto permettant de définir les sigils du langage (§ 3.8.3.), il est facile d'ajouter un nouveau sigil à une sous-grammaire de Perl 6. Par exemple, en supposant ici que la grammaire de Perl 6 utilisée par le compilateur s'appelle « Perl6 » (son vrai nom est actuellement plutôt quelque chose du genre STD:ver<xxx>, dans lequel xxx est un numéro de version), on peut ajouter le sigil « µ » :

```
grammar NouveauSigilP6 is Perl6 {  
  token sigil:sym<µ> { <sym> }  
}
```

ou, à l'intention des lecteurs pour lesquels l'utilisation systématique de « \$ » heurte la conviction pro-européenne, de modifier un sigil existant :

```
grammar EuroPerl6 is Perl6 {  
  token sigil:sym<$> { '€' }  
}
```

La grammaire EuroPerl6 permet maintenant d'utiliser le sigil « € » pour les scalaires, mais comme c'est la même règle avec le même paramètre (sigil:sym<\$>) que la grammaire d'origine, le compilateur n'a aucune difficulté à savoir ce qu'il doit faire. Les eurosceptiques britanniques pourront s'ils le désirent adopter le sigil « £ », les mélomanes « ♪ » et les altermondialistes également allergiques au « \$ » la faucille et le marteau (« # ») ou le symbole qui leur plaira.

3-8-6 - Perspectives

Il y a de nombreuses autres possibilités d'extension réellement extraordinaires et stupéfiantes, mais elles sont plus destinées à des *geeks* ou gourous du langage qu'à des utilisateurs ordinaires. Les explorer n'est pas possible ici, car cela nécessiterait sans doute des dizaines de pages supplémentaires et sortirait complètement du cadre de ce tutoriel.

Il y a très peu de documentation actuellement disponible sur le sujet (et d'ailleurs sur les grammaires Perl 6 en général, même en anglais), la source d'information la plus riche est sans doute de consulter la grammaire standard de Perl 6 et des modules existants (et d'expérimenter). Le lecteur intéressé pourra cependant approfondir en consultant les liens suivants : [🇬🇧 The Perl 6 standard grammar](#), de Patrick Michaud, et [🇬🇧 A Mutable Grammar For Perl 6](#) de Moritz Lenz. Ces sources sont malheureusement assez anciennes, mais si certains points de détail peuvent être périmés, la discussion conceptuelle reste parfaitement d'actualité.

4 - Bonnes pratiques et pièges à éviter

Les regex et les grammaires Perl 6 forment à elles seules un véritable modèle de programmation qui est souvent au moins en partie nouveau pour beaucoup et qu'il faut donc apprendre à maîtriser.

Pour aider le lecteur à écrire des regex et des grammaires robustes, voici quelques bonnes pratiques qu'il paraît judicieux d'appliquer (avec bon sens, il ne s'agit pas non plus de règles à suivre aveuglément). Ces bonnes pratiques vont du simple formatage à petite échelle du code à une compréhension fine des reconnaissances, en passant par une assistance pour éviter les pièges possibles et le code illisible.

4-1 - Formatage du code

Lorsque l'adverbe `:sigspace` n'est pas utilisé, les espaces blancs (et les commentaires) sont ignorés dans les regex de Perl 6. Cela offre l'avantage de pouvoir insérer des espaces et des commentaires pour améliorer la lisibilité.

Comparez cette regex très compacte pour reconnaître un nombre à virgule flottante (*float*) :

```
my regex float {<[+-]>?\d*'\.\d+[e<[+-]>?\d+]?}
```

avec celle-ci, équivalente, mais bien plus lisible :

```
my regex float {
  <[+-]>?          # signe optionnel
  \d*            # chiffres de début, optionnels
  '\.'          # séparateur décimal
  \d+
  [
    e <[+-]>? \d+  # exposant optionnel
  ]?
}
```

D'une façon générale, il est souhaitable :

- d'utiliser des espaces autour des atomes et à l'intérieur des groupes ;
- de placer les quantificateurs directement après l'atome, sans insérer d'espace, et ;
- d'aligner verticalement les crochets et parenthèses ouvrants ou fermants.

De même, il convient généralement d'aligner les barres verticales « `|` » séparant les différentes possibilités d'une alternative :

```
my regex exemple {
  <début>
  [
    || <choix_1>
    || <choix_2>
    || <choix_3>
  ]+
  <fin>
}
```

4-2 - Limiter la taille

Les regex ont besoin de très peu de fioritures ou de code bateau aux alentours, elles sont donc souvent nettement plus compactes que du code ordinaire. Il importe de les garder assez petites.

Quand le nombre de captures devient élevé ou quand on en vient à utiliser des captures nommées pour mieux s'y retrouver, il convient de se demander si l'on ne ferait pas mieux de franchir un pas de plus et de passer à des regex nommées.

4-2-1 - Reconnaître un nombre à virgule flottante

Par exemple, la regex `float` du chapitre précédent (§ 4.1.) pourrait être décomposée en parties plus petites :

```
my token signe { <[+-]> }
my token décimal { \d+ }
my token exposant { 'e' <signe>? <décimal> }
my regex float {
  <signe>?
```

```
<décimal>?
','
<décimal>
<exposant>?
}
```

Cela aide quand la regex devient plus complexe. Par exemple, si l'on désire rendre le point décimal (la « virgule ») optionnel quand il y a un exposant :

```
my regex float {
  <signe>?
  [
    || <décimal>? ',' <décimal> <exposant>?
    || <décimal> <exposant>
  ]
}
```

Cela permet aussi une meilleure réutilisation du code. Les règles (*tokens*) `signe` et `décimal` définies ci-dessus permettent aussi de définir très simplement un entier :

```
my regex entier {
  <signe>?
  <décimal>
}
```

4-2-2 - Reconnaître un nombre complexe

De même, toujours en réutilisant les règles (*tokens*) `signe` et `décimal` définies précédemment, on pourrait définir un nombre complexe (en notation algébrique cartésienne) comme suit :

```
my rule nombre { <float> || <signe>? <décimal> }
my regex complexe {
  [
    || [<nombre> \s* <signe> \s* ]? <nombre> \s* 'i'
    || <nombre>
  ]
}

say "Reconnu" if '3+4i' ~~ /<complexe>/; # -> Reconnu
# Accéder aux différents éléments du nombre complexe reconnu :
say ~$/<complexe>
  if '3.5e-7 + 4.17i' ~~ /<complexe>/; # 3.5e-7 + 4.17i
say ~$/<complexe><nombre>[0];          # -> 3.5e-7
say ~$/<complexe><nombre>[1];          # -> 4.17
say ~$/<complexe><nombre>[0]<float>    # -> 3.5e-7
say ~$/<complexe><signe>              # -> +
```

On voit bien ici que, même sans utiliser de grammaire (du moins pour ce genre de cas très simple), on peut construire les briques d'un véritable jeu de Lego avec les règles nommées. La reconnaissance d'une adresse IP en donnera à nouvel exemple.

4-2-3 - Reconnaître une URL

On considérera ici en simplifiant qu'une URL (une adresse Web) peut être soit une adresse IP (séries de quatre nombres séparés par des points), soit d'une chaîne de caractères représentant le protocole et le nom de domaine, suivie du chemin d'accès de la ressource.

On peut commencer par essayer de reconnaître une adresse IP.

4-2-4 - Reconnaître une adresse IP

Une adresse IPv4 se compose de quatre nombres de 1 à 3 chiffres séparés par des points.

Une première tentative de regex pour reconnaître une adresse IP pourrait être :

```
/(\d**1..3) \. (\d**1..3) \. (\d**1..3) \. (\d**1..3)/
```

Mais c'est pour le moins assez laborieux et maladroit.

Le modificateur % (voir § 2.4. ci-dessus), appliqué à un quantificateur, permet de spécifier un séparateur qui doit être présent entre les reconnaissances répétées :

```
/ (\d ** 1..3) ** 4 % '.' /
```

C'est déjà beaucoup mieux, mais c'est malheureusement d'un certain point de vue faux : cette regex va reconnaître sans problème la chaîne « 125.266.742.12 », qui n'est pas une adresse IP valide (chacun des quatre nombres doit représenter un octet et donc être compris, en notation décimale, entre 0 et 255). Tout dépend bien sûr de ce que l'on cherche à faire exactement, la regex ci-dessus peut suffire à capturer ce que l'on cherche, mais elle n'assure pas complètement la validation de la donnée en entrée.

Pour valider que l'on capture bien des octets (nombres inférieurs à 255), on peut construire une regex (ou un *token*) octet qui vérifiera ces conditions, puis une regex ip utilisant octet :

```
my regex octet {
    || (25 <[0..5]>                # 250 à 255
    || 2 <[0..4]> \d                # 200 à 249
    || 1 \d**2                     # 100 à 199
    || \d**1..2)                  # 0 à 99
}
my regex ip { <octet> ** 4 % '.' }
say "Reconnu" if "244.7.245.23" ~~ /<ip>/; # -> Reconnu
say ~$/;                                   # -> 244.7.245.23
```

À noter que l'on pourrait simplifier notablement la regex `octet` en utilisant une assertion de type code (fermeture) :

```
my regex octet { (\d ** 1..3) <?{0 <= $0 <= 255 }> }
```

Ces deux dernières versions de la regex `octet` souffrent encore d'un défaut qui risque de se manifester dans certains cas d'utilisation sur le dernier octet de la chaîne : dans la chaîne « 244.7.245.263 » (qui n'est pas une adresse IP correcte puisque le dernier nombre est supérieur à 255), la regex `IP` va cependant reconnaître une adresse IP apparemment correcte, mais sans doute non désirée : « 244.7.245.26 ». Pour éviter ce problème, on peut ajouter une assertion avant négative (<!before ...>) dans la partie de la définition d'octet concernant les octets à moins de trois chiffres :

```
my regex octet {
    || (25 <[0..5]>                # 250 à 255
    || 2 <[0..4]> \d                # 200 à 249
    || 1 \d**2                     # 100 à 199
    || \d**1..2) <!before \d>    # 0 à 99
}
my regex ip { <octet> ** 4 % '.' }
```

Ici encore, le cas d'une simple adresse IP est assez trivial, on peut sans doute se passer d'écrire une grammaire, car la combinaison des regex ci-dessus est largement suffisante.

Mais pourquoi devrait-on éviter la définition d'une grammaire ? Écrire et utiliser une grammaire n'est guère plus compliqué (à condition d'en prendre l'habitude) qu'assembler une série de règles nommées et permettra d'étendre

plus facilement la syntaxe à d'autres éléments tels qu'une URL. Cela permettra en outre, le cas échéant, de bénéficier d'avantages complémentaires associés aux grammaires (espace de nom confiné, héritage, méthodes-actions, méthodes .parse et .fileparse, etc.)

4-2-5 - Une grammaire pour reconnaître une URL

Une grammaire (assez sommaire) pour analyser une URL peut avoir la forme suivante :

```
grammar URL {
  token TOP {
    <schéma> '://'
    [<ip> | <nom-domaine> ]
    [ ':' <port> ]?
    '/' <chemin>?
  }
  token octet {
    (\d**1..3) <?{ $0 < 256 }>
  }
  token ip {
    <octet> [\ . <octet> ] ** 3
  }
  token schéma {
    \w+ # Ce pourrait aussi être : [http | https | ftp | ...]
  }
  token nom-domaine {
    (\w+) ( \ . \w+ ) *
  }
  token port {
    \d+
  }
  token chemin {
    <[ a..z A..Z 0..9 \_ - ! ~ * ' ( ) : @ & = + $ , / ] > +
  }
}

my $cible = URL.parse('http://perl6.org/documentation/');
say $cible<nom-domaine>; # -> perl6.org
```

4-3 - Que reconnaître ?

Souvent, le format des données en entrée n'est pas clairement spécifié, ou la spécification, si elle existe, n'est pas connue du programmeur. Il est alors souvent utile d'être assez libéral ou souple sur ce que l'on attend, mais seulement dans la mesure où il n'y a pas de risque d'ambiguïté.

Si l'on reprend l'exemple du fichier .ini :

```
[section]
key=value
```

Que peut-il y avoir dans l'entête de section ? N'autoriser qu'un seul mot est peut-être trop restrictif, peut-être quelqu'un va écrire [deux mots], ou utiliser des tirets, ou Dieu seul sait quoi d'autre... Plutôt que se demander ce qui est autorisé à l'intérieur de cet entête, il peut être utile de se demander ce qui n'est pas permis.

Clairement, un crochet fermant est interdit, parce que [a]b] serait pour le moins ambigu. Du même point de vue, un crochet ouvrant doit être proscrit. Ce qui peut nous donner la règle suivante :

```
token entête { '[' <- [ \ [ \ ] ] >+ ']' }
```

ce qui semble bien fonctionner si l'on analyse une seule ligne. Mais si l'on traite un fichier entier, soudain la règle analyse :

```
[ avec un
retour à la ligne entre deux]
```

comme un entête correct, ce qui n'est sans doute pas une bonne idée. Un compromis pragmatique pourrait être d'écrire :

```
token entête { '[' <-[ \[\] \n ]>+ '[' }
```

puis, dans le post-traitement, d'éliminer les espaces, tabulations, etc. au début et à la fin de l'entête de section.

4-4 - Reconnaître des espaces blancs

L'adverbe `:sigspace` (ou l'utilisation d'une règle de type `rule`, plutôt que `regex` ou `token`) est très pratique pour l'analyse d'espaces qui peuvent apparaître en de multiples endroits.

Si l'on revient à l'exemple d'analyse de fichiers ini, la règle peut être la suivante :

```
my regex kvpair { \s* <clef=identifiant> '=' <val=identifiant> \n+ }
```

Cela fonctionne en principe, mais n'est peut-être pas aussi flexible qu'on pourrait le désirer. Comme l'utilisateur peut décider de mettre des espaces autour du signe égal, peut-être faudrait-il plutôt :

```
my regex kvpair { \s* <clef=identifiant> \s* '=' \s* <val=identifiant> \n+ }
```

Cela peut rapidement devenir peu pratique. On pourrait donc tenter d'utiliser une `rule` au lieu d'une `regex` et d'écrire :

```
my rule kvpair { <clef=identifiant> '=' <val=identifiant> \n+ }
```

Mais attention ! La reconnaissance implicite des espaces après la valeur consomme tous les espaces disponibles, y compris les caractères de retour à la ligne, si bien que le `\n+` final n'a plus rien à reconnaître. Et comme une `rule` désactive le retour arrière, cela ne marche pas.

C'est là qu'il peut être fort utile de redéfinir l'espace implicite comme un espace qui n'est pas significatif dans le format en entrée, ce que l'on peut faire en redéfinissant le `token ws` (mais cela ne fonctionne qu'à l'intérieur de grammaires) :

```
grammar IniFormat {
  token ws { <!ww> \h* }
  rule entête { '[' (\w+) '[' \n+ }
  token identifiant { \w+ }
  rule kvpair { \s* <clef=identifiant> '=' <val=identifiant> \n+ }
  token section {
    <entête>
    <kvpair>*
  }

  token TOP {
    <section>*
  }
}

# Exemple de fichier ini dans un document "ici même" :
my $contenu = q:to/FIN_INI/;
[passwords]
  jean=mdpl
  anne=plusfiable123
[quotas]
  jean=123
  anne=42
FIN_INI

say so IniFormat.parse($contenu);
```

Outre l'idée de mettre toutes les regex dans une grammaire et de les transformer en tokens ou rules (car ces regex n'ont pas besoin de retour arrière de toute façon), le point intéressant et nouveau est la définition suivante :

```
token ws { <!ww> \h* }
```

qui est appelée lors de l'analyse implicite des espaces. Il reconnaît zéro ou plusieurs espaces horizontaux qui ne sont pas entre deux caractères de type mot (<!ww>, négation de l'assertion « within word »). La limitation à des espaces horizontaux est essentielle, parce que les caractères de retour à la ligne (qui sont des espaces verticaux) délimitent les enregistrements et ne doivent donc pas être reconnus implicitement comme simples espaces.

Il peut encore y avoir un problème lié aux espaces au détour du chemin. Le regex `\n+` ne va pas reconnaître une chaîne telle que `\n\n`, parce qu'il existe un espace entre les deux retours à la ligne. Pour reconnaître ce genre de chaînes en entrée, il est possible de remplacer `\n+` par `\ns*`.

4-5 - Déboguer des regex ou des grammaires Perl 6

Quand on commence à apprendre à programmer, on perd beaucoup de temps avec de petites erreurs stupides. Avec l'expérience, on apprend à faire moins d'erreurs et à écrire plus rapidement du code qui fonctionne.

Avec les grammaires (et plus généralement les regex), tout semble recommencer comme au début : même des programmeurs expérimentés recommencent à faire des erreurs stupides quand il s'attaquent aux grammaires. Écrire une regex et plus encore une grammaire n'a pas grand-chose à voir avec le développement d'un programme procédural ordinaire et nécessite une nouvelle phase d'apprentissage.

Voici quelques pistes pour aider à écrire et déboguer des grammaires (qui rejoignent en partie certains conseils déjà donnés ci-dessus) :

- **avancez par petites étapes**, règle par règle, et testez les règles au fur et à mesure ;
- **testez les règles individuellement** : si votre grammaire ne fonctionne pas, testez chaque règle une par une afin de déterminer si une règle est erronée, mal appelée (ou jamais appelée), etc. :

```
grammar MaGrammaire {
  token TOP {
    ^ [ <commentaire> | <truc> ]* $
  }

  token commentaire {
    '#' \N* $$
  }

  token truc {
    ^^(\S+) \= (\S+) $$
  }
}

# Essayer de parser l'ensemble:
say ?MaGrammaire.parse("#commentaire\ntoto = titi");      # 0
# La grammaire ne reconnaît pas le test, voyons les règles une à une
say ?MaGrammaire.parse("#commentaire\n", :rule<commentaire>); # 1 - OK
say ?MaGrammaire.parse("toto = titi", :rule<truc>);         # 0 - KO
# C'est la règle <truc> qui ne fonctionne pas.
```

- **insérez des affichages** (print ou say) ; il suffit de les mettre entre des accolades pour que ces affichages s'exécutent comme du code ordinaire (voir § 3.4.2.). Reprenons l'exemple précédent là où nous étions arrivés (la règle <truc> ne fonctionne pas) :

```
grammar MaGrammaire {
  token truc {
    { say "truc: appelé" }
    ^^
    { say "truc: trouvé début de ligne" }
    (\S+)
  }
}
```



```

        { say "truc: trouvé premier identifiant: $0" }
        \=
        { say "truc: trouvé =" }
        (\S+) $$
    }
}

say ?MaGrammaire.parse("toto = titi", :rule<truc>);

# Affichage:
#
# truc: appelé
# truc: trouvé début de ligne
# truc: trouvé premier identifiant: toto
# 0
# C'est le signe égal qui n'est pas reconnu. Pourquoi? À cause
# de l'espace qui le précède et qui n'est pas dans la règle. Il
# suffit par exemple de transformer le token en rule

```

- **Attention au contrôle du retour arrière** : beaucoup de programmeurs habitués à Perl 5 ou à des systèmes de regex apparentés sont rompus à l'utilisation du retour arrière, très naturelle et très puissante dans de simples regex de petite dimension. Mais le retour arrière devient très vite difficile à maîtriser dans une grammaire (ou même un ensemble de regex) imbriquée. La plupart des problèmes d'analyse lexicale et syntaxique peuvent se formuler d'une façon qui ne nécessite pas (ou très peu) de retour arrière, et il est donc fortement recommandé d'éviter le retour arrière dans les grammaires, aussi bien pour des raisons d'efficacité que par mesure de sauvegarde de la santé mentale du développeur. Cela dit, certains motifs sont bien plus faciles à écrire avec du retour arrière ; si vous les utilisez, veillez à en limiter la portée à la seule regex en ayant besoin, soit en définissant une sous-regex séparée pour la seule partie nécessitant le retour arrière, soit en utilisant une règle de type rule ou token n'autorisant pas le retour arrière et en l'autorisant pour la seule partie en ayant besoin :

```

rule verbatim {
    ' [% ' ~ ' % ] ' verbatim
    # Autorise le retour arrière à partir d'ici seulement
    :!ratchet
    .*? ' [% ' endverbatim ' % ] '
}
# Le retour arrière sera activé dans la regex finale, mais, dès
# qu'une reconnaissance aura été trouvée, on n'en essaiera pas d'autre

```

- Signalons enfin l'**excellent module de débogage de grammaires et de regex** de Jonathan Worthington sous Rakudo/Perl 6. Ajoutez un `use Regex::Tracer;` dans votre code, et toutes vos grammaires dans la portée lexicale afficheront en couleur des informations de débogage détaillées, montrant notamment quelles règles ont été appelées, lesquelles ont fonctionné et lesquelles ont échoué. 🇬🇧 **Le Calendrier de l'Avent Perl 6** donne des exemples et plus de détails.



Les informations de la présente section **4.5 Déboguer des regex ou des grammaires Perl 6** sont empruntées pour l'essentiel au document 🇬🇧 **How to Debug a Perl 6 Grammar** de Moritz Lenz.



5 - Conclusion


Les regex et les grammaires de Perl 6 sont loin d'être la seule nouveauté de Perl 6, mais elles ouvrent à elles seules des perspectives radicalement nouvelles, aussi bien pour l'analyse de documents textuels qu'en termes de richesse fonctionnelle, d'expressivité et d'évolutivité du langage. Il est vraisemblable que l'on trouvera avec l'expérience des utilisations auxquelles il serait difficile de penser aujourd'hui. Les grammaires font à notre avis partie de ce qui devrait faire de Perl 6 un langage qui restera résolument moderne pour peut-être 20 ans ou plus.

6 - Voir aussi/Sources

La documentation sur Perl 6 a fait des progrès considérables depuis un an, mais reste parfois incomplète, et même exceptionnellement très incomplète.


Le chapitre **2. Les regex de Perl 6** du présent document est dans une large mesure une adaptation française de la documentation officielle  **Regexes** (auteur anonyme), à laquelle nous avons ajouté des informations provenant de la *Synopsis*  **S05** et de quelques autres documents périphériques de la documentation officielle, d'articles trouvés ici ou là, et des éléments issus des expérimentations personnelles. Ce chapitre du présent document est certainement à ce jour le document le plus complet sur le sujet, langues française et anglaise confondues.

La documentation officielle en anglais sur les grammaires ( **Grammars**) est actuellement très très incomplète (elle couvre à peu près correctement les règles nommées et les classes d'actions, mais, à l'heure où nous écrivons, la partie sur les grammaires proprement dites tient en trois lignes !). La *Synopsis*  **S05** semble assez datée et n'apporte pas grand-chose de plus sur ce sujet. Nous avons mentionné ici ou là quelques documents sur des points de détail intéressants, mais il n'existe clairement aucune documentation officielle digne de ce nom sur les grammaires Perl 6. L'examen du code source de Perl 6 et les expérimentations personnelles sont donc les principales sources de ce document. Il en résulte que le chapitre **3. Grammaires** du présent document n'est certainement pas exhaustif (il y a peut-être des fonctionnalités entières qui nous ont échappé), mais c'est, pour autant que nous puissions le dire, le document le plus complet disponible aujourd'hui sur le sujet, et ce de très loin.

Le chapitre **4. Bonnes pratiques et pièges à éviter** est également en partie une adaptation française de la documentation officielle  **Regexes** (document anonyme), enrichie de divers éléments trouvés ici ou là ou issus de réflexions personnelles.

En bref, selon les sujets, soit nous avons repris à peu près tout ce que nous pouvions trouver dans la documentation existante (en français ou en anglais, voire en allemand), soit nous avons créé la première documentation sérieuse. Il en résulte que nous ne pouvons pas, à ce stade, proposer de lectures complémentaires permettant d'approfondir ces sujets en plus des liens déjà fournis au fil du texte.

Dans ces circonstances, nous mettons le présent document en licence *Creative Common* « Attribution CC BY » afin d'autoriser quiconque le désirerait à en reprendre librement le contenu, tout en souhaitant si possible que la source et son auteur soient cités.

Il faut noter cependant que la documentation Perl 6 s'est beaucoup améliorée ces derniers mois et l'on peut penser qu'elle va continuer à s'enrichir rapidement. Il se peut donc que ce que nous écrivons (fin 2015) sur les lacunes actuelles de la documentation soit inexact d'ici quelques semaines ou quelques mois. Le lecteur est donc encouragé à aller vérifier dans la  **documentation officielle Perl 6** s'il trouve ce qu'il cherche.

7 - Remerciements

Je remercie les auteurs anonymes de  **la documentation officielle Perl 6** dont une partie non négligeable du présent document est une adaptation en français.

Je remercie **Djibril**, **Roland Chastain**, **Claude Leloup** et **Cognominal** pour leur relecture et leurs très utiles suggestions d'amélioration.