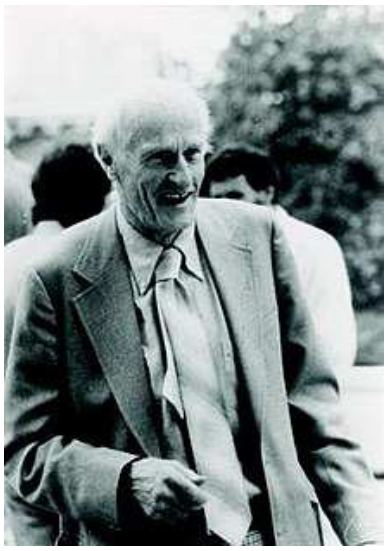
WIKIPÉDIA

Expression régulière

En informatique, une **expression régulière** ou **expression normale**[note 1] ou **expression rationnelle** ou **motif**, est une chaîne de caractères, qui décrit, selon une syntaxe précise, un ensemble de chaînes de caractères possibles. Les expressions régulières sont également appelées *regex*. Les expressions rationnelles sont issues des théories mathématiques des langages formels des années 1940. Leur capacité à décrire avec concision des *ensembles réguliers* explique qu'elles se retrouvent dans plusieurs domaines scientifiques dans les années d'après-guerre et justifie leur adoption en informatique. Les expressions régulières sont aujourd'hui utilisées pour programmer des logiciels avec des fonctionnalités de lecture, de contrôle, de modification, et d'analyse de textes ainsi que dans la manipulation des langues formelles que sont les langages informatiques.

Ces expressions *régulières* ont la qualité de pouvoir être décrites par des formules ou motifs, (en anglais patterns) bien plus simples que les autres moyens[1].



Stephen Cole Kleene, dont les travaux ont fait émerger l'utilisation du concept d'expression régulière.

Sommaire

Histoire

Dans les années 1940, Warren McCulloch et Walter Pitts ont décrit le système nerveux en modélisant les neurones par des automates simples. En 1956, le logicien Stephen Cole Kleene[2, note 2] a ensuite décrit ces modèles en termes d'*ensembles réguliers* et d'automates. Il est considéré comme l'inventeur des expressions régulières. En 1959, Michael Rabin et Dana Scott proposent le premier traitement

mathématique et rigoureux de ces concepts[3], ce qui leur vaudra le prix Turing en 1976.

Dans ce contexte, les expressions régulières correspondent aux grammaires de type 3 (voir Grammaire formelle) de la hiérarchie de Chomsky ; elles peuvent donc être utilisées pour décrire la morphologie d'une langue.

Ken Thompson a mis en œuvre la notation de Kleene dans l'éditeur qed, puis l'éditeur ed sous Unix, et finalement dans grep. Depuis lors, les expressions régulières ont été largement utilisées dans les utilitaires tels que lex ainsi que dans les langages de programmation nés sous Unix, tels que expr, awk, Perl, Tcl, Python, etc.

En sortant du cadre théorique, les expressions régulières ont acquis des fonctionnalités permettant de décrire des langages non rationnels. Un glissement sémantique s'est ainsi produit : la notion d'expression régulière n'a pas le même sens dans le contexte de l'informatique appliquée et dans la théorie des langages formels.

Utilisation

Initialement créées pour décrire des langages formels, les expressions régulières sont utilisées dans l'analyse et la manipulation des langages informatiques ; compilateurs et interprètes sont ainsi basés dessus.

Exemples d'expressions régulières

Expression régulière	Mots décrits	Mots non décrits
détecté	« détecté »	« détect », « détecta », « détectés », « »
ex-(a?e|æ|é)quo	« ex-équo », « ex-equo », « ex-aequo » et « ex-æquo »	« ex-quo », « ex-aiquo », « ex-aeko », « ex-ææquo »
^Section .+	« Section 1 », « Section 22 », « Section A », …	« voir Section 1 », « Sectionner »
6,66*$	« 6,6 », « 6,666 », « 6,6666 », …	« 6,66667 »,
[1234567890]+(,[1234567890]+)?	« 2 », « 42 », « 0,618 », « 49,3 », …	« 3, », « ,75 » , « »

Utilisée à la manière des outils de recherche de texte dans un document, une expression régulière décrit des chaînes de caractères ayant des propriétés communes, dans le but de les trouver dans un bloc de texte pour leur appliquer un traitement automatisé, comme un ajout, leur remplacement, leur modification ou leur suppression.

Beaucoup d'éditeurs de texte et la plupart des environnement de développement intégrés permettent de mettre en œuvre les expressions régulières. Un grand nombre d'utilitaires Unix savent les utiliser nativement. Les plus connus desquels étant GNU grep ou GNU sed qui, à la manière des éditeurs de texte, utilisent ces expressions pour parcourir de façon automatique un document à la recherche de morceaux de texte compatibles avec le motif de recherche, et éventuellement effectuer un ajout, une substitution ou une suppression.

Les interface en ligne de commande (ou shells) utilisent un système apparenté mais distinct et moins expressif appelé glob `(en)` ou globbing.

Les expressions régulières sont fréquemment employées dans les activités d'administration système, de développement logiciel et de traitement automatique du langage naturel. Elles ont vu un nouveau champ d'application avec le développement d'Internet, et la diffusion de code malveillant ou de messages pourriels. Des filtres et des robots utilisant ces expressions sont utilisés pour détecter les éléments potentiellement nuisibles.

En théorie des langages formels, une expression régulière est une expression représentant un langage rationnel. Dans ce contexte, les expressions régulières ont un pouvoir expressif plus limité : cette notion a un sens plus large en informatique appliquée qu'en théorie des langages formels.

Principes de base

Une expression régulière est une suite de caractères typographiques (qu'on appelle plus simplement « motif » – « *pattern* » en anglais) décrivant un ensemble de chaînes de caractères. Par exemple l'ensemble de mots « ex-équo, ex-equo, ex-aequo et ex-æquo » peut être condensé en un seul motif « ex-(a?e|æ|é)quo ». Les mécanismes de base pour former de telles expressions sont basés sur des caractères spéciaux de substitution, de groupement et de quantification.

Une barre verticale sépare le plus souvent deux expressions alternatives : « equo|aequo » désigne soit equo, soit aequo. Il est également possible d'utiliser des parenthèses pour définir le champ et la priorité de la détection, « (ae|e)quo » désignant le même ensemble que « aequo|equo » et de quantifier les groupements présents dans le motif en apposant des caractères de quantification à droite de ces groupements.

Les quantificateurs les plus répandus sont :

- ? qui définit un groupe qui existe zéro ou une fois : `toto?` correspondant alors à « tot » ou « toto » mais pas « totoo » ;
- * qui définit un groupe qui existe zéro ou plusieurs fois (l'étoile de Kleene) : `toto*` correspondant à « tot », « toto », « totoo », « totooo », etc. ;
- + qui définit un groupe qui existe une ou plusieurs fois : `toto+` correspondant à « toto », « totoo », « totooo », etc. mais pas « tot ».

Les symboles dotés d'une sémantique particulière peuvent être appelés « opérateurs », « métacaractères » ou « caractères spéciaux ». Les caractères qui ne représentent qu'eux-mêmes sont dits « littéraux ».

Les expressions régulières peuvent être combinées, par exemple par concaténation, pour produire des expressions régulières plus complexes.

Lorsqu'une chaîne de caractère correspond à la description donnée par l'expression régulière, on dit qu'il y a « correspondance » entre la chaîne et le motif, ou que le motif « reconnaît » la chaîne. Cette correspondance peut concerner la totalité ou une partie de la chaîne de caractères. Par exemple, dans la phrase « Les deux équipes ont terminé ex-æquo et se sont saluées. », la sous-chaîne « ex-æquo » est reconnue par le motif « ex-(a?e|æ|é)quo ».

Par défaut, les expressions régulières sont sensibles à la casse. Lorsque c'est possible, elles tentent de reconnaître la plus grande sous-chaîne correspondant au motif : on dit qu'elles sont « gourmandes ». Par exemple, `Aa+` reconnaît la totalité de la chaîne « Aaaaaaa » plutôt qu'une partie « Aaa » (gourmandise), mais elle ne reconnaît pas la chaîne « aaaA » (sensibilité à la casse).

Opérateurs de base

Opérateurs	Description	Exemples		
		Expression régulière	Chaînes décrites	Chaînes non décrites
expr$_1$ *expr*$_2$	Opérateur de concaténation de deux expressions (implicite).	ab	« ab »	« a », « b », chaîne vide
.	Un caractère et un seul	.	« a », « b », ...	chaîne vide, « ab »
expr?	Ce quantificateur correspond à ce qui le précède, présent **zéro ou une fois**. Si de multiples correspondances existent dans un texte, il trouve d'abord ceux placés en tête du texte et retourne alors la plus grande longueur possible à partir de cette position initiale.	a?	chaîne vide, « a »	« aa », « b »
expr+	Ce quantificateur correspond à ce qui le précède, répété **une ou plusieurs fois**. Si de multiples correspondances existent dans un texte, il trouve d'abord ceux placés en tête du texte et retourne alors la plus grande longueur possible à partir de cette position initiale.	a+	« a », « aa », « aaaaa », ...	chaîne vide, « b », « aaab »
*expr**	Ce quantificateur correspond à ce qui le précède, répété **zéro ou plusieurs fois**. Si de multiples correspondances existent dans un texte, il trouve d'abord ceux placés en tête du texte et retourne alors la plus grande longueur possible à partir de cette position initiale.	a*	chaîne vide, « a », « aaa », ...	« b », « aaab »
expr$_1$|*expr*$_2$	C'est l'opérateur de choix entre plusieurs alternatives, c'est-à-dire l'union ensembliste. Il peut être combiné autant de fois que nécessaire pour chacune des alternatives possibles. Il fait correspondre **l'une des expressions placées avant ou après l'opérateur**. Ces expressions peuvent éventuellement être vides, et donc (x|) équivaut à x?.	a|b	« a », « b »	chaîne vide, « ab », « c »
[*liste*]	Un des caractères entre crochets (« classe de caractères »)	[aeiou]	« a », « e », « i », ...	chaîne vide, « b », « ae »
[^*liste*]	Un caractère n'étant pas entre crochets (« classe de caractères »)	[^aeiou]	« b », ...	chaîne vide, « a », « bc »
(*expr*)	Groupement de l'expression entre parenthèses	(détecté)	« détecté »	« détect », « détecta », « détectés »
expr{*n*}	Exactement *n* occurrences de l'expression précédant les accolades	a{3}	« aaa »	« aa », « aaaa »
expr{*n*,*m*}	Entre *n* et *m* occurrences de l'expression précédant les accolades	a{2,4}	« aa », « aaa », « aaaa »	« a », « aaaaa »
expr{*n*,}	Au moins *n* occurrences de l'expression précédant les accolades	a{3,}	« aaa », « aaaa », « aaaaa »	« aa »
^	Ce prédicat ne correspond à aucun caractère mais fixe une condition nécessaire permettant de trouver un accord sur ce qui le suit en indiquant que ce doit être au **début d'une ligne** (donc être au début du texte d'entrée ou après un saut de ligne). Il ne peut être considéré ainsi qu'au début de l'expression régulière, ailleurs il est considéré littéralement. Il s'applique comme condition à la totalité du reste de l'expression régulière (et concerne donc toutes les alternatives représentées).	^a trouve « a » en début de ligne mais pas dans « ba ».		
$	Ce prédicat ne correspond à aucun caractère mais fixe une condition nécessaire permettant de trouver un accord sur ce qui le précède en indiquant que ce doit être à **la fin d'une ligne** (donc être à la fin du texte d'entrée ou juste avant un saut de ligne). Il ne peut être considéré ainsi qu'à la fin de l'expression régulière, ailleurs il est considéré littéralement. Il s'applique comme condition à la totalité du reste de l'expression régulière (et concerne donc toutes les alternatives représentées).	a$ trouve « a » en fin de ligne mais pas dans « ab ».		

Standards

Dans le domaine de l'informatique, un outil permettant de manipuler les expressions régulières est appelé un *moteur d'expressions régulières* ou *moteur d'expressions rationnelles*. Il existe des standards permettant d'assurer une cohérence dans l'utilisation de ces outils.

Le standard POSIX propose trois jeux de normes :

- BRE (Basic Regular Expressions) pour les expressions régulières basiques. C'est par exemple le standard par défaut pour *sed* et *grep*
- ERE (Extended Regular Expressions) pour les expressions régulières étendues.
- SRE (Simple Regular Expressions) qui est devenu obsolète.

Les expressions régulières de perl sont également un standard de fait, en raison de leur richesse expressive et de leur puissance. Tout en suivant leur propre évolution, elles sont par exemple à l'origine de la bibliothèque PCRE. ECMAScript propose également dans le document Standard ECMA-262 une norme employée par exemple par JavaScript.

Les notations ou leurs sémantiques peuvent varier légèrement d'un moteur d'expression régulière à l'autre. Ils peuvent ne respecter que partiellement ces normes, ou de manière incomplète, ou proposer leurs propres fonctionnalités, comme GNU ou le Framework .NET. Les spécificités de chacun sont abordées plus loin dans cet article.

Classe de caractères

Une classe de caractères désigne un ensemble de caractères. Elle peut être définie de différentes manières :

- en extension (`[0123456789]` pour les caractères de « 0 » à « 9 ») ;
- en intension (`[0-9]` en conservant cet exemple) ;
- négativement : les classes `[^0123456789]` et `[^0-9]` désignent chacune l'ensemble des caractères qui ne sont pas des chiffres décimaux.

Des unions de classes de caractères peuvent être faites : `[0-9ab]` désigne l'ensemble constitué des caractères « 0 » à « 9 » et des lettres « a » et « b ». Certaines bibliothèques permettent également de faire des intersections de classes de caractères.

Entre les crochets, les métacaractères sont interprétés de manière littérale : `[.?*]` désigne l'ensemble constitué des caractères « . », « ? » et « * ».

Standardisation et application

Les classes de caractères les plus utilisées sont généralement fournies avec le moteur d'expression régulière. Un inventaire de ces classes est dressé dans la table ci-dessous.

La bibliothèque POSIX définit des classes au départ pour l'ASCII, puis, par extensions, pour d'autres formes de codages de caractères, en fonction des paramètres régionaux.

Dans Unicode et des langages comme le perl, des ensembles de caractères sont définis au travers de la notion de propriétés de caractères. Cela permet de désigner un ensemble de caractères en fonction de sa catégorie (exemples : lettre, ponctuation ouvrante, ponctuation fermante, séparateur, caractère de contrôle), en fonction du sens d'écriture (par exemple de gauche à droite ou de droite à gauche), en fonction de l'alphabet (exemples : latin, cyrillique, grec, Hiragana) ; en fonction de l'allocation des blocs, ou même selon les mêmes principes que les classes de caractères POSIX[4] (à ce sujet, lire la section Expressions régulières et Unicode).

POSIX	Non-standard	perl, Python	Vim	Java	Unicode[5, 6]	ASCII	Description
				\p{ASCII}		[\x00-\x7F]	Caractères ASCII
[:alnum:]				\p{Alnum}		A-Za-z0-9	Caractères alphanumériques
	[:word:]	\w	\w	\w		A-Za-z0-9_	Caractères alphanumériques, et « _ »
		\W	\W	\W		^A-Za-z0-9_	Caractères ne composant pas les mots
[:alpha:]			\a	\p{Alpha}	\p{L} ou \p{Letter}	A-Za-z	Caractères alphabétiques
[:blank:]			\s	\p{Blank}		\t	Espace et tabulation
		\b	\< \>	\b		(?<=\W)(?=\w)|(?<=\w)(?=\W)	Positions de début et fin de mots
		\B		\B		(?<=\W)(?=\W)|(?<=\w)(?=\w)	Positions ne correspondant pas à un début ou une fin de mot
[:cntrl:]				\p{Cntrl}	\p{Cc} ou \p{Control}	\x00-\x1F\x7F	Caractères de contrôle
[:digit:]		\d	\d	\p{Digit} ou \d	\p{Nd} ou \p{Decimal_Digit_Number}	0-9	Chiffres décimaux
		\D	\D	\D	\P{Nd} ou \P{Decimal_Digit_Number}	^0-9	Autre chose qu'un chiffre décimal
[:graph:]				\p{Graph}		\x21-\x7E	Caractères visibles
[:lower:]			\l	\p{Lower}	\p{Ll} ou \p{Lowercase_Letter}	a-z	Lettres en minuscule
[:print:]			\p	\p{Print}		\x20-\x7E	Caractères imprimables
[:punct:]				\p{Punct}	\p{P} ou \p{Punctuation}	] [!"#$%&'()*+,./:;<=>?@\^_`{|}~-	Caractères de ponctuation
[:space:]		\s	_s	\p{Space} ou \s	\p{Z} ou \p{Separator}	\t\r\n\v\f	Caractères d'espacement
		\S	\S	\S	\P{Z} ou \P{Separator}	^ \t\r\n\v\f	Autre chose qu'un caractère d'espacement
[:upper:]			\u	\p{Upper}	\p{Lu} ou \p{Uppercase_Letter}	A-Z	Lettres capitales
[:xdigit:]			\x	\p{XDigit}		A-Fa-f0-9	Chiffres hexadécimaux
		\A					Début de chaîne de caractère
		\Z					Fin de chaîne de caractère

Par exemple, dans le standard POSIX, `[[:upper:]ab]` fait correspondre un caractère parmi l'ensemble formé par les lettres capitales et les lettres minuscules « a » et « b ». Dans le standard ASCII, cette expression régulière s'écrirait `[A-Zab]`.

Classe d'équivalence

La notion de classe d'équivalence ne doit pas être confondue avec la notion de classe de caractères.

Par exemple, dans la locale FR, la classe [=e=] regroupe l'ensemble des lettres {e, é, è, ë, ê}.

Ceci signifie que lorsqu'elles sont collationnées, les lettres {e, é, è, ë, ê} apparaissent dans le même jeu de caractères, après le *d*, et avant le *f*.

Fonctions avancées

La plupart des standards et moteurs d'expressions régulières proposent des fonctions avancées. Notamment :

- **Quantificateurs non gloutons** : Par défaut, les quantificateurs « + » et « * » recherchent la plus grande séquence correspondant au motif recherché. Ces nouveaux quantificateurs, souvent notés « +? » et « *? », permettent à l'inverse de rechercher la plus petite séquence correspondante. Par exemple, l'expression régulière `ab+?` appliquée à la chaîne « abbbbc » entre en correspondance avec la sous-chaîne « ab » plutôt que « abbbb ».
- **Capture des groupements** : La capture des groupements permet de réutiliser un groupement entré en correspondance pour un traitement ultérieur, par exemple une substitution. Dans la plupart des syntaxes, il est possible d'accéder au $n^{ème}$ groupement capturé par la syntaxe « `\n` » ou parfois « `$n` », où *n* est un entier.
- **Groupements non capturants** : Lorsqu'elle est implémentée, la capture des groupements est souvent le comportement par défaut. Comme elle a un coût algorithmique important, il est parfois possible de désactiver la capture de certains groupements. On peut par exemple citer la syntaxe « `(?:groupement)` ».
- **Captures nommées** : Les longues expressions régulières avec de nombreux groupements peuvent être complexes à lire et à maintenir. Pour faciliter cette tâche, certains moteurs permettent de nommer les groupements, par exemple avec la syntaxe « `(?P<nom>groupement)` » en Python.
- **Références arrières** : Les références arrières permettent de faire référence à un même groupement capturé. Par exemple « `b(.)b\1` » entrera en correspondance avec « bébé » et « bobo » mais pas « baby ». Cette fonctionnalité, proposée par la plupart des moteurs, permet de reconnaître des langages non rationnels tels que $a^n b a^n$ pour tout *n* entier positif.
- **Modificateurs de mode de correspondance** : ces modificateurs permettent de faire varier localement le comportement de l'expression régulière. Par exemple, alors qu'elles sont normalement sensibles à la casse, l'expression perl « `(?i:nom)-Prénom` » entrera en correspondance avec « NOM-Prénom » et « Nom-Prénom » mais pas « Nom-prénom ». Parmi les autres modificateurs, on peut citer le mode multi-lignes, le mode non-gourmand ou le « free-spacing mode ».
- **Conditions** : certains moteurs permettent de construire des structures « `if ... then ... else ...` » au sein-même des expressions régulières. Par exemple, en perl, l'expression « `^(?(?=[abc])[def]|[ghi])` » se lit : « si la chaîne commence par la lettre a, b ou c, chercher à leur suite la lettre d, e ou f, sinon chercher la lettre g, h ou i. »
- **Commentaires** : Dans un souci de lisibilité, des moteurs permettent de commenter les expressions régulières au sein-même de celles-ci.
- **Code embarqué** : Lorsqu'une expression régulière est utilisée au sein d'un programme cette fonctionnalité permet de déclencher des actions lorsqu'une partie de la chaîne est entrée en correspondance.

Notations : implémentations et standardisation

Les notations utilisées sont très variables. Ce chapitre regroupe d'une part les notations propres à différentes implémentations, et d'autre part, l'entreprise de normalisation.

Standard POSIX

Le standard POSIX a cherché à remédier à la prolifération des syntaxes et fonctionnalités, en offrant un standard d'expressions régulières configurables. On peut en obtenir un aperçu en lisant le manuel de `regex` sous une grande partie des dialectes Unix dont GNU/Linux. Toutefois, même cette norme n'inclut pas toutes les fonctionnalités ajoutées aux expressions régulières de Perl.

Enfin, POSIX ajoute le support pour des plates-formes utilisant un jeu de caractère non basé sur l'ASCII, notamment EBCDIC, et un support partiel des locales pour certains méta-caractères.

Expressions régulières basiques

Les utilitaires du monde Unix tels que sed, GNU grep, ed ou vi utilisent par défaut la norme BRE (« *Basic Regular Expression* ») de POSIX. Dans celle-ci, les accolades, les parenthèses, le symbole « ? » et le symbole « + » ne sont pas des métacaractères : ils ne représentent qu'eux même. Pour prendre leur sémantique de métacaractères, ils ont besoin d'être échappés par le symbole « \ ».

Exemple : l'expression régulière `(ab.)+` reconnaît « (abc)+ » mais pas « abcabd », pour laquelle `\(ab.\)\+` convient.

Expressions régulières étendues

Les expressions régulières étendues POSIX (ERE pour « *Extented Regular Expression* ») sont souvent supportées dans les utilitaires des distributions Unix et GNU/Linux en incluant le drapeau *-E* dans la ligne de commande d'invocation de ces utilitaires. Contrairement aux expressions régulières basiques, elles reconnaissent les caractères vus précédemment comme des métacaractères. Ils doivent ainsi être échappés pour être interprétés littéralement.

La plupart des exemples donnés en présentation sont des expressions régulières étendues POSIX.

Séquences d'échappement

Comme les caractères `(`, `)`, `[`, `]`, `.`, `*`, `?`, `+`, `^`, `|`, `$` , `-` et `\` sont utilisés comme symboles spéciaux, ils doivent être référencés dans une séquence d'échappement s'ils doivent désigner littéralement le caractère correspondant. Ceci se fait en les précédant avec une barre oblique inversée `\`.

Notation étendue dans vim et emacs

Des extensions semblables sont utilisées dans l'éditeur alternatif *emacs* qui utilise un jeu de commandes différent mais reprend les mêmes expressions régulières en apportant une notation étendue. Les **expressions régulières étendues** sont maintenant supportées aussi dans *vim*, la version améliorée de *vi*.

Opérateur étendu (non POSIX)	Description	Exemple
`\{m,n\}`	Dans la notation étendue, cela crée un quantificateur borné personnalisé, permettant de faire correspondre exactement de *m* à *n* occurrences de ce qui précède, *m* et *n* étant deux entiers tels que $m < n$. Chacun des deux paramètres peut être omis : si le premier paramètre *m* est omis, il prend la valeur par défaut 0 ; si le second paramètre *n* est omis, mais la virgule est présente, il est considéré comme infini ; si le second paramètre *n* est omis ainsi que la virgule séparatrice, il prend la valeur par défaut égale au premier paramètre *m*.	Voir exemples ci-dessous.
`\( \)`	Dans la notation étendue, les parenthèses de groupement (dans une séquence d'échappement) permettent de délimiter un ensemble d'alternatives, ou toute sous-expression régulière (à l'exception des conditions de début et fin de ligne) pour leur appliquer un quantificateur. De plus, ces parenthèses délimitent un groupe de capture numéroté qui peut être utilisé pour les substitutions (on référence alors les groupes capturés dans la chaîne de substitution avec `$`*n* où *n* est le numéro de groupe de capture entre 1 et 9, la totalité de la chaîne trouvée étant représentée par `$&`).	Voir exemples ci-dessous.

De plus, de nombreuses autres séquences d'échappement sont ajoutées pour désigner des classes de caractères prédéfinies. Elles sont spécifiques à chaque utilitaire ou parfois variables en fonction de la version ou la plate-forme (cependant elles sont stables depuis longtemps dans emacs qui a fait figure de précurseur de ces extensions, que d'autres auteurs ont partiellement implémentées de façon limitée ou différente).

Python

Python utilise des expressions régulières basées sur les expressions régulières POSIX, avec quelques extensions ou différences.

Les éléments compatibles POSIX sont les suivants :

- opérateurs `[ ]`, `.`, `*`, `?`, `+`, `|`, `( )`
- caractères `\t`, `\n`, `\v`, `\f`, `\r`
- `\`*ooo* : caractère littéral dont le code octal (entre 0 et 377, sur 1 à 3 chiffres) est *ooo*.
- `\x`*NN* : caractère littéral dont le code hexadécimal est *NN* (sur 2 chiffres).

La séquence `\b` désigne le caractère de retour arrière (`0x08` avec un codage compatible ASCII) lorsqu'elle est utilisée à l'intérieur d'une classe de caractère, et la limite d'un mot autrement.

Bibliothèque BSD

Le système d'exploitation BSD utilise la bibliothèque *regex* écrite par Henry Spencer `(en)`. Compatible avec la norme POSIX 1003.2, cette bibliothèque est également utilisée par MySQL [7] (avec les opérateurs REGEXP et NOT REGEXP[8]) et PostgreSQL[9] (avec l'opérateur « ~ » et ses variantes).

Tcl

Le moteur d'expressions régulières du langage Tcl est issu de développements d'Henry Spencer postérieurs à ceux de la bibliothèque BSD[10,11]. Les expressions régulières sont appelées Expressions régulières avancées (ou ARE, Advanced Regular Expressions) et sont légèrement différentes des expressions régulières étendues de POSIX[11]. Les expressions régulières basiques et étendues sont également supportées.

Contrairement à d'autres moteurs, celui-ci fonctionne à base d'automates, ce qui le rend moins performant lorsque les captures ou le *backtracking* sont nécessaires, mais plus performant dans le cas contraire. [réf. souhaitée]

Perl

Perl offre un ensemble d'extensions particulièrement riche. Ce langage de programmation connaît un succès très important dû à la présence d'opérateurs d'expressions régulières inclus dans le langage lui-même. Les extensions qu'il propose sont également disponibles pour d'autres programmes sous le nom de *lib PCRE* (*Perl-Compatible Regular Expressions*, littéralement *bibliothèque d'expressions régulières compatible avec Perl*). Cette bibliothèque a été écrite initialement pour le serveur de courrier électronique Exim, mais est maintenant reprise par d'autres projets comme Python, Apache, Postfix, KDE, Analog, PHP et Ferite.

Les spécifications de Perl 6 régularisent et étendent le mécanisme du système d'expressions régulières. De plus il est mieux intégré au langage que dans Perl 5. Le contrôle du retour sur trace y est très fin. Le système de regex de Perl 6 est assez puissant pour écrire des analyseurs syntaxiques sans l'aide de modules externes d'analyse. Les expressions régulières y sont une forme de sous-routines et les grammaires une forme de classe. Le mécanisme est mis en œuvre en assembleur Parrot par le module PGE dans la mise en œuvre Parrot de Perl 6 et en Haskell dans la mise en œuvre Pugs. Ces mises en œuvre sont une étape importante pour la réalisation d'un compilateur Perl 6 complet. Certaines des fonctionnalités des regexp de Perl 6, comme les captures nommées, sont intégrées depuis Perl 5.10[12].

PHP

PHP supporte deux formes de notations : la syntaxe POSIX[13] (POSIX 1003.2) et celle, beaucoup plus riche et performante[14], de la bibliothèque PCRE[15] (Perl Compatible Regular Expression).

Un des défauts reprochés à PHP est lié à son support limité des chaînes de caractères, alors même qu'il est principalement utilisé pour traiter du texte, puisque le texte ne peut y être représenté que dans un jeu de caractères codés sur 8 bits, sans pouvoir préciser clairement quel codage est utilisé. En pratique, il faut donc adjoindre à PHP des bibliothèques de support pour le codage et le décodage des textes, ne serait-ce que pour les représenter en UTF-8. Toutefois, même en UTF-8, le problème se pose immédiatement avec la sémantique des expressions régulières puisque les caractères ont alors un codage de longueur variable, qui nécessite de complexifier les expressions régulières. [réf. nécessaire] Des extensions optionnelles de PHP sont donc développées pour créer un nouveau type de donnée pour le texte, afin de faciliter son traitement (et être à terme compatible avec Perl6 qui, comme Haskell, disposera nativement du support intégral d'Unicode). [réf. nécessaire]

ICU

ICU définit une bibliothèque portable pour le traitement de textes internationaux. Celle-ci est développée d'abord en langage C (version nommée ICU4C) ou pour la plate-forme Java (version nommée ICU4J). Des portages (ou adaptations) sont aussi disponibles dans de nombreux autres langages, en utilisant la bibliothèque développée pour le langage C (ou C++).

Les expressions régulières utilisables dans ICU reprennent les caractéristiques des expressions régulières de Perl, mais les complètent pour leur apporter le support intégral du jeu de caractères Unicode (voir la section suivante pour les questions relatives à la normalisation toujours en cours). Elles clarifient également leur signification en rendant les expressions régulières indépendantes du jeu de caractère codé utilisé dans les documents, puisque le jeu de caractères Unicode est utilisé comme codage pivot interne.

En effet, les expressions régulières de Perl (ou PCRE) ne sont pas portables pour traiter des documents utilisant des jeux de caractères codés différents, et ne supportent pas non plus correctement les jeux de caractères codés multi-octets (à longueur variable tels que ISO 2022, Shift-JIS, ou UTF-8), ou codés sur une ou plusieurs unités binaires de plus de 8 bits (par exemple UTF-16) puisque le codage effectif de ces jeux sous forme de séquences d'octets dépend de la plate-forme utilisée pour le traitement (ordre de stockage des octets dans un mot de plus de 8 bits).

ICU résout cela en adoptant un traitement utilisant en interne un jeu unique défini sur 32 bits et supportant la totalité du jeu de caractères universel (UCS), tel qu'il est défini dans la norme ISO/IEC 10646 et précisé sémantiquement dans le standard Unicode (qui ajoute à la norme le support de propriétés informatives ou normatives sur les caractères, et des recommandations pour le traitement automatique du texte, certaines de ces recommandations étant optionnelles ou informatives, d'autres étant devenues standards et intégrées au standard Unicode lui-même, d'autres enfin ayant acquis le statut de norme internationale à l'ISO ou de norme nationale dans certains pays).

ICU supporte les extensions suivantes[16], directement dans les expressions régulières, ou dans l'expression régulière d'une classe de caractères (entre `[…]`) :

- `\u`*`hhhh`* : correspond à un caractère dont le point de code (selon ISO/IEC 10646 et Unicode) a la valeur hexadécimale *hhhh*.
- `\U`*`hhhhhhhh`* : correspond à un caractère dont le point de code (selon ISO/IEC 10646 et Unicode) a la valeur hexadécimale *hhhhhhhh* ; exactement huit chiffres hexadécimaux doivent être fournis, même si le point de code le plus grand accepté est `\U0010ffff`.
- `\N{`*`NOM DU CARACTÈRE UNICODE`*`}` : correspond au caractère désigné par son nom normalisé, c'est-à-dire tel qu'il est défini de façon irrévocable dans la norme ISO/IEC 10646 (et repris dans le standard Unicode). Cette syntaxe est une version simplifiée de la syntaxe suivante permettant de désigner d'autres propriétés sur les caractères :

- `\p{`*code d'une propriété Unicode*`}` : correspond à un caractère doté de la propriété Unicode spécifiée.
- `\P{`*code d'une propriété Unicode*`}` : correspond à un caractère non doté de la propriété Unicode spécifiée.
- `\s` : correspond à un caractère séparateur ; un séparateur est défini comme `[\t\n\f\r\p{Z}]`.

Les expressions régulières d'ICU sont actuellement parmi les plus puissantes et les plus expressives dans le traitement des documents multilingues. Elles sont largement à la base de la normalisation (toujours en cours) des expressions régulières Unicode (voir ci-dessous) et un sous-ensemble est supporté nativement dans la bibliothèque standard du langage Java (qui utilise en interne un jeu de caractères portable à codage variable, basé sur UTF-16 avec des extensions, et dont les unités de codage sont sur 16 bits).

ICU est une bibliothèque encore en évolution. En principe, elle devrait adopter toutes les extensions annoncées dans Perl (notamment les captures nommées), dans le but d'assurer l'interopérabilité avec Perl 5, Perl 6, et PCRE, et les autres langages de plus en plus nombreux qui utilisent cette syntaxe étendue, et les auteurs d'ICU et de Perl travaillent en concert pour définir une notation commune. Toutefois, ICU adopte en priorité les extensions adoptées dans les expressions régulières décrites dans le standard Unicode, puisque ICU sert de référence principale dans cette annexe standard d'Unicode.

Toutefois, il n'existe encore aucun standard ou norme technique pour traiter certains aspects importants des expressions régulières dans un contexte multilingue, notamment :

- La prise en compte de l'ordonnancement propre à chaque locale (c'est-à-dire l'ordre de tri, éventuellement multiniveau, des textes en fonction de la langue et de l'écriture, et des unités inséparables de texte qui peuvent comprendre un ou plusieurs caractères codés éventuellement de plusieurs façons possibles mais toutes équivalentes dans cette langue) ; ce manque de normalisation (expérimentation toujours en cours) fait apparaître des différences de traitement et donc de portabilité des classes de caractères contenant des étendues (de la forme `[`*a*`-`*b*`]`). Actuellement, ICU ne supporte encore que les étendues dans l'ordre binaire des points de code Unicode, un ordre qui n'est pas du tout approprié pour le traitement correct de nombreuses langues puisqu'il contrevient à leur ordre de collation standard.
- L'ordre des occurrences multiples trouvées dans le texte quand celles-ci peuvent se chevaucher totalement ou partiellement. Cela résulte de l'utilisation de quantificateurs variables, et influe sur le contenu des sous-chaînes capturées. Si cet ordre peut changer, et que seule la première occurrence trouvée est utilisée par exemple dans une opération de recherche et substitution automatique ou le traitement des captures, alors le traitement dépendra de l'implémentation. En théorie, les expressions régulières désignent chacune un ensemble **non ordonné** de textes, et les accords trouvés dans un texte source peuvent être eux aussi à des positions quelconque dans le texte source, puisque le résultat d'une recherche contient en fait non seulement les captures, mais aussi les positions relatives.

Pour préciser ces derniers aspects manquants, des métacaractères supplémentaires devraient pouvoir être utilisés pour contrôler ou filtrer les occurrences trouvées, ou bien un ordre normalisé imposé à la liste des occurrences retournées. Les auteurs d'applications doivent donc être vigilants sur ces points et s'assurer de lire toutes les occurrences trouvées et pas seulement la première, afin de pouvoir décider laquelle des occurrences est la mieux appropriée à une opération donnée.

Expressions régulières et Unicode

Les expressions régulières ont originellement été utilisées avec les caractères ASCII. Beaucoup de moteurs d'expressions régulières peuvent maintenant gérer l'Unicode. Sur plusieurs points, le jeu de caractères codés utilisés ne fait aucune différence, mais certains problèmes surgissent dans l'extension des expressions régulières pour Unicode.

Une question est de savoir quel format de représentation interne d'Unicode est supporté. Tous les moteurs d'expressions régulières en ligne de commande attendent de l'UTF-8, mais pour les bibliothèques, certaines attendent aussi de l'UTF-8, mais d'autres attendent un jeu codé sur UCS-2 uniquement (voire son extension UTF-16 qui restreint aussi les séquences valides), ou sur UCS-4 uniquement (voire sa restriction normalisée UTF-32).

Une deuxième question est de savoir si l'intégralité de la plage des valeurs d'une version d'Unicode est supportée. Beaucoup de moteurs ne supportent que le Basic Multilingual Plane, c'est-à-dire, les caractères encodables sur 16 bits. Seuls quelques moteurs peuvent (dès 2006) gérer les plages de valeurs Unicode sur 21 bits.

Une troisième question est de savoir comment les constructions ASCII sont étendues à l'Unicode.

- Par exemple, dans les mises en œuvre ASCII, les plages de valeurs de la forme `[`*x*`-`*y*`]` sont valides quels que soient *x* et *y* dans la plage 0x0..0x7F et `codepoint(`*x*`) ≤ codepoint(`*y*`)`.
- L'extension naturelle de ces plages de valeurs Unicode changerait simplement l'exigence sur la plage de valeurs [0..0x7F] en exigence sur la plage étendue à 0..0x1FFFFF.

Cependant, en pratique ce n'est souvent pas le cas :

- Certaines mises en œuvre, telles que celle de gawk, ne permettent pas aux plages de valeurs de couvrir plusieurs blocs Unicode. Une plage de valeurs telle que [0x61..0x7F] est valide puisque les deux bornes tombent à l'intérieur du même bloc *Basic Latin*, comme 0x530..0x560 qui tombe dans le bloc arménien, mais une plage telle que [0x61..0x532] est invalide puisqu'elle est à cheval sur plusieurs blocs Unicode. Cette restriction s'avère très gênante car de nombreuses langues nécessitent des caractères appartenant à des blocs différents, la définition même des blocs étant arbitraire et provenant seulement du processus historique d'allocation et d'évolution de la norme ISO/IEC 10646 (et en conséquence aussi, des évolutions du standard Unicode). Une telle restriction devrait être à terme levée par une amélioration de l'implémentation.

- D'autres moteurs tels que celui de l'éditeur Vim, permettent le chevauchement de blocs mais limitent le nombre de caractères d'une plage à 128, ce qui est encore plus pénalisant car cela ne permet pas de traiter directement certaines écritures, où il faudrait lister un nombre considérable de sous-plages, avec des conséquences importantes sur les performances. Là aussi, des améliorations sont attendues dans l'implémentation pour lever ces restrictions.

Un autre domaine dans lequel des variations existent est l'interprétation des indicateurs d'insensibilité à la casse.

- De tels indicateurs n'affectent que les caractères ASCII ; d'autres affectent tous les caractères (et prennent en compte la correspondance de casse soit caractère par caractère dans les implémentations les plus simples, soit au niveau du texte entier dans les implémentations respectant les contraintes de langues, ceci pouvant utiliser les correspondances d'un sous-ensemble d'une version donnée d'Unicode, d'autres pouvant utiliser toutes les correspondances de n'importe quelle version d'Unicode, éventuellement précisable au sein même de l'expression régulière).
- Certains moteurs ont deux indicateurs différents, l'un pour ASCII, l'autre pour Unicode.
- Les caractères exacts qui appartiennent aux classes POSIX varient également.

Une autre réponse à Unicode a été l'introduction des classes de caractères pour les blocs Unicode et les propriétés générales des caractères Unicode:

- En Perl et dans la bibliothèque `java.util.regex` de Java, les classes de la forme `\p{`*`InX`*`}` valident les caractères du bloc *X* et `\P{`*`InX`*`}` valide le complément. Par exemple, `\p{Arabic}` valide n'importe quel caractère de l'écriture arabe (dans l'un quelconque des blocs normalisés d'Unicode/ISO/IEC 10646 où de tels caractères sont présents).
- Similairement, `\p{`*`X`*`}` valide n'importe quel caractère ayant la propriété de catégorie générale de caractère *X* et `\P{`*`X`*`}` le complément. Par exemple, `\p{Lu}` valide n'importe quelle lettre capitale (*upper-case letter*).
- D'autres propriétés que la catégorie générale peuvent être désignées avec la notation `\p{`*`prop=valeur`*`}` et son complément `\P{`*`prop=valeur`*`}`, où *prop* est le code d'une propriété de caractères, et *valeur* sa valeur attribuée à chaque caractère.
- Enfin des extensions ont été proposées pour utiliser un autre opérateur que la seule égalité (ou différence) de valeur de propriété, en utilisant une syntaxe d'expression régulière ou une simple alternation pour la *valeur*.

Notes :

- Certaines propriétés de caractères sont normatives et ne devraient pas dépendre de la version utilisée, c'est le cas des propriétés définies dans ISO/IEC 10646 : le nom normalisé du caractère, le point de code, le nom du bloc où le caractère est codé.
- D'autres propriétés sont standards et correspondent à des propriétés normatives du standard Unicode : ce sont essentiellement les propriétés de base définies dans la table principale de caractères Unicode. En principe, elles sont invariables. C'est le cas des correspondances simples de casse (caractère par caractère), ou de la catégorie générale du caractère. Dans bien des cas, ces propriétés ne sont pas adaptées à toutes les langues.
- D'autres propriétés sont informatives, et peuvent faire l'objet de révision d'une version d'Unicode à l'autre : ce sont essentiellement les propriétés définies dans les tables supplémentaires d'Unicode. En particulier elles sont adaptables en fonction de la langue utilisée et désignent par exemple les propriétés de correspondances de casse étendues (traitant le texte dans sa globalité), ou les propriétés d'ordonnancement (*collation*).
- Cependant certaines de ces dernières tables ont acquis le statut de standard (en étant intégrées dans des annexes standards du standard Unicode) et sont même utilisées dans la définition de certaines normes, ou bien d'autres propriétés historiques sont encore maintenues mais d'usage non recommandé. Consulter le standard Unicode pour connaître le statut actuel de ces propriétés.

Implémentations et complexité algorithmique

Il existe au moins trois familles d'algorithmes qui déterminent si une chaîne de caractères correspond à une expression régulière.

La plus ancienne approche, dite explicite, repose sur la traduction de l'expression régulière en un automate fini déterministe (AFD). La construction d'un tel automate pour une expression régulière de taille m a une complexité en taille et en mémoire en $O(2^m)$ mais peut être exécutée sur une chaîne de taille n en un temps $O(n)$.

Une approche alternative, dite implicite, consiste à simuler un automate fini non déterministe en construisant chaque AFD à la volée et en s'en débarrassant à l'étape suivante. Cette approche évite la complexité exponentielle de l'approche précédente, mais le temps d'exécution augmente en $O(mn)$. Ces algorithmes sont rapides mais certaines fonctionnalités telles que la recapture de sous-chaînes et la quantification non gourmande sont difficiles à mettre en oeuvre[17].

La troisième approche consiste à confronter le motif à la chaîne de caractères par séparation et évaluation (« *backtracking* »). Sa complexité algorithmique est exponentielle dans le pire des cas, par exemple avec des motifs tels que `(a|aa)*b`, mais donne de bons résultats en pratique. Elle est plus flexible et autorise un plus grand pouvoir expressif, par exemple en simplifiant la recapture de sous-chaînes.

Certaines implémentations tentent de combiner les qualités des différentes approches, en commençant la recherche avec un AFD, puis en utilisant le backtracking lorsque c'est nécessaire.

Notes et références

Notes

1. terme promulgué par la norme ISO/IEC 9945 [réf. nécessaire]
2. P.46, Kleene introduit la notion d'*événement régulier* (regular event), pour ce qui a été appelé plus tard *ensemble régulier* ou *langage régulier* et demande qu'on lui suggère un terme plus descriptif ! La définition des événements réguliers est p. 48.

Références

1. Théorie des langages Notes de Cours *François Yvon et Akim Demaille avec la participation de Pierre Senellart* (https://www.lrde.epita.fr/~akim/thl/lecture-notes/theorie-des-langages-1.pdf)
2. S. C. Kleene, « Representation of events in nerve nets and finite automata. », dans C.E. Shannon and J. McCarthy, *Automata Studies*, vol. 34, Princeton, N. J., Princeton University Press, coll. « Annals of Mathematics Studies », 1956 (lire en ligne (http://www.dtic.mil/get-tr-doc/pdf?AD=ADA596138)), p. 3 -- 41.
3. Dans leur article fondateur *Finite Automata and Their Decision Problems*, Rabin, M. O. ; Scott, D. (1959). IBM Journal of Research and Development, les auteurs, dans leur définition 3, appellent *ensembles définissables de bandes* (definissable sets of tapes) ce qui a été appelé plus tard *langages réguliers* ou *langages rationnels*, suivant les terminologies adoptées.
4. Voir perlunicode (http://perldoc.perl.org/perlunicode.html), Perl Programming Documentation **(en)**
5. **(en)** Unicode characters and properties (https://www.regular-expressions.info/unicode.html), Regex tutorial
6. **(en)** Unicode Character Categories (http://www.fileformat.info/info/unicode/category/index.htm)
7. Source : Regular Expressions (http://dev.mysql.com/doc/refman/5.6/en/regexp.html), MySQL 5.6 Reference Manual **(en)**
8. MySQL AB :: MySQL 5.0 Reference Manual :: F expressions régulières MySQL (http://dev.mysql.com/doc/refman/5.0/fr/regexp.html).
9. Source : Regular Expression Details (http://www.postgresql.org/docs/9.3/static/functions-matching.html#POSIX-SYNTAX-DETAILS), PostgreSQL 8.4+ Reference Manual **(en)**
10. Source : regex - Henry Spencer's regular expression libraries (https://garyhouston.github.io/regex/) **(en)**
11. Source : Regular expressions (http://wiki.tcl.tk/396), Wiki Tcl **(en)**
12. http://perldoc.perl.org/perl5100delta.html#Regular-expressions.
13. PHP: Regex POSIX - Manual (http://php.net/manual/fr/ref.regex.php).
14. **(en)** Computerworld - PHP Developer's Guide | Pattern matching with PHP (http://www.computerworld.com.au/index.php/id;347504672;pp;4;fp;2;fpid;76768).
15. PHP: PCRE - Manual (http://php.net/manual/fr/ref.pcre.php).
16. D'après : http://www.icu-project.org/userguide/regexp.html.
17. **(en)** Regular Expression Matching Can Be Simple and Fast (https://swtch.com/~rsc/regexp/regexp1.html), Cox, Russ (2007).

Voir aussi

Articles connexes

- Filtrage par motif
- Algèbre de Kleene

Bibliographie

- S. C. Kleene, « Representation of events in nerve nets and finite automata. », dans C.E. Shannon and J. McCarthy, *Automata Studies*, vol. 34, Princeton, N. J., Princeton University Press, coll. « Annals of Mathematics Studies », 1956 (lire en ligne (http://www.dtic.mil/get-tr-doc/pdf?AD=ADA596138)), p. 3 -- 41
- MO Rabin, D Scott, « Finite automata and their decision problems », *IBM journal of research and development*, 1959 (lire en ligne (https://www.researchgate.net/profile/Dana_Scott3/publication/230876408_Finite_Automata_and_Their_Decision_Problems/links/582783f808ae950ace6cd752/Finite-Automata-and-Their-Decision-Problems.pdf))
- Bernard Desgraupes, *Introduction aux expressions régulières*, Paris, Vuibert, septembre 2001, 248 p., broché (ISBN 2-7117-8680-3).
- Jeffrey E. F. Friedl (trad. Laurent Dami), *Maîtrise des expressions régulières* [« Mastering regular expressions »], Paris, O'Reilly, avril 2001 (réimpr. 2003), 364 p., broché (ISBN 2-84177-126-1).
- Martial Bornet, *Expressions Régulières, Syntaxe et mise en œuvre*, Paris, ENI, septembre 2015, 465 p., broché (ISBN 978-2-7460-9806-0).

Sur les autres projets Wikimedia :

Expression régulière (https://commons.wikimedia.org/wiki/Category:Regex?uselang=fr), sur Wikimedia Commons

regex, sur le Wiktionnaire

Expression régulière, sur Wikiversity

Expression régulière, sur Wikibooks