Technique & Pratique
Collection dirigée par Emmanuel Cornet et Alexandre Hérault

Les expressions régulières

par l'exemple

Vincent Fourmond
Ancien élève de l'École Normale Supérieure
Quinze années d'expérience dans dix langages de programmation

Avant-propos

Quiconque ayant déjà écrit un code « à la main » pour extraire des numéros de téléphone, des dates ou même des nombres d'une chaîne de caractères sait que ce n'est ni facile ni intéressant. En conséquence, les programmeurs ont tendance à éviter dans la mesure du possible de tirer des données d'une chaîne de caractères, et quand ils n'ont pas le choix, ils écrivent un code réduit à sa plus simple expression. Cela génère des programmes rigides et des utilisateurs frustrés.

Pourquoi réécrire des routines qui ont déjà été écrites mille fois, alors que, en utilisant des expressions régulières, on peut obtenir toutes les informations dont on a besoin en quelques lignes de code ?

Dans cet ouvrage, nous vous proposons une prise en main rapide et efficace des expressions régulières. Nous vous montrerons notamment, à travers de nombreux exemples, comment :

- construire une expression régulière ;
- effectuer des remplacements ;
- tirer parti de la souplesse des expressions régulières dans un programme.

Les expressions régulières sont des outils indispensables dès que l'on travaille sur des chaînes de caractères contenant des informations. Au moyen d'une notation compacte et efficace, elles permettent de détecter facilement des schémas dans une chaîne et d'en extraire ce dont on a besoin. Munis de ce livre, vous pourrez rapidement vous approprier leur puissance, et oublier les vieux codes longs et difficiles à modifier pour traiter les chaînes.

Ce livre procède en deux mouvements. Dans la première partie (chapitres 1 à 3), vous apprendrez progressivement à construire des expressions régulières, au travers de nombreux exemples et de 44 exercices, tous corrigés en détail (voir page 111). Dans la seconde (chapitres 4 et 5), vous découvrirez comment exploiter les expressions régulières dans un programme. Nous vous invitons à vous référer à l'index (page 124) tout au long de votre lecture.

Nous espérons que vous aurez autant de plaisir à lire cet ouvrage que nous en avons eu à l'écrire ; nous savons qu'il vous sera utile en pratique. Vos critiques comme vos éloges nous aideront à l'améliorer encore : vous pouvez en faire part à l'éditeur, à l'adresse

`contact@H-K.fr`

Si vous rencontrez ce que vous estimez être une erreur ou une imprécision gênante dans l'ouvrage, nous vous serions reconnaissant de nous en faire part également.

Bonne lecture et bon amusement !

Vincent Fourmond

Table des matières

Annexes

Chapitre 1

Qu'est-ce qu'une expression régulière ?

Les ordinateurs n'ont pas du tout la même conception des textes que nous : pour nous, un texte est un ensemble d'idées couchées sur papier. Nous nous en servons pour transmettre des sentiments, des émotions ou, plus prosaïquement, des informations. Un ordinateur a une vision beaucoup plus pragmatique : un texte, c'est une suite de lettres ou plus exactement une suite de nombres qui représentent chacun une lettre. Il ne connaît ni langage ni signification.

Les ordinateurs ne savent pas naturellement repérer des informations dans un texte : il est souvent laborieux et rarement intéressant d'extraire des données d'une chaîne de caractères « à la main ». De plus, dès que l'on change un tant soit peu le type de données auxquelles on s'intéresse (des dates, des numéros de téléphone, des comptes bancaires, etc.), il faut récrire un nouveau code, à grand renforts de « copier/coller », de fautes de frappe et d'oublis.

À quoi bon réinventer la roue ? Les *expressions régulières* fournissent un moyen simple et particulièrement élégant de repérer des schémas dans une chaîne de caractères, en extraire des données et y apporter des modifications.

Quelques conventions

Avant de passer à un premier exemple, détaillons quelques conventions que nous avons adoptées dans ce livre.

Un texte entre `"guillemets américains"` représente une chaîne de caractères : c'est une entité qu'en tant que programmeur vous ne taperez quasiment jamais. De telles chaînes

sont le substrat des expressions régulières et seront, pour la plupart, connues uniquement au moment de l'exécution du programme.

En revanche, les textes entre « `guillemets français` » représentent des éléments que vous serez amenés à utiliser directement dans vos programmes.

Beaucoup de langages incorporent des fonctionnalités plus ou moins évoluées d'expressions régulières, parfois grâce à des bibliothèques externes ; chaque langage a sa manière de les écrire et, surtout, sa manière de les utiliser. Nous avons choisi de nous appuyer sur la syntaxe de Perl, car c'est le langage par excellence des expressions régulières : c'est là qu'elles sont les plus puissantes et aussi les plus faciles à utiliser. Nous verrons de plus que la syntaxe varie très peu d'un langage à un autre ; la majorité des exemples que nous proposons est directement utilisable dans les autres langages. Dans de nombreux langages on définit d'ailleurs les expressions régulières par rapport à celles de Perl. Ce choix n'aura donc presque pas d'influence sur la conception des expressions régulières.

Nous prendrons l'habitude de noter les expressions régulières entre barres obliques, par exemple `/expression/`, pour bien les démarquer du reste du texte.

Enfin, pour lever toute ambiguïté, nous matérialiserons les espaces par le symbole ␣, aussi bien dans les expressions régulières que dans les chaînes de caractères.

Exemple : est-ce une adresse électronique ?

Entrons dès maintenant dans le vif du sujet au moyen d'un exemple complet de construction d'une expression régulière. Nous introduirons très brièvement les symboles les plus courants, mais, ne vous inquiétez pas, nous reviendrons en détail sur chacun d'eux, de façon progressive, à partir du chapitre 2.

Certains sites Internet proposent des formulaires en ligne, à remplir directement dans une page web ; il est fréquent que

des utilisateurs y saisissent des données incomplètes ou inexactes suite à une erreur de manipulation ou à une faute de frappe. Pour alléger la charge des serveurs, il est donc important de savoir filtrer les informations qui y parviennent.

Supposons que vous travaillez sur un formulaire en ligne dans lequel l'utilisateur doit entrer une adresse électronique. Vous souhaitez faire rapidement le tri entre les adresses invalides, comme `"a"`, `"a@b"`, etc., et les adresses correctes, comme `"contact@h-k.fr"`. Nous allons construire pas à pas une expression régulière qui permet de vérifier si une adresse électronique est valide. Notons dès maintenant que les expressions régulières ne peuvent pas faire la différence entre une adresse réelle, comme `"contact@h-k.fr"` et une adresse inexistante mais « syntaxiquement correcte », par exemple `"bidule@biniou.edu"`. Pour cela, il faut recourir à d'autres mécanismes, comme l'envoi d'un courrier de confirmation.

Pour commencer, posons-nous la question : « À quoi ressemble une adresse électronique ? » La réponse la plus élémentaire que l'on puisse donner est : « C'est une chaîne de caractères qui contient une arobase (`@`). » On peut représenter de telles chaînes au moyen de l'expression régulière

```
/@/
```

Elle se lit : « Détecter toutes les chaînes de caractères qui contiennent un arobase. » Si l'adresse fournie par l'utilisateur ne satisfait pas à ce critère, nous pouvons le lui signaler par un message d'erreur.

Le test précédent permet de repérer les erreurs les plus grossières ; imposons maintenant qu'il y ait quelque chose avant et après l'arobase. « `.` » est un *joker* : il peut représenter n'importe quel caractère. « `+` » s'applique au caractère qui le précède et signifie « au moins une fois » (rappelons que nous reviendrons en détail sur tous ces symboles à partir du chapitre 2). On peut les assembler en « `.+` », soit « une suite quelconque d'au moins un caractère » ; nous nous en servi-

rons pour représenter « quelque chose ». Nous pouvons alors combiner cet élément avec l'arobase pour former

```
/.+@.+/
```

qui représente « une chaîne d'au moins un caractère (`.+`) suivie d'un arobase, puis d'une autre chaîne d'au moins un caractère (`.+`). » Nous avons progressé : cette dernière expression règle le sort de `"a@"` et `"@bcd"` car il leur manque l'une ou l'autre des chaînes autour de l'arobase. En revanche, elle laisse encore passer `"a@a"` qui n'est manifestement pas plus correcte que les deux précédentes.

Pour affiner notre tri, remarquons que le membre de droite d'une adresse électronique se compose d'au moins deux parties séparées par un point, comme dans `"h-k.fr"`. Nous savons déjà exprimer « n'importe quelle chaîne d'au moins un caractère », grâce à « `.+` ». La première idée pour vérifier que la partie de droite est conforme serait d'employer « `.+..+` ». Mais on sait déjà que « `.` » représente non pas un point, mais n'importe quel caractère. Il faut donc « protéger » le point pour conserver sa signification d'origine en l'associant au caractère antislash : « `\.` » correspond spécifiquement au caractère « . ». Ce que l'on est en train de construire peut donc se représenter au moyen de « `.+\..+` ». Nous pouvons alors modifier l'expression régulière précédente en

```
/.+@.+\..+/
```

pour tenir compte de la forme attendue pour le membre de droite. Cette expression se lit : « N'importe quelle chaîne non vide de caractères (`.+`) suivie d'une arobase (`@`), puis de n'importe quelle chaîne (`.+`), d'un point (`\.`) et enfin d'une troisième chaîne (`.+`). »

Continuons notre progression : en réalité, on ne peut pas utiliser n'importe quel caractère autour de l'arobase. En effet, les adresses électroniques ne peuvent comporter qu'une seule arobase ; cette dernière doit donc être proscrite dans les parties de gauche et de droite. Dans ce but, nous pouvons utiliser

en lieu et place des jokers (les points) la construction « `[^@]` » qui signifie « n'importe quel caractère *sauf* une arobase ». En effectuant les remplacements dans l'expression régulière précédente, nous obtenons :

```
/[^@]+@[^@]+\.[^@]+/
```

Elle se lit : « Une chaîne d'au moins un caractère ne comprenant pas d'arobase (« `[^@]+` ») suivie d'une arobase (« `@` »), puis d'une autre chaîne d'au moins un caractère sans arobase (« `[^@]+` »), d'un point (« `\.` ») et d'une troisième chaîne sans arobase (« `[^@]+` »). »

De même que l'expression `/@/` détecte les chaînes qui contiennent une arobase, toutes les expressions régulières que nous avons introduites jusqu'à présent permettent de vérifier que la chaîne fournie par l'utilisateur *contient* une adresse syntaxiquement valide ; s'il envoie `"a@contact@h-k.fr@b"`, l'expression se contente de détecter la partie du milieu (ici, `"contact@h-k.fr"`). Notre test laisse passer cette chaîne, ce qui n'est pas satisfaisant. Il faut s'assurer que la chaîne *entière* passe le test. On peut dans ce but utiliser « `^` » pour repérer le début d'une chaîne de caractères et « `$` » pour repérer sa fin. En les plaçant de part et d'autre de l'expression précédente, nous obtenons :

```
/^[^@]+@[^@]+\.[^@]+$/
```

Cette nouvelle expression permet de s'assurer que la chaîne entière correspond : le début et la fin de la chaîne à vérifier coïncident exactement avec ceux de notre expression. Elle se lit « Détecter une chaîne qui

- commence (`^`)
- par une chaîne sans arobase (`[^@]+`),
- suivie d'un arobase (`@`),
- puis d'une autre chaîne sans arobase (`[^@]+`),
- d'un point (`\.`),

– et d'une troisième chaîne sans arobase (`[^@]+`)
– allant jusqu'à la fin (`$`). »

Nous pourrions encore améliorer cette expression en prenant en compte d'autres facteurs, comme le nombre de caractères de la dernière chaîne, entre deux et quatre (`fr`, `com`, `info`, etc.), la présence éventuelle d'espaces de part et d'autre de l'adresse, de la liste exacte des caractères autorisés dans une adresse électronique, etc. Vous pourrez compléter cette expression après la lecture du chapitre 3.

Définition d'une expression régulière

La démarche précédente nous a montré ce qu'est une expression régulière : il s'agit d'un mini-programme, écrit dans un langage rudimentaire très compact, qui permet de décrire efficacement toute une série de chaînes de caractères qui se ressemblent. Elles servent à formaliser suffisamment ces ressemblances pour qu'un ordinateur puisse les reconnaître.

On parle souvent de *regexp*[1] ou *regex*, qui sont les abréviations de l'anglais *regular expression*, « expression régulière ». « Régulière » fait référence à la régularité[2] des chaînes que les expressions régulières permettent de reconnaître.

Quand utiliser des expressions régulières

Les expressions régulières apparaissent naturellement dès que l'on souhaite appliquer des traitements automatiques à du texte. On s'en sert pour :

– vérifier si celui-ci est syntaxiquement correct, comme dans le cas des adresses électroniques ;

1. Prononcer « règue-expe ».
2. La régularité est une notion mathématique de la théorie des langages. C'est de cette notion qu'il est question dans « expression régulière ».

– en extraire des informations, par exemple pour filtrer le résultat d'une commande particulièrement bavarde dont seule une partie nous intéresse ;
– effectuer des remplacements, par exemple pour « nettoyer » une chaîne issue de l'interaction avec l'utilisateur avant de la passer à une fonction qui exige qu'une syntaxe rigide soit respectée.

Nous verrons aussi qu'une bonne expression régulière ne peut pas remplacer un programme bien conçu !

À qui s'adresse ce livre ?

Ce livre s'adresse aux programmeurs qui ne connaissent pas ou peu les expressions régulières, et qui manquent d'outils pour traiter les entrées textuelles de leurs programmes.

Nous vous montrerons progressivement au cours des chapitres 2 et 3 comment exprimer des idées de plus en plus complexes en termes d'expressions régulières ; nous verrons ensuite au chapitre 4 comment les incorporer dans vos programmes et en tirer parti.

Le chapitre 4 s'appuie sur des codes en Perl. Que les lecteurs qui ne sont pas familiers avec ce langage ne s'en inquiètent pas : nous n'utiliserons pas les spécificités syntaxiques de Perl dans nos programmes. En effet, on peut écrire des codes en Perl dans une syntaxe « universelle », proche des autres langages de haut niveau. Ils seront compréhensibles par tous les programmeurs. Nous expliquerons en outre tout point de syntaxe inhabituel que nous ne pouvons pas contourner.

Les expressions régulières s'acquièrent par l'expérience ; ce n'est qu'en les manipulant que l'on comprend vraiment tout les bénéfices que l'on peut en tirer. C'est pourquoi nous nous appuierons sur maints exemples que vous pourrez réutiliser tels quels ou adapter à votre guise. Nous vous proposerons également de nombreux exercices dont la correction se trouve

dans l'annexe A (page 111). Ils peuvent se résoudre de tête, et nous vous invitons à y réfléchir avant de regarder leurs solutions. Nous vous invitons aussi à consulter systématiquement ces denières même si vous pensez avoir trouvé : un détail peut vous avoir échappé. De plus, nous y présenterons souvent des prolongements qui peuvent vous intéresser.

Enfin, le chapitre 5 expose dans le détail les manières d'utiliser les expressions régulières dans dix langages différents : Perl, Python, PHP, C#, Java, JavaScript, C, C++, Emacs Lisp et les scripts shell. Un exemple complet y est traité dans les dix langages ; vous pourrez adapter ces codes pour tester les nouveaux éléments au fur et à mesure de leur découverte. Ces programmes sont en téléchargement libre sur la page

`http://www.h-k.fr/liens/tp/regexps.html`

À la fin de ce livre, vous vous demanderez comment vous avez pu vous passer des expressions régulières jusqu'à présent !

Chapitre 2

Débuter avec les expressions régulières

Nous allons découvrir dans ce chapitre le principe de base des expressions régulières : la correspondance. Nous verrons ensuite plusieurs constituants élémentaires des expressions régulières qui servent à exprimer des idées relativement simples. Certains manques se feront ressentir ; ils seront presque tous comblés dans le chapitre suivant.

2.1 La notion de correspondance

Un *outil* est un « objet fabriqué, utilisé manuellement ou sur une machine pour réaliser une opération déterminée[1]. » En s'appuyant sur cette définition, il est aisé de vérifier qu'une *pioche* est un *outil* car c'est un objet fabriqué que l'on utilise manuellement pour réaliser l'opération « creuser un trou ». On peut dire que l'objet « pioche » *correspond* à la définition d'un « outil ».

Les expressions régulières sont aux chaînes de caractères ce que les mots génériques, comme « outil », sont à la langue française : elles désignent un ensemble plus ou moins grand de chaînes en donnant leur définition. Une chaîne de caractères fait partie de cet ensemble si elle vérifie point par point chacun des critères de la définition. Dans ce cas, on dit que la chaîne *correspond*[2] à l'expression régulière. Nous avons vu dans le premier chapitre que `"contact@h-k.fr"` correspond à `/.+@.+/` parce que `"contact"` correspond à « `.+` », `"@"` à « `@` » et `"h-k.fr"` à « `.+` ».

1. Larousse, dictionnaire encyclopédique en 4 volumes, édition 1998.
2. On utilise *match* en anglais.

Chapitre 3

Enrichir ses expressions régulières

Nous verrons dans ce chapitre des éléments pour mieux cibler les correspondances de nos expressions régulières. Ainsi, à l'issue de ce chapitre, vous saurez tout sur la conception des expressions régulières, à l'exception de quelques constructions spécifiques que nous verrons dans le chapitre suivant.

3.1 Remplacements

Nous nous sommes limités dans le chapitre précédent à observer des chaînes de caractères et à y chercher des correspondances. Nous allons aussi dans ce chapitre apprendre à les modifier, à l'aide de la fonction de remplacement :

> `s/a/b/` remplace la première correspondance de `/a/` dans une chaîne par `"b"`[1].

`s/chat/chien/` permet donc de remplacer par `"chien"` la *première* occurrence de `"chat"` dans une chaîne. Si nous l'appliquons à `"Le chat vient, le chat s'en va"`, nous obtenons `"Le chien vient, le chat s'en va"`.

Exercice 20. Quelle transformation a-t-on effectué pour changer `"65535"` en `"65536"` (plusieurs possibilités) ?

Dans la plupart des cas, il est beaucoup plus utile de remplacer toutes les correspondances non coïncidentes[2] d'une expression régulière dans une chaîne. On peut le faire en une seule opération à l'aide du modificateur `g`.

1. Le `s` initial est un raccourci pour « substitution ».
2. Ce sont toutes les correspondances disjointes d'une chaîne (p. 18).

Chapitre 4

Bien utiliser les expressions régulières dans un programme

Après avoir appris à construire des expressions régulières dans les chapitres précédents, le temps est venu de voir comment s'en servir. Nous allons vous montrer comment utiliser les tests dans vos programmes, comment extraire des données de chaînes de caractères ainsi que des utilisations courantes des remplacements. Nous vous donnerons ensuite quelques règles de « savoir programmer » avec les expressions régulières, en particulier ce qu'il faut éviter. Enfin, ce chapitre se terminera par deux sujets importants : la gestion des lignes et quelques pistes pour travailler dans l'international.

Tous les codes de ce chapitre sont écrits en Perl. Nous avons cependant porté une attention particulière à la syntaxe, de manière à la rendre claire et compréhensible pour tous, quitte à s'éloigner de la manière dont un « perliste » les aurait écrits[1].

4.1 Utilisations courantes

a. Les tests simples

On utilise naturellement les expressions régulières pour vérifier si une entrée de l'utilisateur ou une partie d'un fichier sont dans un format bien déterminé. Ainsi, dans le programme qui gère le formulaire en ligne de l'introduction, on pourrait trouver le code suivant :

```
if($adresse =~ /^\s*[^@]+@[^@]+\.[^@]+\s*$/)
  { utiliseAdresse($adresse); }
else {afficheErreur("Adresse incorrecte");}
```

1. Qu'ils nous en excusent par avance !

Chapitre 5

Syntaxe dans dix langages

Au contraire du chapitre précédent, où nous avons utilisé Perl comme langage de référence, nous allons maintenant détailler quelques exemples sur une sélection de dix langages. Vous y apprendrez quelles bibliothèques il faut utiliser et quels objets ou fonctions implémentent les expressions régulières. Nous vous montrerons dans chacun de ces langages comment vérifier si une chaîne correspond à une expression régulière, comment réutiliser les groupes capturés, comment progresser dans une chaîne pour trouver toutes les correspondances non coïncidentes d'une expression et enfin comment effectuer des remplacements.

Pour illustrer ces notions, nous avons implémenté dans les dix langages un exemple complet dont voici la motivation : une association a décidé de saisir toutes les informations sur ses adhérents dans une base de données. Elle a confié cette tâche à un stagiaire américain qui l'a très bien accomplie, au détail près qu'il a tapé toutes les dates (comme les dates de naissance et d'adhésion) au format américain, soit « mois-jour-année ». Depuis, les secrétaires de l'association se trompent systématiquement : il est urgent de les convertir dans un format plus habituel pour des Français, comme « jour.mois.année ».

En tant qu'adhérent de cette association, vous avez décidé de prendre vous-même en charge la conversion. Vous commencez par exporter la base de données dans un format texte, comme :

```
"Fourmond:Sophie:07-28-2004:02-14-2005:etc."
```

À l'aide d'un petit programme utilisant des expressions régulières, il est facile de modifier ce fichier texte pour ensuite le réimporter dans la base de données.

Annexe A

Solutions des exercices

A.1 Exercices du chapitre 2

Solution 1 *(énoncé p. 18).* Quand on cherche des correspondances de `/aba/` dans `"abababa"`, on trouve la première correspondance, soit `"abababa"`. Ensuite, il faut chercher les suivantes à partir de la fin de la première ; on ne trouve que `"abababa"`. En résumé, les correspondances sont :

`"abababa"`

Solution 2 *(énoncé p. 20).* Le mot que l'on recherche est constitué d'un `"b"` suivi d'un caractère inconnu, d'un `"l"`, de deux autres caractères inconnus et finalement de `"re"`. On peut donc le représenter au moyen de l'expression

```
/b.l..re/
```

Solution 3 *(énoncé p. 21).* On cherche des mots qui commencent par deux minuscules, soit « `[a-z][a-z]` », puis `"re"`, puis une minuscule, « `[a-z]` ». En combinant cela, on obtient

```
/[a-z][a-z]re[a-z]/
```

Notons que cette expression ne nous assure pas que les correspondances sont des mots entiers. Nous verrons plus loin plusieurs manières d'y remédier.

Solution 4 *(énoncé p. 21).* Nous savons maintenant exprimer spécifiquement un chiffre, au moyen de « `[0-9]` ». En remplaçant chaque point dans `/..-..-..../` par cette construction, on obtient :

```
/[0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]/
```

Cette expression élimine bien les correspondances contenant des lettres ou des espaces.

Annexe B

Que faire quand ça ne marche pas ?

Construire une expression régulière n'est pas toujours une tâche facile, d'autant qu'il n'est pas aisé d'obtenir un diagnostic. Pour vous aider, nous avons rassemblé quelques pistes à essayer quand quelque chose ne fonctionne pas.

Dans un premier temps, écrivez un petit programme de test à la manière de ceux du chapitre 5 avec l'expression régulière fautive et un exemple de chaîne sur laquelle vous souhaitez l'utiliser.

Si cela ne permet pas de cerner le problème, reconstruisez l'expression régulière morceau par morceau, en vérifiant à chaque étape les correspondances de chaque élément à l'aide de groupes capturés.

B.1 Règles générales

- Changez les « ␣ » en « `\s` » si votre implémentation le supporte ; sinon, essayez « `[␣\t\n]` » tel quel (sans doubler les antislashs).
- Essayez des quantificateurs non avides, surtout si vous utilisez « `.*` » ou « `.+` ».
- Si vous utilisez « `[a-z]` » dans des correspondances avec du texte accentué, préférez « `\w` » (voir page 71).
- Attention, « `\w` » inclut aussi les chiffres : `"78"` correspond à « `\w+` ». S'il vous faut vraiment des lettres, utilisez `/[^\W\d_]/` ou bien des constructions de type POSIX ou Unicode si c'est possible (voir page 71).
- Vérifiez la numérotation de vos groupes capturés.

Index

Symboles

Nombres

A

B

C