Introduction à la compilation

Arnaud Calmettes (nohar)

02/03/2014

Contents

Introduction **1**

Architecture et fonctionnement d'un compilateur **2**

Expressions rationnelles **6**

Analyse lexicale et syntaxique d'expressions rationnelles **8**

Arbres syntaxiques abstraits 8

Notation post-fixée et parseur à pile 10

Notation infixe et grammaires non contextuelles 12

Un lexeur et un parseur simples 17

Évaluation d'expressions rationnelles **25**

Algorithme de *backtracking* 26

Algorithme de Thompson 28

Représentation intermédiaire et algorithme de McNaughton-Yamada-Thompson 29

Implémentation de la machine virtuelle 37

Pistes d'amélioration **39**

Mise en cache de l'AFD d'une expression régulière 39

Application réaliste des expressions régulières 43

Conclusion **44**

Introduction

La compilation est sans conteste l'un des domaines les plus riches et les plus passionnants de l'informatique. Pour autant c'est aussi l'un des plus difficiles à aborder de par la diversité et la complexité des problématiques qu'il englobe et la quantité de notions théoriques qu'il fait intervenir.

Depuis les années 1950, pendant lesquelles sont nés les premiers assembleurs puis les tous premiers langages dits "de haut niveau" (Fortran, Cobol et Lisp), le monde entier de l'informatique respire grâce à ces organes vitaux que sont les compilateurs et les langages de programmation. Dès lors, ceux-ci n'ont eu de cesse de devenir de plus en plus sophistiqués, de plus en plus optimisés, de plus en plus efficaces, entraînant avec eux des générations d'ingénieurs et de chercheurs toujours plus avides de voir leurs machines faire plus, plus vite, plus facilement, plus sûrement ; il n'est donc pas surprenant de constater que le fonctionnement interne des langages de programmation modernes soit devenu hermétique au profane après tout ce temps. Cependant, aussi complexe le sujet soit-il, l'étude des compilateurs est une porte d'entrée vers la maîtrise d'un grand nombre de notions importantes d'algorithmique via leur application directe, et sa difficulté le prix à payer pour acquérir une compréhension profonde des enjeux des recherches actuelles dans diverses branches de l'informatique. Cela vaut bien la peine de verser un peu de sueur, d'autant que la création d'un langage de programmation et de son compilateur peut s'avérer être une tâche aussi amusante que stimulante !

Cela entraîne une question épineuse : comment aborder un domaine aussi riche que celui-ci sans perdre sa motivation ? D'aucuns ne jurent que par la lecture de l'incontournable référence, le fameux *Dragon Book* où sont abordés par le menu et de façon théorique tous les aspects de la compilation, alors que d'autres ouvrages que l'on trouve en abondance sur internet proposent une démarche purement pratique, en exposant l'écriture d'un compilateur ou d'un interpréteur pour un langage-jouet, quitte à renvoyer le lecteur vers le *Dragon Book* pour une introduction plus rigoureuse.

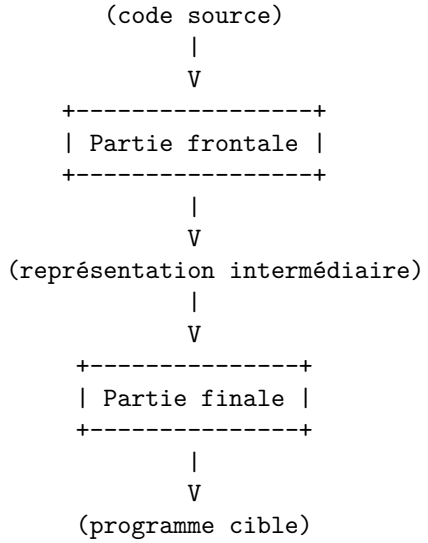
Ce texte est le premier d'une série d'articles se voulant le juste milieu entre les deux approches, en introduisant le lecteur à la compilation sans faire l'impasse sur la théorie, mais en travaillant essentiellement sur des cas concrets. Le but recherché n'est certainement pas de se substituer au *Dragon Book*, mais de proposer une initiation plus digeste à l'écriture de compilateurs, sans pour autant en masquer la difficulté. Dans cet article, nous survolerons le domaine et les différentes problématiques qui le composent en nous intéressant à un exemple complet et non trivial : nous allons créer un moteur d'expressions régulières de A à Z. Cela nous permettra d'introduire les différentes phases d'analyse opérées

par les compilateurs, tout en sensibilisant le lecteur à une approche des expressions rationnelles plus rigoureuse algorithmiquement parlant que le modèle omniprésent du backtracking.

Architecture et fonctionnement d'un compilateur

Un compilateur n'est rien d'autre qu'un programme effectuant la traduction d'un **code source** lisible et manipulable par l'être humain vers un **langage cible**.

Toute la subtilité de cette définition tient dans le caractère relativement vague de la notion de "langage cible". Contrairement à une idée reçue très répandue, le langage cible ne désigne pas nécessairement un langage machine, ni même un langage à plus bas niveau d'abstraction que le langage source ; celui-ci peut être un langage d'assemblage, un *bytecode*, ou bien un autre langage de programmation. À vrai dire, quelle que soit la nature du langage cible, celui-ci n'impacte qu'une partie limitée du compilateur, dont nous pouvons dès à présent dériver un premier schéma d'architecture.



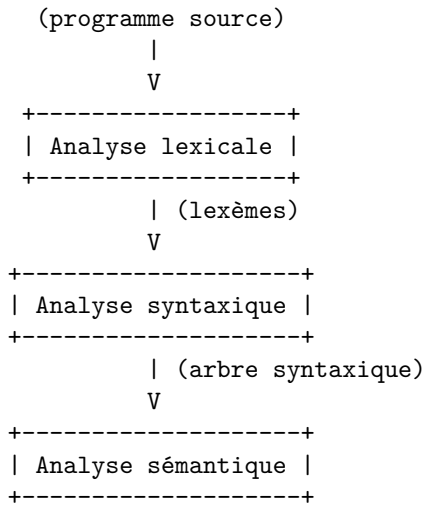
Schématiquement, la **partie frontale** (le *frontend*) du compilateur se charge de réaliser différentes analyses du code-source en même temps qu'elle le transforme en une représentation intermédiaire. Cette représentation intermédiaire est ensuite communiquée à la partie **finale** (le *backend*) du compilateur, dont le rôle est de générer le code du programme dans le langage cible, en opérant éventuellement une série d'optimisations propres à l'architecture visée. Cette séparation logique n'est pas obligatoirement visible dans tous les compilateurs,

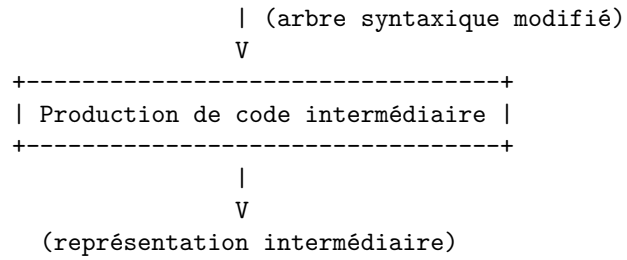
néanmoins elle facilite la conception de programmes portables, puisque leur partie frontale est totalement indépendante du langage cible, et leur partie finale indépendante du langage source. Des exemples bien connus de compilateurs exploitant cette séparation sont LLVM et le framework de GCC. En effet, GCC est conçu comme un ensemble de *frontends* (un pour chaque langage supporté) produisant la même représentation intermédiaire, elle-même comprise par une multitude de *backends* (un pour chaque architecture cible possible). Ajouter à GCC un compilateur vers un nouveau langage revient donc à écrire son *frontend* : ce compilateur supportera automatiquement toutes les architectures déjà gérées par GCC. Réciproquement, supporter une nouvelle architecture dans GCC revient à écrire un nouveau *backend* qui sera automatiquement compatible avec tous les compilateurs de la collection.

Il arrive que le travail de la partie frontale du compilateur soit lui-même désigné par le terme “compilation”. Cet usage est d’ailleurs justifié lorsque le *backend* est remplacé par un interpréteur qui, au lieu de générer le programme dans un langage cible, va lui-même exécuter le programme à partir de sa représentation intermédiaire.

On peut remarquer que la partie frontale d’un interpréteur tel que CPython, qui traduit le programme initial en *bytecode*, reste un compilateur au sens strict : suivant le point de vue que l’on désire adopter, on peut considérer que le *bytecode* est un langage cible à part entière puisqu’il est compris par une machine virtuelle, ou bien une représentation intermédiaire servant d’entrée à un *backend* comme dans le cas de LLVM. Le choix de l’une ou l’autre de ces interprétations dépend avant tout du contexte.

Classiquement, le *frontend* d’un compilateur est chargé de réaliser trois phases d’analyses : lexicale, syntaxique et sémantique. Son architecture canonique est la suivante :





Le rôle de **l'analyse lexicale** est de reconnaître les différents "mots" du programme et de leur associer une étiquette décrivant la nature de ces mots. La structure ainsi obtenue est appelée *lexème*. Prenons par exemple l'expression suivante :

```
int n = 40 + 2;
```

L'analyseur lexical d'un compilateur C pourrait produire, à partir de cette expression, la suite de lexèmes (ou *tokens*) suivante :

```
(TYPE, "int")
(IDENTIFIER, "n")
("=", "=")
(INTEGER, "40")
(OP, "+")
(INTEGER, "2")
```

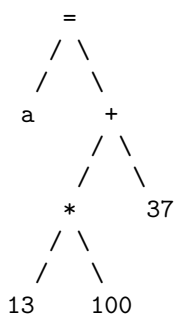
Les étiquettes (ou "labels") des lexèmes décrivent à quoi correspondent les mots en fonction du **vocabulaire** du langage. L'analyse lexicale sert donc d'une part à vérifier que les mots employés par l'utilisateur sont tous bien formés (par exemple, `8ball` est un mot qui ne peut pas exister dans des langages comme C ou Python, alors que `8-ball` serait décomposé en l'expression `8 - ball`), et d'autre part à discerner les mots de nature différente, de la même manière que l'on distingue, dans une phrase en français, les ponctuations, les noms, les adjectifs ou les verbes. Notez que dans notre exemple, le symbole `"="` n'est pas affecté à une classe particulière : sa classe est égale à sa valeur. L'utilisation de ce genre de *singletons* est justifiée : cela permet de rendre plus lisible le code qui le manipule. Alternativement, et parce que tous les analyseurs lexicaux ne supportent pas nécessairement ce genre de choses, on aurait tout aussi bien pu lui affecter la classe `EQUALS`.

Pour continuer dans notre analogie avec le français, le flux de lexèmes est ensuite envoyé en entrée d'un **analyseur syntaxique**, chargé d'assembler les mots pour reconnaître des phrases. La forme des phrases acceptées par le langage est décrite par une **grammaire** constituée d'un certain nombre de **règles**. L'analyse syntaxique permet donc de détecter des constructions erronnées telles

que “`a = + * /; 32`”, mais aussi de représenter celles-ci sous forme d'une structure arborescente, appelée **arbre syntaxique** (ou AST pour *Abstract Syntax Tree*), qui décrit plus précisément leur forme. Par exemple, pour l'expression :

```
a = 13 * 100 + 37
```

Un analyseur syntaxique pourrait produire l'arbre suivant :



Remarquez en passant que l'AST tient bien compte des règles de *priorité* des opérateurs arithmétiques. Ici, la priorité de la multiplication sur l'addition.

L'AST représentant le programme est ensuite soumis à une **analyse sémantique** qui peut éventuellement le modifier. Le rôle de celle-ci est de s'assurer que les phrases produites à l'étape précédente, qui sont *syntaxiquement correctes*, expriment quelque chose de sensé dans le langage, de la même manière que l'on peut vérifier qu'une phrase *grammaticalement correcte* en français veut effectivement dire quelque chose. Par exemple, la phrase “J'ai dévissé le gâteau sous la voile du chat.” est correctement formée du point de vue grammatical mais elle ne veut absolument rien dire. C'est durant l'analyse sémantique que sont vérifiées toutes les contraintes concernant le type des variables et des expressions, pour ne citer qu'elles. Le code suivant passera avec succès l'analyse lexicale d'un compilateur C, mais déclenchera une erreur de type lors de l'analyse sémantique puisqu'il essaye d'incrémenter une chaîne de caractères :

```
char a[] = "quarante";
a += 2;
```

Cette analyse dépend très fortement de la nature du langage qui est compilé. Par exemple, c'est à ce moment de la compilation que sera réalisée l'*inférence de types* ou le déroulement des expressions de filtrage par motifs (*pattern matching*) dans les langages fonctionnels dérivés de ML (OCaml, Haskell…), alors que les langages dynamiques tels que Python ou Javascript, beaucoup plus permissifs en comparaison, n'opéreront que très peu de vérifications ou de transformations sur l'AST. Dans certains cas, l'analyse sémantique peut même être absente : cela dépend entièrement des langages.

Expressions rationnelles

Les expressions rationnelles (ou *expressions régulières*) permettent de décrire une certaine catégorie de langages que l'on appelle *langages réguliers*. En réalité, ces langages réguliers correspondent plus à une succession de motifs que l'on utilise couramment pour reconnaître ou valider le format d'un texte.

L'expression a désigne *toutes les chaînes constituées uniquement du symbole* `a`. Il faut différencier ici le langage décrit par une expression rationnelle des opérateurs de *matching* que vous avez peut-être l'habitude d'employer. Ainsi :

- `a` est une chaîne validée par l'expression a,
- `b` ne l'est pas,
- `abcd` non plus,
- `aaa` non plus.

Les symboles élémentaires des expressions rationnelles sont évidemment des caractères. Il existe néanmoins un symbole spécial, ε (*epsilon*), qui désigne la chaîne vide.

Passons maintenant aux opérateurs. L'*union* de deux langages, notée $|$, désigne tous les motifs exprimés par l'un ou l'autre des langages. Par exemple :

- `a` est une chaîne validée par l'expression $a \mid b \mid \varepsilon$,
- la chaîne vide (ε) aussi,
- `b` aussi,
- `ab` ne l'est pas,
- `c` non plus.

La *concaténation* de deux langages réguliers est elle-même un langage régulier. Ainsi :

- `salut` est acceptée par l'expression *salut*,
- `compilation` et `compilateur` le sont par $compilat(ion \mid eur)$.

Notez que ce dernier exemple illustre la priorité de la concaténation sur l'union, ainsi que la distributivité de la concaténation par rapport à l'union :

$$a(b \mid c) = ab \mid ac$$

$$(a \mid b)c = ac \mid bc$$

Enfin, la *fermeture de Kleene* d'une expression X, notée $X*$, désigne le plus petit langage qui contienne X et ε et qui soit stable par la concaténation. En termes moins obscurs :

- $a*$ désigne les chaînes ε, `a`, `aaa`, etc.,
- $(a \mid b \mid c)*$ désigne tous les mots constitués des symboles a, b et c (`bac`, `cab`, `baba`, `abcccabcbab`, etc.).

La fermeture est l'opérateur qui a la priorité la plus forte. Ainsi, si nous classons les trois opérateurs par ordre de priorité croissante, nous avons :

1. L'union,
2. La concaténation,
3. La fermeture.

Plutôt que nous attarder plus longtemps sur les propriétés mathématiques des expressions rationnelles que nous aurons l'occasion de découvrir au fil de cet article, nous allons maintenant passer à la pratique en écrivant quelques programmes permettant de jouer avec. Mais avant cela, remarquons que les expressions régulières que nous avons l'habitude de manipuler ne comprennent pas de symbole ε. En fait, l'absence de ce symbole peut être facilement compensée par l'ajout d'un opérateur ? tel que, pour toute expression régulière X :

$$X? = X \mid \varepsilon$$

Tant que nous y sommes, offrons-nous également l'opérateur + tel que pour toute expression régulière X :

$$X+ = XX*$$

Ces deux opérateurs ont la même priorité que la fermeture ($*$).

Analyse lexicale et syntaxique d'expressions rationnelles

Arbres syntaxiques abstraits

Pour démarrer, nous allons représenter les expressions régulières sous la forme de structures arborescentes (nos arbres syntaxiques abstraits, ou AST). Nous

les implémenterons en Python. Le but de ce premier programme est uniquement d'afficher une expression régulière dans la console en fonction de son AST, comme ceci :

```
regex = cat(
            union(
                char('a'),
                char('b')),
            option(
                char('c')))

print(regex)
# affiche (a|b)c?
```

Commençons par créer trois classes, pour les trois types de noeuds de nos AST, à savoir les symboles, les opérateurs unaires (?, $*$, +) et les opérateurs binaires (l'union | et la concaténation).

```
CAT = ''
UNION   = '|'
CLOSURE = '*'
OPTION  = '?'
REPEAT  = '+'

class Char:
    def __init__(self, val):
        self.val = val


class Unop:
    def __init__(self, op, arg):
        self.op = op
        self.arg = arg


class Binop:
    def __init__(self, op, left, right):
        self.op = op
        self.left = left
        self.right = right

char = Char
option = lambda x: Unop(OPTION, x)
closure = lambda x: Unop(CLOSURE, x)
```

```
repeat = lambda x: Unop(REPEAT, x)
union = lambda a, b: Binop(UNION, a, b)
cat = lambda a, b: Binop(CAT, a, b)
```

Maintenant, nous pouvons surcharger leurs méthodes `__str__` de manière à les afficher sous la forme que nous connaissons. La seule subtilité de ce code est le placement des parenthèses : si un nœud correspond à une opération de priorité inférieure à celle de son nœud père, alors la chaîne de caractères produite doit être parenthésée. Voici une façon d'implémenter ceci :

```
# Operator precedences
prec = {
    UNION: 1,
    CAT: 2,
    CLOSURE: 3,
    REPEAT: 3,
    OPTION: 3,
}

class Char:
    def __init__(self, val):
        self.prec = 4
        self.val = val

    def __str__(self):
        return self.val

class Unop:
    def __init__(self, op, arg):
        self.op = op
        self.prec = prec[op]
        self.arg = arg

    def __str__(self):
        arg = str(self.arg)
        if self.arg.prec < self.prec:
            arg = "(%s)" % arg
        return arg + self.op

class Binop:
    def __init__(self, op, left, right):
        self.op = op
        self.prec = prec[op]
        self.left = left
        self.right = right
```

```
    def __str__(self):
        left, right = str(self.left), str(self.right)
        if self.left.prec < self.prec:
            left = "(%s)" % left
        if self.right.prec < self.prec:
            right = "(%s)" % right
        return left + self.op + right
```

Notation post-fixée et parseur à pile

Avant de nous lancer dans le *parsing* définitif de nos expressions rationnelles, examinons au passage une forme de notation commode pour décrire un arbre : la notation post-fixée. Celle-ci permet d'écrire n'importe quelle expression de manière non ambiguë sans avoir à utiliser de parenthèses ni gérer la priorité des opérateurs, puisque chaque opérateur est placé après ses opérandes. Par commodité, nous noterons `"."` l'opérateur de concaténation.

- `ab.` désigne le langage ab,
- l'expression `d?ab.c|.+` désigne le langage $(d?(ab \mid c))+$

Si cette notation demande une certaine gymnastique mentale pour être lue, elle n'en est pas moins facile à *parser*. En effet, il suffit pour cela d'utiliser une pile en lisant l'expression de gauche à droite et de réaliser les opérations suivantes pour chaque symbole :

- Si le symbole courant est un caractère, on empile simplement le nœud `char` correspondant,
- Si c'est un opérateur unaire, on dépile le dernier élément, on crée le nœud correspondant à l'opération, et on empile ce nouveau nœud,
- Si c'est un opérateur binaire, on dépile les deux derniers éléments, et on empile le nœud correspondant à l'opération.

Cela s'implémente de façon très naturelle :

```
def from_postfix(input_str):
    unops = {
        '*': closure,
        '+': repeat,
        '?': option
    }
    binops = {
```

```
        '.': cat,
        '|': union,
    }
    stack = []
    for c in input_str:
        if c in binops:
            right, left = stack.pop(), stack.pop()
            stack.append(binops[c](left, right))
        elif c in unops:
            arg = stack.pop()
            stack.append(unops[c](arg))
        else:
            stack.append(char(c))

    return stack.pop()


if __name__ == '__main__':
    while True:
        try:
            print(from_postfix(input('>>> ')))
        except IndexError:
            print("[ERROR] malformed expression")
        except KeyboardInterrupt:
            break
```

Il est important de remarquer que cette fonction ne gère pas finement les erreurs lorsque la pile est vide. En toute rigueur nous devrions donner au minimum à l'utilisateur une indication sur l'opération qui fait planter le parseur afin qu'il sache où il doit corriger son expression, et nous devrions aussi afficher un *warning* lorsque la pile n'est pas vide à la fin de la conversion. Toutefois, ce parseur n'étant pas une finalité pour nous, nous ne nous inquiéterons des mécanismes de rapports d'erreurs que plus loin dans cet article.

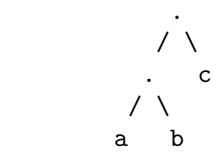
Voici un exemple d'exécution dans la console :

```
>>> co.m.p.i.l.a.t.io.n.eu.r.|.
compilat(ion|eur)
>>> compilat.......ion..eur..|.
compilat(ion|eur)
>>> d?ab.c|.*
(d?(ab|c))*
>>> abc...
[ERROR] malformed expression
```

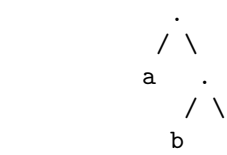
Notez dans l'exemple que plusieurs expressions postfixées donnent la même expression rationnelle. En fait, seuls les arbres syntaxiques vont différer. Si l'on

prend par exemple les expressions `ab.c.` et `abc..` nous aurons les deux arbres syntaxiques équivalents suivants :

`ab.c.` :



`abc..` :



Ceci est dû à la propriété d'associativité de la concaténation :

$$a(bc) = (ab)c = abc$$

L'union possède la même propriété :

$$a \mid (b \mid c) = (a \mid b) \mid c = a \mid b \mid c$$

Notation infixe et grammaires non contextuelles

Pour reconnaître des expressions régulières dans leur forme infixe, nous allons devoir nous attarder quelques instants sur la notion de **grammaire non contextuelle** (ou juste **grammaire**). Une grammaire est une notation permettant de décrire la syntaxe d'un langage au moyen de *règles*, et ainsi de structurer la partie frontale d'un compilateur.

Une grammaire non contextuelle comporte quatre types d'éléments :

- Les symboles **terminaux**, qui sont les symboles élémentaires du langage décrit par la grammaire. On les appelle aussi des *unités lexicales*. Dans un compilateur, les *terminaux* sont bien souvent les types associés aux lexèmes qui sortent de l'analyseur lexical.
- Les **non-terminaux** représentent des chaînes de terminaux, comme nous allons le voir dans un instant.

- Les **productions** ou **règles** de la grammaire. Une *règle* décrit une façon possible d'écrire une construction du langage. Chaque règle est associée à un non-terminal, et chaque non-terminal peut être défini par une ou plusieurs règles.
- L'**axiome**, ou *symbole de départ* de la grammaire, est le non-terminal d'où sont créées toutes les constructions du langage. Par convention, c'est généralement le non-terminal exprimé par la toute première règle de la grammaire.

Prenons la grammaire suivante comme exemple. La syntaxe employée est la *Forme de Backus-Naur* (ou *BNF*) :

```
somme   ::= somme '+' nombre
somme   ::= somme '-' nombre
somme   ::= nombre
nombre  ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Cette grammaire définit la forme d'une `somme` ne comprenant que des entiers naturels inférieurs à 10. Elle comporte :

- Deux non-terminaux : `somme` et `nombre`,
- Quatre règles pour définir ces deux non-terminaux,
- Douze terminaux : `'+'`, `'-'`, `'0'`, `'1'`, …, `'9'`,

Son axiome est le non-terminal `somme`. Pour rendre sa lecture plus aisée, nous pouvons récrire cette grammaire comme ceci :

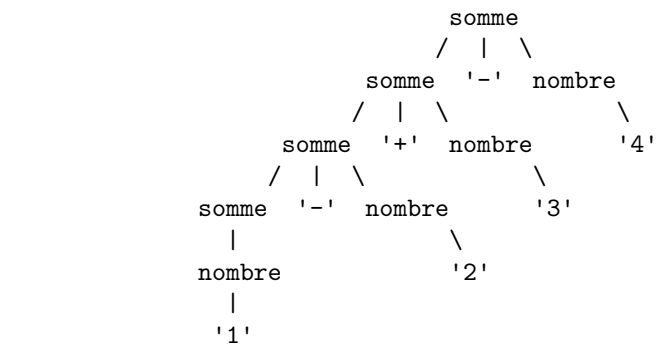
```
somme   ::= somme '+' nombre
          | somme '-' nombre
          | nombre

nombre  ::= '0' .. '9'
```

Les productions suivantes sont acceptées par cette grammaire :

- `4`
- `1 - 2 + 5 - 9`
- `5 + 7 + 2 + 1 - 0`

L'analyse syntaxique consiste à essayer de dériver les règles de la grammaire pour décrire la façon dont est structurée la chaîne d'entrée. Par exemple, la chaîne `"1 - 2 + 3 - 4"` correspond à la dérivation suivante :



Remarquons que cela signifie que l'expression a été parsée comme ceci :

```
((((1) - 2) + 3) - 4)
```

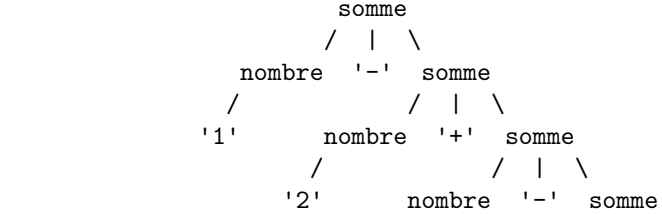
Ainsi, dans cette grammaire, les opérateurs `+` et `-` sont associatifs à gauche. Ceci est dû au fait que le non-terminal `somme` est dit *récursif à gauche* puisque certaines de ses productions commencent par lui-même. La récursion s'établit alors *à gauche* de l'opérateur, ou bien à l'extrémité gauche de la production.

Si nous voulions rendre nos opérateurs associatifs à droite, il aurait fallu écrire cette grammaire avec des productions récursives à droite pour le non-terminal `somme`, comme ceci :

```
somme   ::= nombre '+' somme
          | nombre '-' somme
          | nombre

nombre  ::= '0' .. '9'
```

Ce qui donne, pour la chaîne `"1 - 2 + 3 - 4"` la dérivation suivante :



Sur des considérations plus pratiques, il faut remarquer que les productions récursives à gauche peuvent engendrer une boucle infinie dans certains analyseurs syntaxiques (en particulier celui que nous nous apprêtons à écrire), ce qui n'est pas le cas des productions récursives à droite puisqu'elles consomment généralement des entrées avant de boucler. Cela dit, que ce soit via une récursion à droite ou à gauche ces grammaires ont le mérite de ne pas être ambiguës, contrairement à celle-ci dont il est impossible, faute de plus d'informations, de déduire l'associativité de l'opérateur `+` :

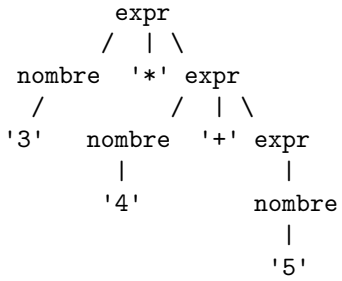
```
somme   ::= somme '+' somme
          | nombre

nombre ::= '0' .. '9'
```

Outre l'associativité, la façon dont on formule une grammaire influence également la priorité des opérateurs. Par exemple, on pourrait être tentés d'écrire la règle suivante :

```
expr    ::= nombre '+' expr
          | nombre '*' expr
          | nombre
```

Malheureusement, celle-ci n'aura probablement pas l'effet escompté sur la chaîne `"3 * 4 + 5"` :



Cela signifie que la chaîne `"3 * 4 + 5"` donne, après analyse, l'expression `3 * (4 + 5)`, ce qui n'est pas du tout le comportement attendu. Une façon simple et naturelle de régler ce problème est de passer par une règle intermédiaire :

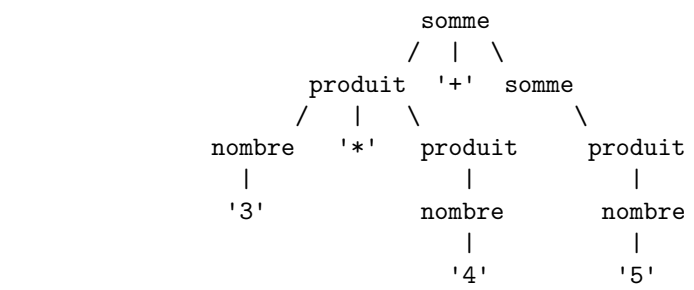
```
somme   ::= produit '+' somme
          | produit

produit ::= nombre '*' produit
          | nombre

nombre  ::= '0' .. '9'
```

Cette fois-ci, la priorité de la multiplication sur l'addition est respectée :

```
"3 * 4 + 5"
```



L'étude des différents types de parseurs et des classes de grammaires qu'ils supportent dépasse d'assez loin le cadre de cet article. En fait, elle pourrait faire l'objet d'un article entier. Ici, nous nous contenterons d'évoquer le fait que le parseur que nous allons écrire parcourt sa grammaire par *descente récursive* : on parle d'approche *top-down*, contrairement aux parseurs créés automatiquement par des générateurs d'analyseurs syntaxiques tels que Yacc, qui implémentent plutôt un algorithme de *décalage/réduction* (*shift-reduce*), ce qui correspond à une approche *bottom-up*. Pour l'heure, nous connaissons suffisamment de théorie pour formuler une grammaire des expressions régulières. Il suffit pour cela de formuler les règles en traitant les opérations par ordre de priorité croissante, et en prenant bien soin d'éviter les productions récursives à gauche :

```
expr    ::= cat '|' expr
          | cat

cat     ::= unop cat
          | unop

unop    ::= operand '*'
          | operand '?'
          | operand '+'
          | operand
```

```
% Le terminal CHAR désigne un caractère quelconque
operand ::= CHAR
          | '(' expr ')'
```

Le lecteur observateur aura remarqué que cette grammaire interdit de chaîner les opérations unaires en écrivant quelque chose comme `a+?`. À vrai dire, cela n'est pas une véritable limitation puisqu'aucune combinaison d'opérations unaires n'aurait de sens. Il est donc préférable de ne pas les prévoir dans notre grammaire : si l'utilisateur formule une telle combinaison, il vaut mieux attirer son attention dessus au moyen d'une erreur plutôt que de faire une contorsion pour la compiler. Si c'est *vraiment* ce qu'il veut faire, rien ne l'empêchera de forcer le compilateur en écrivant quelque chose comme `"(a+)?"`.

Un lexeur et un parseur simples

L'écriture de l'analyseur lexical (ou *lexeur*) n'est pas compliquée, il suffit de spécifier clairement, à l'avance, les unités lexicales que celui-ci utilise. Dans notre cas, seuls les caractères seront regroupés sous un label commun. Tous les autres symboles du langage seront des singletons :

```
# Caractères
('CHAR', 'a')
('CHAR', 'b')
...

# Opérateurs unaires
('*', '*')
('+', '+')
('?', '?')

# Union
('|', '|')
```

Nous allons définir une classe `Lexer` dotée des deux méthodes suivantes :

- La méthode `token()` consomme une partie de l'entrée et retourne le lexème (ou *token*) courant.
- La méthode `peek()` ne consomme pas d'entrée, elle retourne simplement le prochain lexème à venir. C'est ce que l'on appelle *l'opérateur de prévision*.

Lorsqu'il n'y a plus de caractères d'entrée à consommer, ces deux méthodes retournent le singleton `None`.

```
class Lexer:
    def __init__(self, input_str):
        self.it = iter(input_str)
        self.buf = None

    def token(self):
        if self.buf:
            token, self.buf = self.buf, None
            return token
        token = None
        try:
            token = next(self.it)
            if token in "*|()+?":
                token = (token, token)
            elif token == '\\':
                # '\' introduces an escape sequence
                token = ('CHAR', next(self.it))
            else:
                token = ('CHAR', token)
        except StopIteration:
            pass

        return token

    def peek(self):
        if not self.buf:
            self.buf = self.token()
        return self.buf
```

Notez que l'on autorise l'utilisateur à échapper certains caractères : si son expression contient le caractère `'+'`, il peut l'échapper en écrivant `'\+'`. Pour rester cohérents, nous allons donc modifier légèrement la méthode `__str__` de la classe `Char` :

```
class Char:
    def __init__(self, val):
        self.prec = 4
        self.val = val

    def __str__(self):
        s = self.val
        if s in "()?*+|":
            s = "\\" + s
        return s
```

Jusqu'ici, aucune difficulté. Passons maintenant à l'écriture du parseur. Voici le squelette de la classe `Parser` :

```
class ParseError(Exception):
    pass

class Parser:
    def __init__(self, lexer):
        self.lexer = lexer

    def p_expr(self):
        '''
        expr ::= cat '|' expr
               | cat
        '''

    def p_cat(self):
        '''
        cat ::= unop cat
              | unop
        '''

    def p_unop(self):
        '''
        unop ::= operand '?'
               | operand '*'
               | operand '+'
               | operand
        '''

    def p_operand(self):
        '''
        operand ::= '(' expr ')'
                  | CHAR
        '''
```

Nous nous sommes ici contentés d'écrire une méthode par non-terminal de notre grammaire, et de rappeler les règles de production en commentaire de chaque méthode. Ceci est une bonne pratique à adopter pour ne pas perdre la vue générale de la grammaire du langage lorsque l'on est plongé dans l'implémentation des règles de production. L'idée de base pour implémenter ces méthodes est d'en parcourir les productions comme ceci :

- Pour parser un non-terminal : appeler la méthode correspondante qui retournera son AST et continuer le parsing.

- Pour parser un terminal : examiner le prochain *token*. S'il correspond au prochain terminal d'une règle de production, consommer ce token et continuer le parsing. Sinon, c'est que l'expression est mal formée, donc il faut retourner une erreur à l'utilisateur.
- Une fois arrivé à la fin d'une production, générer et retourner l'AST correspondant. On dit alors que la règle est *réduite*.

Cependant, ce principe de base ne suffira pas à éliminer tous les cas d'erreur. En particulier, il ne nous permettra pas de parser élégamment le non-terminal `cat`, puisqu'aucune de ses productions ne contient de symbole terminal. Pour pallier ce problème et implémenter un mécanisme de détection d'erreurs robuste, nous allons employer une méthode bien connue des générateurs d'analyseurs lexicaux, en déterminant pour chaque non-terminal l'ensemble *Premier* des terminaux par lesquels il peut commencer, ainsi que l'ensemble *Suivant* des terminaux qui peuvent le suivre.

Commençons par les ensembles *Premier*. Et pour cela parcourons notre grammaire du bas vers le haut.

Un non-terminal `operand` peut démarrer soit par un lexème `'('`, soit par un `'CHAR'`. D'où :

```
first = {
    'operand': ['(', 'CHAR'],
}
```

Un non-terminal `unop` démarre nécessairement par un non-terminal `operand`, donc son ensemble *Premier* sera celui d'`operand`. De même, un `cat` démarre nécessairement par un `unop` et une expression démarre nécessairement par un `cat`. Ainsi, tous les non-terminaux de notre grammaire ont le même ensemble *Premier* :

```
class Parser:
    #...
    first = {
        'operand': ['CHAR', '('],
        'unop': ['CHAR', '('],
        'cat': ['CHAR', '('],
        'expr': ['CHAR', '(']
    }

    def begins(self, nonterminal):
        tok = self.lexer.peek()
        term = tok[0] if tok else None
        return term in self.first[nonterminal]
    #...
```

La méthode `begins` sert simplement à vérifier que le non-terminal passé en argument peut démarrer sur le prochain token.

Pour l'ensemble *Suivant*, il faut parcourir la grammaire de haut en bas et examiner pour chaque non-terminal les terminaux qui le suivent dans toutes les productions de la grammaire. Une `expr` peut être suivie :

- Soit de la fin du flux (on utilisera pour ce faire le singleton `None`),
- Soit, dans la production d'un `operand`, d'une parenthèse fermante.

Ainsi :

```
following = {
    'expr': [')', None],
}
```

Le non-terminal `cat` peut conclure une `expr` donc son ensemble *Suivant* contiendra celui de `expr`. Un `cat` peut également servir d'opérande gauche dans une union, donc être suivi par un `'|'`, d'où :

```
following = {
    'expr': [')', None],
    'cat': ['|', ')', None],
}
```

C'est là que cela devient intéressant : un `unop`, dans les règles de production de `cat`, peut être suivi par un `cat`. Cela signifie que l'ensemble *Suivant* de `unop` contient l'ensemble *Premier* de `cat`. Il peut être également le non-terminal par lequel se termine un `cat`, donc son ensemble *Suivant* contient également l'ensemble *Suivant* de `cat` :

```
following = {
    'expr': [')', None],
    'cat': ['|', ')', None],
    'unop': ['CHAR', '(', '|', ')', None],
}
```

Enfin, un `operand` peut être suivi de l'un des trois opérateurs unaires dans les règles de production du non-terminal `unop`, d'où :

```
class Parser:
    # ...
    following = {
```

```
        'expr': [')', None],
        'cat': ['|', ')', None],
        'unop': ['CHAR', '(', '|', ')', None],
        'operand': ['?', '*', '+', 'CHAR', '(', '|', ')', None],
    }

    def error(self, current_token, token_list):
        s = 'end of input'
        if current_token:
            s = "'%s'" % current_token[1]
        raise ParseError("Found %s. Expecting one of %s"
                         % (s, repr(token_list)))

    def check_ends(self, nonterminal):
        tok = self.lexer.peek()
        term = tok[0] if tok else None
        if term not in self.following[nonterminal]:
            self.error(tok, self.following[nonterminal])
    # ...
```

Nous nous sommes dotés au passage d'une méthode que nous appellerons après avoir fini de parser une production, et qui lèvera une exception si le token suivant dans le flux d'entrée est incohérent avec le non-terminal que nous venons de réduire. Munis de ces méthodes, nous pouvons aborder sereinement l'implémentation de nos méthodes de parsing qui, pour le coup, est devenue triviale :

```
class Parser:

    # ...

    def parse(self):
        ast = self.p_expr()
        if self.lexer.peek():
            raise ParseError("Unbalanced ')'")
        return ast

    def p_expr(self):
        '''
        expr ::= cat '|' expr
               | cat
        '''
        ast = self.p_cat()
        tok = self.lexer.peek()
        if tok and tok[0] == '|':
```

```
            self.lexer.token()
            ast = union(ast, self.p_expr())
        self.check_ends('expr')
        return ast

    def p_cat(self):
        '''
        cat ::= unop cat
              | unop
        '''
        ast = self.p_unop()
        if self.begins('cat'):
            ast = cat(ast, self.p_cat())
        self.check_ends('cat')
        return ast


    def p_unop(self):
        '''
        unop ::= operand '?'
               | operand '*'
               | operand '+'
               | operand
        '''
        ast = self.p_operand()
        tok = self.lexer.peek()
        if tok:
            if tok[0] == '?':
                self.lexer.token()
                ast = option(ast)
            elif tok[0] == '*':
                self.lexer.token()
                ast = closure(ast)
            elif tok[0] == '+':
                self.lexer.token()
                ast = repeat(ast)

        self.check_ends('unop')
        return ast


    def p_operand(self):
        '''
        operand ::= '(' expr ')'
                  | CHAR
        '''
```

```
        tok = self.lexer.peek()
        if not tok:
            self.error(tok, self.first['operand'])

        ast = None
        if tok[0] == '(':
            self.lexer.token()              # consume '('
            ast = self.p_expr()             # parse expr
            if not self.lexer.token():      # consume ')'
                raise ParseError("Unbalanced '('")
        elif tok[0] == 'CHAR':
            self.lexer.token()
            ast = char(tok[1])
        else:
            self.error(tok, self.first['operand'])

        self.check_ends('operand')
        return ast
```

Il ne nous reste plus qu'à tester ce code en réalisant une petite REPL qui parse chaque ligne comme une expression régulière :

```
if __name__ == '__main__':
    while True:
        try:
            instr = input('>>> ')
            print(Parser(Lexer(instr)).parse())
        except ParseError as e:
            print("[ERROR] %s" % e)
        except KeyboardInterrupt:
            break
```

Nous pouvons vérifier que tout fonctionne bien dans la console :

```
>>> (a|b)*
(a|b)*
>>> (((a|b)))*
(a|b)*
>>> \e\c\h\a\p\p\e\m\e\n\t\?
echappement\?
>>> a*+
[ERROR] Found '+'. Expecting one of ['CHAR', '(', '|', ')', None]
>>> (a*)+
a*+
>>> (ab(c|d)e
```

```
[ERROR] Unbalanced '('
>>> ab(c|d)e)*
[ERROR] Unbalanced ')'
>>>
```

Nous avons maintenant un parseur d'expressions régulières plutôt propre, qui nous sort des erreurs verbeuses. Nous pourrions encore améliorer ces erreurs en essayant d'obtenir quelque chose comme :

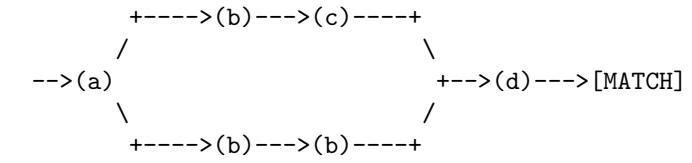
```
>>> a*+(b|c)?
[ERROR] Found '+' while expecting character, '(', '|', ')' or end of input.
Here:
      a*+(b|c)?
        ^
```

Ceci est plutôt facile à réaliser : il suffit que les tokens produits par le lexeur embarquent leur position dans la chaîne d'entrée, ce qui permettrait à la fonction d'erreur de générer cet indice visuel. L'implémentation de ce mécanisme d'erreurs est laissé au lecteur en guise d'exercice.

Évaluation d'expressions rationnelles

Lorsque Kleene a créé les expressions régulières dans les années 1950, ce n'était pas pour l'application pratique que l'on leur connaît aujourd'hui. À l'origine, il s'agissait simplement d'une notation pour décrire les langages réguliers, dont il a enrichi la modélisation de l'activité neuronale par *automates finis* que McCulloch et Pitts avaient formulée dix ans plus tôt dans un article précurseur des réseaux de neurones formels. De ce fait, les expressions régulières sont intrinsèquement liées au concept de *machine à états*, puisqu'elles en sont une représentation formelle.

Prenons par exemple l'expression rationnelle $a(bc \mid bb)d$. Celle-ci peut être représentée par l'automate suivant :



Confronter une chaîne de caractères à une expression régulière revient à parcourir l'automate associé, jusqu'à ce que l'entrée ne corresponde plus à aucun état possible de ce dernier (ce qui correspond à un échec de la reconnaissance),

ou bien que l'on tombe sur l'état final (`[MATCH]`) qui indique que l'entrée est validée. Intuitivement, on peut imaginer deux méthodes pour parcourir cet automate. La première de ces méthodes est couramment employée dans de très nombreux logiciels (`grep`, `sed`, `PCRE`…) et langages de programmation (Perl, Python, Ruby, Java…) : il s'agit du *backtracking*.

Algorithme de *backtracking*

Si l'on soumet la chaîne `abbd` à l'automate précédent, la stratégie de *backtracking* reviendrait à effectuer les opérations suivantes. Initialement, l'automate se trouverait positionné comme ceci (le symbole $*$ représente la position courante) :

```
position: .a b b d


        +---->(b)--->(c)----+
    * /                     \
-->(a)                       +-->(d)--->[MATCH]
      \                     /
       +---->(b)--->(b)----+
```

Le premier symbole de la chaîne (`a`) serait consommé. Celui-ci correspond au caractère porté par l'état initial `(a)`. Étant donné que plusieurs transitions partent de cet état, le programme doit en choisir une et garder en mémoire le chemin qu'il a pris :

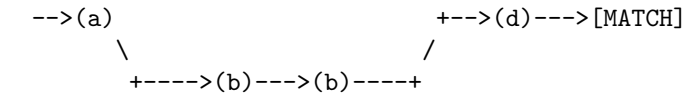
```
position: a.b b d

              *
       +---->(b)--->(c)----+
      /                     \
-->(a)                       +-->(d)--->[MATCH]
      \                     /
       +---->(b)--->(b)----+
```

Le second symbole (`b`) correspondant à l'état `(b)` courant, le programme avancerait encore d'un cran :

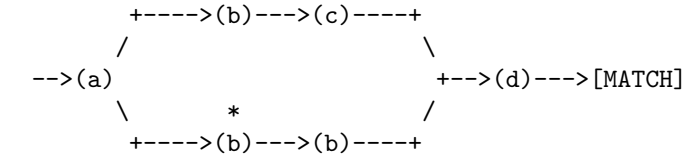
```
position: a b.b d

                     *
       +---->(b)--->(c)----+
      /                     \
```



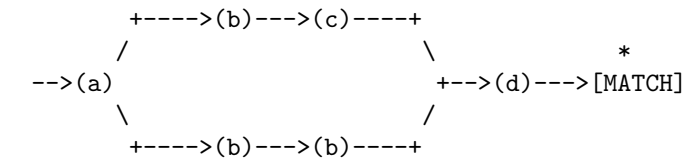
Le symbole suivant, `b`, ne correspond pas à celui attendu par l'état `(c)`. C'est ici qu'intervient le *backtracking*. Le programme va revenir sur ses pas jusqu'au dernier embranchement qu'il a pris, et reculer d'autant de caractères dans la chaîne d'entrée :

```
position: a.b b d
```



À partir de cet instant, tous les caractères de la chaîne vont correspondre aux états de l'automate. La chaîne va donc être entièrement parcourue jusqu'à ce que le programme arrive à l'étape finale :

```
position: a b b d.
```



Ici, on vérifie que la chaîne ne contient plus aucun caractère : c'est le cas. Le programme se termine donc avec succès.

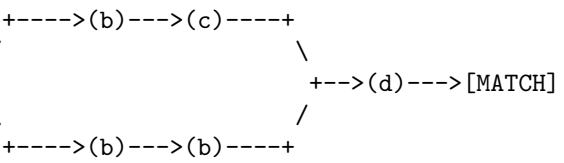
Cette méthode naïve permet aux moteurs qui l'utilisent de proposer en plus du simple *matching* un certain nombre de *features* plus complexes, comme par exemple les opérateurs de *lookahead* et *lookbehind* que l'on retrouve dans les PCRE, et qui servent à exprimer plus simplement certains motifs. Néanmoins, cette stratégie peut nécessiter dans certains cas pathologiques de revenir en arrière un très grand nombre de fois. En fait, on peut montrer que sur certaines expressions pathologiques, le *match* est réalisé avec une complexité exponentielle en $O(2^n)$. Dans cet article, nous prenons le parti de nous détacher de cette méthode qui, comparativement à celle que nous allons voir tout de suite, n'a qu'un intérêt limité en termes pédagogiques et algorithmiques.

Algorithme de Thompson

Une seconde méthode d'évaluation des expressions rationnelles, formulée en 1968 par Thompson, consiste à parcourir la machine à états d'une expression régulière en maintenant plusieurs états simultanés à la fois —donc en considérant ouvertement celle-ci comme un *Automate Fini Non déterministe* (AFN)— quitte à mémoriser des ensembles d'états dans un cache —de façon à transformer incrémentalement cet AFN en un *Automate Fini Déterministe* (AFD)—. Nous reviendrons sur les notions d'AFN et d'AFD plus loin dans cet article. Pour l'heure, examinons la façon dont l'algorithme de Thompson parcourt les expressions régulières.

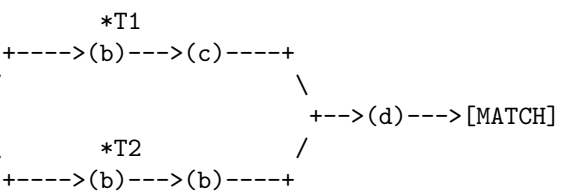
Reprenons l'état initial que nous avons vu plus haut :

```
position: .a b b d
```



À la différence de l'algorithme de *backtracking*, une fois le `a` validé par l'état `(a)` de l'automate, nous n'allons pas choisir une transition quitte à revenir en arrière, mais plutôt examiner les deux possibilités *simultanément*. Ainsi, l'état suivant se caractériserait par deux positions courantes (deux *threads*, au sens formel) :

```
position: a.b b d
```



Lorsque nous allons consommer le caractère suivant, `b`, nous nous apercevrons que celui-ci correspond à la fois à l'état courant du thread `T1` et à celui du thread `T2`. Nous faisons donc avancer les deux threads d'un cran :

```
position: a b.b d
```

```
*T1
```

```
       +---->(b)--->(c)----+
      /                    \
-->(a)                      +-->(d)--->[MATCH]
      \              *T2    /
       +---->(b)--->(b)----+
```

Ici, `T1` s'attend à voir arriver un `c` alors que le prochain caractère est un `b`. `T1` est alors détruit, ne laissant plus que `T2` en lice :

```
position: a b b.d
```

```
       +---->(b)--->(c)----+
      /                    \     *T2
-->(a)                      +-->(d)--->[MATCH]
      \                     /
       +---->(b)--->(b)----+
```

Nous connaissons la fin de l'exécution : `T2` va arriver au bout de l'automate et valider la chaîne. Par rapport à la stratégie du *backtracking*, nous n'avons pas réalisé moins de comparaisons de caractères, cependant nous n'avons parcouru notre chaîne d'entrée qu'une seule fois, en une seule passe, et nous n'avons pas eu besoin de mémoriser le chemin que nous avons parcouru. Cette stratégie, quoique moins efficace dans le cas nominal, est plus *stable* que l'algorithme de backtracking : l'évaluation d'une chaîne de caractères se fera toujours en $O(n)$. Cet algorithme est utilisé en particulier dans l'utilitaire `grep` de la distribution Unix *Plan 9*.

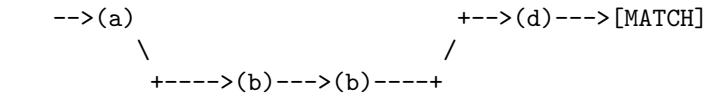
Représentation intermédiaire et algorithme de McNaughton-Yamada-Thompson

Plutôt que représenter directement nos expressions régulières par des structures `Etat` qui pointent les unes sur les autres, nous allons passer par une représentation intermédiaire sur laquelle il est plus facile de raisonner. En fait, il est possible de traduire directement un AFN tel que ceux que nous avons vu dans la section précédente en un bytecode très simple composé uniquement de **quatre** instructions.

Commençons par en voir un exemple pour l'expression $a(bc \mid bb)d$:

```
Automate:
```

```
       +---->(b)--->(c)----+
      /                    \
```




```
Bytecode:

    1:CHAR  'a'
    2:SPLIT 6
    3:CHAR  'b'
    4:CHAR  'c'
    5:JMP   8
    6:CHAR  'b'
    7:CHAR  'b'
    8:CHAR  'd'
    9:SUCCESS
```

Chaque instruction du bytecode est précédée d'un numéro de ligne, son *adresse*. La machine virtuelle qui exécutera ce bytecode maintiendra en permanence deux pointeurs :

- un **pointeur d'entrée** (ou IP pour *Input Pointer*) qui désigne la position courante dans la chaîne d'entrée du programme,
- un **pointeur d'instruction** (ou PP pour *Program Pointer*) qui désigne l'adresse de l'instruction courante.

En fait, étant donné que nous allons implémenter la stratégie de Thompson, PP ne sera pas unique : il y aura un PP par fil d'exécution. Nous pouvons maintenant détailler les *opcodes* compris par la machine virtuelle :

- L'instruction `CHAR` prend en argument le caractère à comparer à celui pointé par IP. Si les deux caractères sont égaux, IP et PP sont tous les deux incrémentés, sinon, le *thread* courant est détruit.
- L'instruction `SUCCESS`, unique, vérifie que l'entrée a bien été entièrement consommée, et quitte le programme sur un *match* positif.
- L'instruction `JMP` représente un saut inconditionnel : elle remplace la valeur de PP par l'adresse qui lui est passée en argument.
- L'instruction `SPLIT` sert à créer un nouveau *thread* dont le PP sera initialisé à l'adresse passée en argument. le PP du *thread* courant est ensuite incrémenté normalement.

Lorsque tous les fils d'exécution sont supprimés, le *match* échoue et le programme est quitté sur un résultat négatif.

Pour générer le bytecode d'une expression régulière à partir de son arbre syntaxique, nous allons utiliser ce que le *Dragon Book* nomme **l'algorithme de McNaughton-Yamada-Thompson**. Il s'agit en fait un algorithme proposé par Thompson dans son article de 1968 pour adapter celui de traduction d'une expression régulière en AFD (formulé par McNaugton et Yamada en 1960) à la création d'un AFN intermédiaire. Cet algorithme a le mérite d'être très simple à mettre en œuvre car il permet de construire un automate de façon incrémentale et directe : il consiste à décomposer l'automate d'une expression régulière en sous-automates liés entre eux par des règles de construction génériques.

Considérons par exemple deux expressions rationnelles A et B. Pour implémenter leur concaténation, nous n'avons pas besoin de connaître la structure interne de leurs automates. En fait, si nous avons le bytecode de A et celui de B, il nous suffit de mettre les deux bout à bout :

```
  1:[Début de A]
    ...
  N:[Fin de A]
N+1:[Début de B]
    ...
N+M:[Fin de B]
```

L'union de deux expressions $(A \mid B)$ consiste à séparer l'exécution en deux *threads* ; le premier exécutera le code de A et le second celui de B. Les deux vont ensuite pointer sur la même instruction en sortie :

```
  1:SPLIT N+1
  2:[Début de A]
    ...
N-1:[Fin de A]
  N:JMP M
N+1:[Début de B]
    ...
M-1:[Fin de B]
  M:_
```

Ici, l'instruction à l'adresse `M` a été remplacée par un underscore (_). Cela signifie que même si au moment où l'on génère le bytecode, nous ne savons pas encore ce qui va suivre $(A \mid B)$ dans l'expression rationnelle, rien ne nous empêche de pointer dessus.

Il ne nous reste plus qu'à définir les règles correspondant aux trois opérateurs unaires. Celles-ci sont en réalité assez simples.

Pour $A?$:

```
  1:SPLIT N+1
  2:[Début de A]
    ...
  N:[Fin de A]
N+1:_
```

Pour $A+$:

```
  1:[Début de A]
    ...
N-1:[Fin de A]
  N:SPLIT 1
```

Pour $A*$:

```
  1:SPLIT N+1
  2:[Début de A]
    ...
N-1:[Fin de A]
  N:SPLIT 2
N+1:_
```

Notez que l'instruction `SUCCESS` n'entre en considération que lorsque l'on appose la "touche finale" à la génération du bytecode. C'est elle qui conclut les instructions de parcours de l'automate.

Nous avons maintenant toutes les clés en main pour implémenter la production de code intermédiaire dans notre compilateur. Commençons par créer quatre classes, pour nos quatre instructions.

```
OPCODE_CHAR = 1
OPCODE_SPLIT = 2
OPCODE_JMP = 3
OPCODE_SUCCESS = 4


class OpSplit:
    def __init__(self, target=None):
        self.target = target

    def __str__(self):
        if self.target is None:
            raise ValueError("target of SPLIT opcode is None")
        return 'SPLIT\t%d' % self.target
```

```
    def tuple(self):
        if self.target is None:
            raise ValueError("target of SPLIT opcode is None")
        return (OPCODE_SPLIT, self.target)


class OpJump:
    def __init__(self, target=None):
        self.target = target

    def __str__(self):
        if self.target is None:
            raise ValueError("target of JMP opcode is None")
        return 'JMP\t%d' % self.target

    def tuple(self):
        if self.target is None:
            raise ValueError("target of JMP opcode is None")
        return (OPCODE_JMP, self.target)


class OpChar:
    def __init__(self, val):
        self.val = val

    def __str__(self):
        return "CHAR\t'%s'" % self.val

    def tuple(self):
        return (OPCODE_CHAR, self.val)


class OpSuccess:
    def __str__(self):
        return "SUCCESS"

    def tuple(self):
        return (OPCODE_SUCCESS, )
```

Chacune de ces classes surcharge sa méthode `__str__` de manière à pouvoir être affichée dans la console, et définit une méthode `tuple()`, dont nous nous servirons lors de l'évaluation. Nous pourrions aussi définir une véritable représentation binaire pour ces instructions, mais cela ne nous apporterait rien de plus dans le cadre de cet article. Notez que les classes `OpJump` et `OpSplit` peuvent être instanciées sans argument, mais **doivent** avoir un argument `target` défini lorsque l'on cherche à les afficher. Cela vient du fait que, comme nous allons le

voir, il est parfois nécessaire de pouvoir inclure l'une de ces instructions dans le bytecode *avant* de savoir où elles vont pointer.

Nous pouvons aussi définir une classe `Bytecode`, qui n'est rien d'autre, en fait, qu'une liste. La propriété `pos` de cette classe sert simplement à retourner l'indice de la prochaine instruction qui sera ajoutée à la représentation intermédiaire.

```
class Bytecode(list):
    @property
    def pos(self):
        return len(self)

    def __str__(self):
        return '\n'.join('%d:%s' % (idx, op) for idx, op in enumerate(self))
```

Il ne nous reste plus qu'à dériver les classes de notre AST de façon à leur ajouter une méthode `code()`. Cette méthode prend en argument un objet `Bytecode`, pour écrire le code généré dans cet objet. En dehors de cela, leur implémentation reprend exactement les règles de l'algorithme de McNaughton-Yamada-Thompson.

```
class Char:
    def __init__(self, val):
        self.prec = 4
        self.val = val

    def __str__(self):
        s = self.val
        if s in "()?*+|":
            s = "\\" + s
        return s

    def code(self, bytecode):
        bytecode.append(OpChar(self.val))


class Closure(Unop):
    def __init__(self, arg):
        Unop.__init__(self, CLOSURE, arg)

    def code(self, bytecode):
        split_end = OpSplit()
        bytecode.append(split_end)

        split_begin = OpSplit(bytecode.pos)
        self.arg.code(bytecode)
```

```
        bytecode.append(split_begin)

        split_end.target = bytecode.pos


class Option(Unop):
    def __init__(self, arg):
        Unop.__init__(self, OPTION, arg)

    def code(self, bytecode):
        split_end = OpSplit()
        bytecode.append(split_end)
        self.arg.code(bytecode)

        split_end.target = bytecode.pos


class Repeat(Unop):
    def __init__(self, arg):
        Unop.__init__(self, REPEAT, arg)

    def code(self, bytecode):
        split_begin = OpSplit(bytecode.pos)
        self.arg.code(bytecode)
        bytecode.append(split_begin)


class Cat(Binop):
    def __init__(self, left, right):
        Binop.__init__(self, CAT, left, right)

    def code(self, bytecode):
        self.left.code(bytecode)
        self.right.code(bytecode)


class Union(Binop):
    def __init__(self, left, right):
        Binop.__init__(self, UNION, left, right)

    def code(self, bytecode):
        split_right = OpSplit()
        bytecode.append(split_right)
        self.left.code(bytecode)

        jump_end = OpJump()
```

```
        bytecode.append(jump_end)
        split_right.target = bytecode.pos

        self.right.code(bytecode)
        jump_end.target = bytecode.pos
```

N'oublions pas de modifier la définition des fonctions que nous utilisons pour générer notre AST :

```
char = Char
option = Option
closure = Closure
repeat = Repeat
union = Union
cat = Cat
```

Nous pouvons maintenant modifier notre code de test, de façon à afficher, en plus de la représentation sous forme d'AST, le bytecode des expressions régulières entrées par l'utilisateur.

```
def compile_re(input_str):
    ast = Parser(Lexer(input_str)).parse()
    b = Bytecode()
    ast.code(b)
    b.append(OpSuccess())
    return ast, b

if __name__ == '__main__':
    while True:
        try:
            instr = input('>>> ')
            ast, b = compile_re(instr)
            print(ast)
            print(str(b))

        # ...
```

Vérifions dans la console :

```
>>> (a|b)*cd
(a|b)*cd
0:SPLIT 6
1:SPLIT 4
2:CHAR  'a'
```

```
3:JMP   5
4:CHAR  'b'
5:SPLIT 1
6:CHAR  'c'
7:CHAR  'd'
8:SUCCESS
>>> a(bc|bb)d?
a(bc|bb)d?
0:CHAR  'a'
1:SPLIT 5
2:CHAR  'b'
3:CHAR  'c'
4:JMP   7
5:CHAR  'b'
6:CHAR  'b'
7:SPLIT 9
8:CHAR  'd'
9:SUCCESS
```

Cela fonctionne parfaitement.

Implémentation de la machine virtuelle

Pour implémenter la stratégie de Thompson de parcours des expressions rationnelles, nous n'avons pas besoin de créer explicitement des *threads*, ni même de maintenir le pointeur IP, puisque nous savons que nous n'avons besoin de boucler au plus qu'une seule fois sur la chaîne d'entrée. L'idée de la fonction suivante est de maintenir à tout instant une liste `state` décrivant l'état courant dans lequel se trouve la VM, c'est-à-dire la liste des instructions courantes à évaluer, ainsi qu'une liste `next_state`, décrivant la liste des instructions qui seront à évaluer lorsque l'on aura "incrémenté IP", c'est-à-dire lorsque l'on bouclera sur le caractère suivant de l'entrée. Cette implémentation, qui diffère assez de la définition formelle que nous avons faite plus haut de la machine virtuelle, a le mérite de laisser la place à une optimisation intéressante que nous évoquerons dans la section suivante.

```
def match(regex, input_str):
    program = compile_re(regex)
    print(str(program))
    next_state = [0]
    for c in list(input_str) + [None]:
        print('-' * 50)
        state, next_state = next_state, []
        if not state:
```

```
            return False

        for pp in state:
            print("input: '%s' pp: %d" % (c, pp), end=' ')
            op = program[pp].tuple()

            if op[0] == OPCODE_SUCCESS:
                print('-> success?', end=' ')
                if c is None:
                    print('yes')
                    return True
                else:
                    print ('no')

            elif op[0] == OPCODE_CHAR:
                if c == op[1]:
                    print("-> continue")
                    next_state.append(pp+1)
                else:
                    print("-> drop")

            elif op[0] == OPCODE_JMP:
                print('-> add state', op[1])
                state.append(op[1])

            elif op[0] == OPCODE_SPLIT:
                print('-> add states', pp+1, op[1])
                state.append(pp+1)
                state.append(op[1])
    return False
```

Par commodité, on ajoute le singleton `None` à la fin de la chaîne d'entrée, de façon à gérer plus simplement l'instruction `SUCCESS`. Dernière subtilité : lorsque l'on tombe sur un saut `SPLIT` ou `JMP`, on ajoute simplement les PP correspondants à la fin de l'état **courant**, de manière à ce qu'ils soient exécutés durant le même tour de boucle. En dehors de cela, le code est plutôt simple à comprendre.

Voici un exemple d'exécution :

```
>>> match("a(bc|cb)d+", "abcdd")
0:CHAR  'a'
1:SPLIT 5
2:CHAR  'b'
3:CHAR  'c'
4:JMP   7
5:CHAR  'c'
```

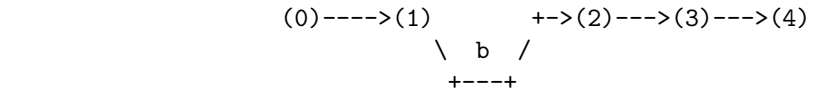
```
6:CHAR  'b'
7:CHAR  'd'
8:SPLIT 7
9:SUCCESS
-------------------------------------------------
input: 'a' pp: 0 -> continue
-------------------------------------------------
input: 'b' pp: 1 -> add states 2 5
input: 'b' pp: 2 -> continue
input: 'b' pp: 5 -> drop
-------------------------------------------------
input: 'c' pp: 3 -> continue
-------------------------------------------------
input: 'd' pp: 4 -> add state 7
input: 'd' pp: 7 -> continue
-------------------------------------------------
input: 'd' pp: 8 -> add states 9 7
input: 'd' pp: 9 -> success? no
input: 'd' pp: 7 -> continue
-------------------------------------------------
input: 'None' pp: 8 -> add states 9 7
input: 'None' pp: 9 -> success? yes
True
```

Pistes d'amélioration

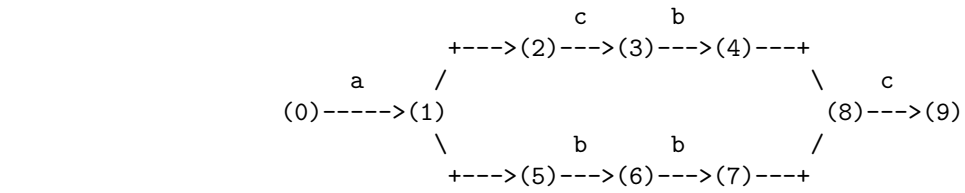
Le moteur d'expressions régulières que nous venons d'implémenter est assez basique : il permet simplement de vérifier qu'une chaîne de caractères arbitraire est conforme à une expression rationnelle. De plus, l'implémentation que nous avons étudiée dans cet article ne tient absolument pas compte des performances : elle a été pensée dans un but pédagogique. Voici quelques pistes que le lecteur peut explorer pour améliorer ce programme.

Mise en cache de l'AFD d'une expression régulière

Une première piste d'amélioration serait une optimisation de l'algorithme de *matching*, en essayant d'obtenir ce que nous appelons un *Automate Fini Déterministe*. Par exemple, un AFD équivalent de l'expression $a(cb \mid bb)c$ serait le suivant :



Où `(0)` est l'état initial de l'automate et `(4)` son état final. Par comparaison, si nous devions représenter l'AFN que l'algorithme de Thompson parcourt pour la même expression, nous aurions ceci :



La différence entre un AFN et un AFD est qu'un AFN :

- Peut contenir des *ε-transitions*, c'est-à-dire des transitions "vides" qui ne correspondent à aucun caractère d'entrée.
- Peut avoir plusieurs transitions sortant d'un état pour le même caractère d'entrée.

Par corollaire, cela signifie qu'un AFN peut, lorsqu'il est parcouru, se trouver dans plusieurs états simultanément. Un AFD, à l'opposé, ne peut se trouver qu'en un seul état à la fois, et ne peut pas avoir plusieurs transitions sortantes d'un état donné pour un caractère donné.

L'algorithme de Thompson, que nous venons d'étudier, permet de générer facilement l'AFD à partir de l'AFN d'une expression régulière. Mieux encore, il permet de générer cet AFD *partiellement* à la volée, alors même que l'on est en train d'évaluer l'expression régulière sur une chaîne d'entrée. Cela permet de mettre cet AFD en cache du programme, pour accélérer les exécutions suivantes sur la même expression régulière. En fait, cet algorithme que nous venons d'implémenter simule déjà le fonctionnement de l'AFD : nous n'avons pas nommé la liste de PP `state` pour rien puisqu'elle correspond exactement à un état de l'AFD équivalent.

Imaginons que nous soyons en train d'évaluer l'expression $a(bc \mid bb)d?$ sur l'entrée `"abbd"`.

Le *bytecode* de l'expression régulière est le suivant :

```
0:CHAR  'a'
1:SPLIT 5
2:CHAR  'b'
```

```
3:CHAR  'c'
4:JMP   7
5:CHAR  'b'
6:CHAR  'b'
7:SPLIT 9
8:CHAR  'd'
9:SUCCESS
```

L'état initial de l'AFD est bien entendu l'instruction `0`. Nous allons donc créer un nouvel état de l'AFD : `A`. Lorsque l'on évalue le premier caractère de la chaîne (`'a'`), on s'aperçoit que l'on est propulsé vers l'état `1`. D'où :

```
A := (0)
A[a] => (1)
```

Nous arrivons à un nouvel état de l'AFD que nous nommerons `B`. Lors de son évaluation, l'instruction `SPLIT` correspondant au PP `1` mènera vers les instructions `2` et `5`. En évaluant le prochain caractère d'entrée (`'b'`), les deux instructions vont matcher. La liste `next_state` vaudra donc `[3, 6]`. On peut donc enrichir notre AFD avant de passer à l'étape suivante.

```
A := (0)
A[a] => B

B := (1)
B[b] => (3, 6)
```

On peut alors créer un nouvel état `C` correspondant aux PP `(3, 6)`. Lorsque le caractère suivant (`'b'`) arrive, l'instruction `3` échoue, mais l'instruction `6` va pousser l'adresse `7` dans `next_state`. D'où

```
A := (0)
A[a] => B

B := (1)
B[b] => C

C := (3, 6)
C[b] => (7)
```

À l'étape suivante, l'instruction `7` ajoute à l'état courant les adresses `8` et `9`, et on évalue l'entrée `'d'`. L'instruction à l'adresse `9` va échouer, mais celle à l'état `8` va ajouter `9` à la liste `next_state`, d'où :

```
A := (0)
A[a] => B

B := (1)
B[b] => C

C := (3, 6)
C[b] => D

D := (7)
D[d] => (9)
```

Lors de la dernière étape, l'instruction à l'adresse `9` vérifie que l'entrée est vide, c'est le cas, d'où :

```
A := (0)
A[a] => B

B := (1)
B[b] => C

C := (3, 6)
C[b] => D

D := (7)
D[d] => E

E := (9)
E[EOI] => MATCH
```

Si nous évaluions une seconde fois cette expression régulière, mais cette fois avec l'entrée `"abcd"`, nous aurions l'exécution suivante :

L'état initial est mis en cache (`A`). Pour le caractère courant, `a`, il renvoie à l'état `B`.

À l'état `B`, on consomme l'entrée suivante `'b'`. Cette entrée est déjà gérée par le cache, on est renvoyé vers l'état `C`.

Lorsque l'on consome le caractère suivant, `'c'`, on s'aperçoit que celui-ci n'a pas encore été évalué. On récupère donc les adresses des instructions de l'état `C`, à savoir `3` et `6`, et on évalue l'AFN normalement. Cette fois-ci, l'instruction `6` échoue, mais `3` nous renvoie vers `4`. On peut donc enrichir l'état `C` :

```
C := (3, 6)
C[b] => D
C[c] => (4)
```

À l'itération suivante, on s'aperçoit que l'instruction `4` est un simple saut inconditionnel vers l'instruction `7`. Or nous avons déjà cette instruction en cache, il s'agit de l'état `D`. Cela nous permet donc de faire la liaison entre `C` et `D` :

```
C := (3, 6)
C[b] => D
C[c] => D
```

L'évaluation va ensuite parcourir l'état D puis l'état E qui sont dans le cache, avant de terminer sur un résultat positif.

Cette mise en cache permet d'accélérer énormément l'évaluation d'une expression rationnelle. Le lecteur est encouragé à l'implémenter. Bien sûr, il est également possible de générer l'AFD directement sans passer par l'évaluation de l'AFN. Cette génération automatique peut en revanche prendre un temps assez considérable dans des cas pathologiques. De plus, l'AFD résultant peut contenir un très grand nombre d'états, très largement supérieur à la taille du *bytecode*, et donc avoir un impact non négligeable sur l'empreinte mémoire du programme. Le choix de générer un AFD initialement ou non doit donc dépendre du type d'application.

Application réaliste des expressions régulières

Dans un cas réaliste, on utilise très rarement le matching comme nous venons de l'implémenter. En fait, on chercherait plutôt à savoir si une expression régulière décrit une sous-chaîne de la chaîne d'entrée, et si oui, extraire la première sous-chaîne qui valide l'expression régulière. Cela implique d'essayer d'évaluer l'expression régulière en chaque position de la chaîne d'entrée jusqu'à obtenir un *match*. On a alors deux stratégies possibles : soit l'algorithme de matching est *glouton* (*greedy*), auquel cas on retourne la sous-chaîne la plus longue qui valide l'expression régulière (c'est-à-dire que l'on exécute l'automate jusqu'à ce qu'il n'y ait plus aucun état courant, et on retourne la dernière sous-chaîne ayant atteint une instruction `SUCCESS`), soit l'algorithme est *non greedy*, auquel cas on retourne le résultat dès que l'on tombe sur une instruction `SUCCESS`.

On peut également envisager étendre notre langage des expressions régulières en lui ajoutant :

- Le symbole spécial `.`, qui correspond à *n'importe quel caractère d'entrée*,
- Les classes de caractères entre crochets (par exemple : `[A-Za-z0-9]`),
- Les classes prédéfinies du Perl, comme par exemple `\d` pour la classe `[0-9]`, et `\D` pour désigner *tout sauf un chiffre*,

Enfin, une dernière fonctionnalité indispensable à un moteur d'expressions régulières réaliste serait de rendre les parenthèses capturantes. Ainsi, chaque fois qu'une sous-chaîne valide une expression entre parenthèses, celle-ci est enregistrée soit dans un registre numéroté de 1 à 9 comme en Perl, ou bien, de façon plus commode à utiliser, dans une variable nommée. Par exemple : `(!nombre:un|deux|trois)` enregistrera la sous-chaîne définie par *un* | *deux* | *trois* dans une variable *nombre*. On peut imaginer pour cela ajouter au bytecode deux instructions `BEGIN_SUBMATCH` et `END_SUBMATCH`. Cette dernière opération est un peu plus difficile à implémenter que les deux autres. En fait, elle impose de maintenir cette fois explicitement les *threads* en mémoire, chacun d'entre eux définissant un environnement qui peut être écrasé en cas d'échec du *thread* lors d'une capture. Cela constitue un excellent exercice pour maîtriser les tenants et les aboutissants de l'algorithme de Thompson.

Conclusion

En implémentant un moteur d'expressions régulières, nous venons de réaliser un compilateur. Nous avons pu ainsi avoir un aperçu de :

- L'analyse lexicale,
- L'analyse syntaxique au moyen d'un parseur par analyse récursive descendante,
- Le développement d'arbres syntaxiques abstraits (AST),
- La production de code intermédiaire,
- L'implémentation d'un interpréteur sous la forme d'une machine virtuelle.

En fait, un langage comme Python n'est pas si différent que ça de ce que nous venons d'étudier : son *bytecode* contient beaucoup plus d'instructions et sa sémantique est différente, mais en dehors de cela, la compilation et l'interprétation d'un programme Python suit exactement les mêmes étapes. Ce que nous venons d'étudier n'est en fait rien d'autre que la base de presque n'importe quel compilateur, le lecteur peut donc dès à présent étudier chacune de ces phases dans le détail, en comparant les nombreuses approches du *parsing* et les différentes classes de grammaire, ou bien en regardant de plus près les différentes opérations qui sont réalisées lors de la compilation de son langage favori. Ce genre de recherches est extrêmement instructif.

Nous avons également étudié la notion d'automate et mis un pied dans le domaine des expressions rationnelles, dont l'énorme majorité des outils qui les utilisent suit encore, étonnament, la stratégie de *backtracking*, sacrifiant ainsi la stabilité au profit d'un plus grand nombre d'opérateurs. Le lecteur est encouragé à continuer de se renseigner sur l'exécution des expressions rationnelles, de

manière à apprendre à formuler celles qu'il utilise de façon plus optimale, en sachant ce qui se produit en coulisse.

Références

Cet article a bien évidemment été écrit avec un exemplaire du *Dragon Book* constamment ouvert à portée de main :

- Aho, A., Sethi, R., & Ullman, J. (1989). **Compilateurs. Principes, techniques et outils.** *InterEditions, Paris.*

Le lecteur y trouvera en particulier une description plus formelle de la création des ensembles *Premier* et *Suivant* des non-terminaux d'une grammaire LL(1), ainsi que de l'algorithme de McNaughton-Yamada-Thompson de génération d'un AFN.

Concernant les expressions rationnelles, ce texte fait référence à l'article fondateur de Kleene, ainsi que les travaux de McNaughton et Yamada, enrichis plus tard par Ken Thompson :

- Kleene, S. C. (1951). **Representation of events in nerve nets and finite automata**.
- McNaughton, R. & Yamada, H. (1960). **Regular expressions and state graphs for automata.** *Electronic Computers, IRE Transactions on, (1), 39-47.*
- Thompson, K. (1968). **Programming techniques: Regular expression search algorithm.** *Communications of the ACM, 11(6), 419-422.*

Sur un plan beaucoup moins formel, on trouve également une série d'articles écrits par Russ Cox, le créateur de la bibliothèque d'expressions régulières `re2`, ainsi que de l'algorithme de recherche de *Google Code Search*, à l'adresse suivante : [http://swtch.com/$_{\text{rsc/regexp/}}$](http://swtch.com/rsc/regexp/)

Ces articles ont été une source d'inspiration pour le *bytecode* de notre moteur d'expressions rationnelles. Ils décrivent également la façon dont fonctionne le moteur `re2`, écrit en C++, et qui est basé sur l'algorithme que nous avons étudié ici.