Chapitre 3

Langages formels et grammaires

Un langage est un ensemble fini ou infini de chaînes de symboles tirés d'un alphabet S.. Par exemple, si $S = \{a, b, X, !\}$ on peut définir

- $L_1 = \{a, bX, bXXa, !!, \}$
- $L_2 =$ toutes les chaînes qui commencent par a
- $L_3 =$ toutes les chaînes qui ont autant de X que de b
- $L_4 =$ les chaînes formées d'exactement 3 symboles

Si on élargit l'alphabet on peut définir les langages composés

- des chaines de lettres, de chiffres et de symboles mathématiques qui représentent des équations écrites selon la syntaxe mathématique usuelle
- des chaines de lettres, de chiffres et de symboles spéciaux qui forment un programme en langage JAVA.

La question qui se pose est : Comment définir formellement un langage ? La technique des grammaires génératives consiste à utiliser des systèmes formels particuliers, appelés grammaires formelles, pour décrire des langages. Il existe plusieurs formes de grammaires, nous commencerons par étudier les grammaires dites « hors contexte ».

3.1 Grammaires hors contexte

Définition 3.1. Une grammaire hors contexte est formée

1. d'un ensemble V de symboles non terminaux (ou variables)
2. d'une ensemble T de symboles terminaux
3. d'une ensemble R de règles de production de la forme $X \to w$ où $X \in V$ et $w \in (V \cup T)^*$ (chaîne de symboles de V et T)
4. d'un symbole initial $S \in V$

L'application d'une règle $X \rightarrow w$ sur une chaîne y consiste à remplacer l'un des symboles X contenus dans y par la chaine w. Autrement dit, si $y = pXq$ l'application de la règle produit : pwq.

Par exemple, la règle $A \rightarrow aCdC$ appliquée sur cAb produit $caCdCb$.

L'application successive de différentes règles à partir de d'une chaine u crée un *séquence de dérivation* :

$$u \xrightarrow{P_1} u_1 \xrightarrow{P_2} u_2 \ldots u_{k-1} \xrightarrow{P_{k-1}} u_k \xrightarrow{P_k} s$$

On note

$$u \xrightarrow{*} s$$

pour indiquer qu'il existe une séquence de dérivations pour obtenir s à partir de u.

Exemple 3.1. À partir des règles

1. $P_1 = C \rightarrow bbd$,
2. $P_2 = D \rightarrow bCd$,
3. $P_3 = D \rightarrow aDa$

On peut produire les séquences

$$bCaC \xrightarrow{P_1} bbbdaC \xrightarrow{P_1} bbbdabbd$$

$$D \xrightarrow{P_3} aDa \xrightarrow{P_2} abCda \xrightarrow{P_1} abbbdda$$

$$D \xrightarrow{P_3} aDa \xrightarrow{P_3} aaDaa \xrightarrow{P_3} aaaDaaa \xrightarrow{P_2} aaabCdaaa \xrightarrow{P_1} aaabbbddaaa$$

Définition 3.2. Étant donné une grammaire $G = (V, T, R, S)$, le langage $L(G)$ engendré par G est l'ensemble de toutes les chaines de symboles terminaux que l'on peut dériver à partir de S. Autrement dit, $L(G) = \{w \in T^* \mid S \xrightarrow{*} w\}$.

Exemple 3.2. Pour la grammaire $G = (V, T, R, S)$ où

$$V = \{S, X\}$$

$$T = \{a, b, c\}$$

$$R = \{S \rightarrow aTc, T \rightarrow bT, T \rightarrow b\}$$

on a

$$L(G) = \{abc, abbc, abbbc, abbbbc, \ldots\}$$

©Université de Genève – G. Falquet

C'est l'ensemble des chaines de la forme formée d'un a suivi d'un ou plusieurs b, suivis d'un c.

Pour la grammaire $H = (V, T, R, S)$ où

$$V = \{S\}$$

$$T = \{a, b\}$$

$$R = \{S \rightarrow aSb, S \rightarrow ab\}$$

on a

$$L(G) = \{ab, aabb, aaabbb, \ldots, a^n b^n, \ldots\}$$

L'ensemble des chaines qui commencent par un certain nombre de a suivis du même nombre de b.

3.2 Arbres syntaxiques

Dans de nombreux cas il est possible de dériver une même chaine en appliquant les mêmes dérivations mais dans différents ordre. Regardons par exemple les règles

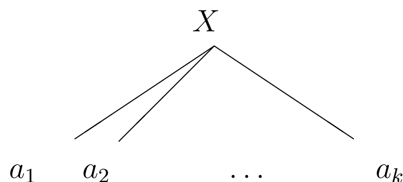
1. $S \rightarrow XY$;
2. $X \rightarrow aY$;
3. $X \rightarrow a$;
4. $Y \rightarrow c$

Les deux séquences de dérivations ci-dessous produisent la même chaîne terminale et utilisent les mêmes dérivations.

$d1 : S \xrightarrow{1} XY \xrightarrow{2} aYY \xrightarrow{4} acY \xrightarrow{4} acc$

$d2 : S \xrightarrow{1} XY \xrightarrow{4} Xc \xrightarrow{2} aYc \xrightarrow{4} acc$

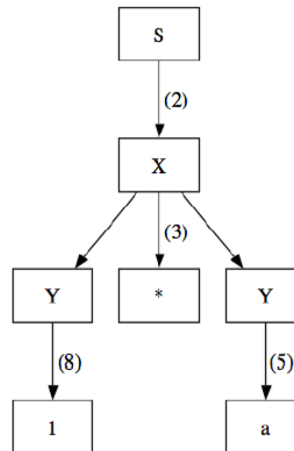
Ces deux séquences ne sont donc pas fondamentalement différentes. Pour s'abstraire de l'ordre d'application des règles on va définir des arbres syntaxiques qui représentent des ensembles de dérivations équivalentes. On construit l'arbre de la manière suivante : 1) la racine est le symbole initial, 2) une dérivation $X \rightarrow a_1 a_2 \ldots a_k$ est représentée par le sous arbre



Les feuilles sont des symboles terminaux ; la chaîne produite correspond à lire les feuilles de gauche à droite.

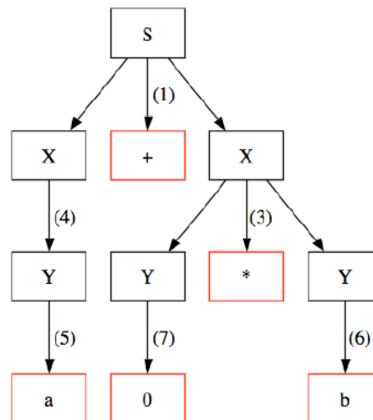
Exemple 3.3. Dérivation de la chaine 1*a .

(1) S → X + X ;
(2) S → X ;
(3) X → Y * Y ;
(4) X → Y ;
(5) Y → a ;
(6) Y → b ;
(7) Y → 0 ;
(8) Y → 1 ;



Exemple 3.4. Dérivation de la chaine a+0*b

(1) S → X + X ;
(2) S → X ;
(3) X → Y * Y ;
(4) X → Y ;
(5) Y → a ;
(6) Y → b ;
(7) Y → 0 ;
(8) Y → 1 ;



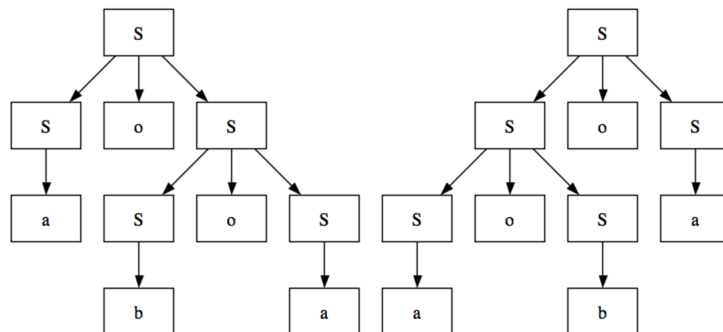
Dérivation de a+0*b

Ambiguïté

Une grammaire est ambiguë s'il existe plusieurs arbres pour dériver la même chaîne de symboles :

Si G possède les règles $\{S \rightarrow SoS, S \rightarrow a; S \rightarrow b\}$

Il y a deux arbres pour dériver la chaîne *aoboa* :



Une chaîne ambiguë possède deux arbres syntaxiques.

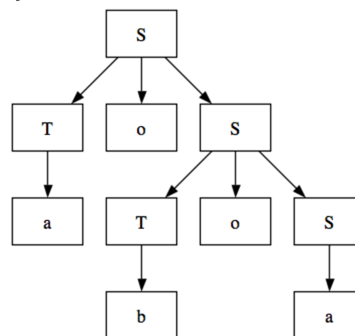
Il a donc deux manière d'analyser cette chaîne.

Exemple informel en français : *La belle ferme le voile* En utilisant la règle *Proposition* $\rightarrow$ *Sujet Verbe Complement* (La belle) (ferme) (le voile) En utilisant la règle *Proposition* $\rightarrow$ *Sujet Complement Verbe* (La belle ferme) (le) (voile)

On peut construire une grammaire non ambiguë pour le même langage

Par exemple en introduisant de nouveaux symboles non-terminaux.

$R' = \{S \rightarrow ToS, S \rightarrow T; T \rightarrow a; ; T \rightarrow b\}$



Analyse de *aoboa* (seul arbre possible)

3.3 Grammaires générales

Certains langages ne peuvent être exprimés avec des grammaires hors contexte, p.ex. $L = \{a^n b^n c^n\}$

Une grammaire générale est formée

- de symboles terminaux (T) et non terminaux (V),
- d'un symbole initial
- d'une ensemble R de règles de production de la forme $\alpha \rightarrow \beta$
  où α et β sont des chaînes de symboles tirés de V et T

Dérivation Si on a la règle

$$P = \alpha \rightarrow \beta$$

et la chaîne

$$u = p\alpha q$$

L'application de P sur α dans u produit $p\beta q$, on note

$$u \xrightarrow{P} p\beta q$$

Si α apparaît plusieurs fois dans la chaîne u, on peut appliquer P sur n'importe quelle occurrence de α , ce qui donne plusieurs dérivations différentes.

Hiérarchie de complexité des langages

Complexe signifie qu'il est difficile de répondre à la question

> « la chaîne w appartient-elle au langage ? »

Autrement dit, il est difficile de trouver une dérivation du symbole initial à w chaîne, ou de montrer qu'il n'y en a pas.

La complexité du langage dépend de la forme des règles. Chomsky a défini 4 classes de complexité pour les langages.

Type 1 : Langage récursivement énumérable

Les règles sont les plus générales possibles :

$$\alpha \to \beta$$

où α et β sont des chaînes quelconques de terminaux et non terminaux)

On ne peut même pas automatiser l'analyse. Il ne peut pas exister de programme qui accepte en entrée n'importe quelle grammaire G de ce type et une chaine w et produise la réponse « oui » si $w \in L(G)$ et « non » si ce n'est pas le cas.

Ceci montre qu'avec de simples règles de remplacement on peut engendrer des ensembles hautement complexes ! En fait, on peut décrire tout langage (ou ensemble) productible par un processus « mécanisable » par une grammaire de type 1.

Type 2 : Langage dépendant du contexte

Les règles ont la forme

$$\alpha \to \beta$$

avec longueur de $\alpha \leq$ longueur de β

L'analyse est plus facile car on peut énumérer toutes les chaines de longueur 1, puis 2, puis 3, etc. Donc on peut toujours écrire un programme qui détermine si une chaine appartient ou non au langage.

Type 3 : Langage hors contexte

Les règles sont de la forme

$$A \rightarrow \beta$$

où A est un symbole non terminal

Ces langages sont analysables avec des automate dont la seule mémoire est une pile

Type 4 : Langage régulier (linéaire)

Les règles sont de la forme

$$A \rightarrow a \text{ et } A \rightarrow aB$$

ou bien

$$A \rightarrow a \text{ et } A \rightarrow Ba$$

où A est un symbole non terminal et a un terminal

Ces langages sont analysables avec un automate à états fini (sans mémoire)

3.4 Expressions régulières

Les langages réguliers possèdent les caractéristiques suivantes :

- si L_1 et L_2 sont réguliers alors $L = L_1 \cup L_2$ est aussi régulier. On le notera $L_1|L_2$ ou $L_1 + L_2$. Cette opération s'appelle « ou » car une chaine de L appartient à L_1 *ou* à L_2.
- si L_1 et L_2 sont réguliers alors la concaténation $L = \{xy|x \in L_1, y \in L_2\}$ est aussi un langage régulier. Une chaine de L est composée d'une chaine de L_1 suivie d'une chaine de L_2. On notera L_1L_2.
- si L est régulier. alors sa fermeture de Kleene[1] $L^* = \{\epsilon\}+L+LL+LLL+L^4+L^5+\ldots$ est aussi un langage régulier. Une chaine de L^* est la concaténation de 0, 1 ou plusieurs chaines de L.
- le langage $\{a\}$ formé d'un seul symbole est régulier, on le note a.
- le langage $\{\epsilon\}$ formé de la chaine vide ϵ est régulier, on le note ϵ.

En utilisant ces opérations on peut définir un langage à partir des symboles d'un alphabet donné. Quelques exemples :

aba	le langage contenant uniquement la chaine aba
$a\|ba\|ca$	le langage contenant les 3 chaines a, ba et ca
$(bab)^*$	les chaines vide, bab, $babbab$, $babbabbab$, etc.
$(a\|b)^*$	toutes les chaines composées de a et de b
$ccc(a\|b\|c)^*aaa$	les chaines commençant pas ccc et finissant par aaa

1. Prononcer KLAY-NI

La représentation d'un nombre entier en notation décimale est soit "0" soit une séquence de chiffres qui ne commence par pas "0" et qui peut être précédée d'un signe "–" (on n'admet pas "007"). L'expression régulière correspondantes est :

$$0|((-|\epsilon)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*)$$

Il faut faire attention au placement des parenthèses et des opérateurs. Par exemple $(a \mid b)^* \neq a^* \mid b^*$!

$$(a \mid b)^* = \{ab, abab, ababab, ...\}$$

$$a^* \mid b^* = \{a, aa, aaa, ..., b, bb, bbb, ...\}$$

Application à la recherche de motifs

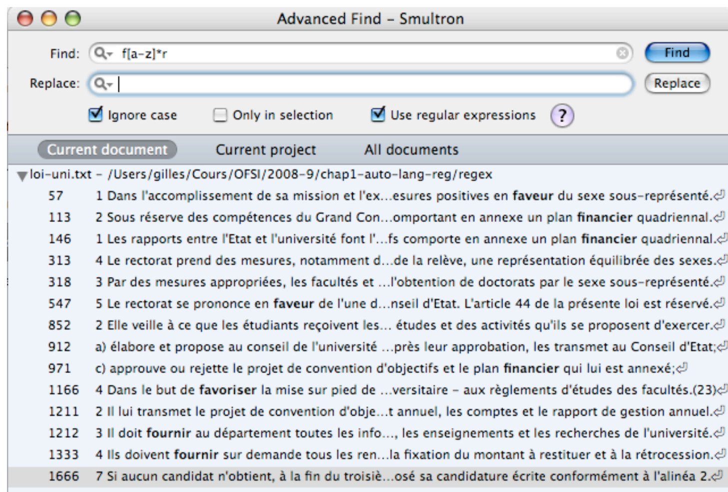
Les e.r. sont bien adaptées à la formulation de recherches de motifs dans des séquences de caractères

- fonction «recherche avancée» dans les éditeurs de texte
- recherche de motifs dans les fichiers d'un répertoire, d'un disque dur
- remplacement «sophistiqué» de sous-chaines
- etc.

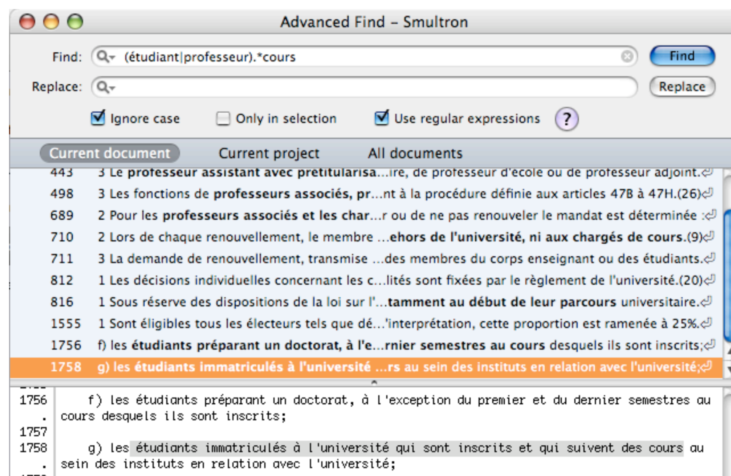
Dans la pratique on dispose d'abréviation pour simplifier l'écriture des e.r. , les plus courantes sont :

opérateur	**signification**
.	n'importe quel caractère
e^+	une ou plusieurs copies de e (e^* autorise 0 copies)
$e\{m\}$	exactement m copies de e
$e\{m,n\}$	entre m et n copies de e
$e?$	$(e\|\epsilon)$ 0 ou 1 fois e
$[x_1x_2x_3\ldots]$	x_1 ou x_2 ou x_3 ou ...
$[x$-$y]$	n'importe quel caractère comprise entre x et y
`\d`	n'importe quel chiffre
`\s`	n'importe quel espace (blanc, tab, fin de ligne, ...
`\w`	n'importe quel caractère alphanumérique
...	...

Exemple 3.5. Recherche avancée dans les éditeurs de texte. Trouver tous les mots, composés uniquement de lettres minuscules, qui commencent par f et finissent par r.



Trouver les lignes qui contiennent étudiant ou professeur et, un peu plus loin, cours.



Exemple 3.6. Utilisation de l'outil `grep` pour rechercher toutes les lignes d'un fichier qui contiennent la déclaration d'une variable de type `int` avec une initialisation par une expression commençant par une lettre.

```
$ grep 'int [a-zA-Z][a-zA-Z0-9]* *= *[a-zA-Z]' ns.java
int nodeNameStart = ix + 14; //!! length of include markup
int nodeNameEnd = s.indexOf("&amp;u=", ix); // xml jg
int paramEnd = s.indexOf(includeSuffix, nodeNameEnd);
int ifocus = s.indexOf("<focus>expanded</focus>");
int beginAnchor = s.indexOf(">", ix) + 1;
```

Exemple 3.7. Un script python pour transformer toutes les dates m/j/a en j.m.a dans un fichier.

```
    import re
f = open('tdates.txt', 'r')
for ligne in f :
transLigne = re.sub(
r"([0-9]{1,2})/([0-9]{1,2})/([0-9]{1,4})",
r"\2.\1.\3", ligne)
print (transLigne.rstrip())
```

3.5 La notation BNF

Inventée par J. Backus et P. Naur (1959) pour décrire le langage ALGOL, cette notations (légèrement étendue) reste la plus utilisée pour décrire la syntaxe des langages informatiques (langages de programmation, de bases de données, de configuration, etc.)

Le but est d'obtenir une notation plus compacte pour les grammaires hors contexte, en réduisant le nombre de règles.

Les règles BNF s'écrivent

$$\langle \textit{symbole non terminal} \rangle ::= \textit{expression régulière}$$

Les symboles non terminaux s'écrivent entre < et >, les terminaux entre guillemets ou apostrophes.

Exemple 3.8. Langage de définition de l'identité d'une personne

<identité> : := <nom> | <numéro-portable>
<nom> : := <prénom> <nom-famille>
<prénom> : := <seq-lettres>
<seq-lettres> : := <lettre> | <lettre> <seq-lettres>
<lettre> : := "A" | "B" | ... | "z"
<numéro-portable> : := <seq-chiffres>
<seq-chiffres> : := <chiffre> | <chiffre> <seq-chiffres>
...

Extension de la notation BNF

Dans les expressions régulières on utilise généralement { et } à la place de * pour la répétition et [et] pour l'option (0 ou une fois).

$e\,?$ ou $[e]$	élément optionnel
e^* ou $\{e\}$	répétition 0, 1 ou plusieurs fois
e^+	répétition 1 ou plusieurs fois
$e_1 \mid e_2$	alternatives

Correspondances avec les règles classiques

©Université de Genève – G. Falquet

$\langle A \rangle ::= e_1 \mid e_2$	$A \rightarrow e_1$ $A \rightarrow e_2$
$\langle A \rangle ::= [e]$	$A \rightarrow e$ $A \rightarrow \epsilon$
$\langle A \rangle ::= \{e\}$	$A \rightarrow \epsilon$ $A \rightarrow eA$

Exemple 3.9. identité amélioré

<identité> : := <nom> | <numéro-portable>
<nom> : := <prénom> <nom-famille>
<prénom> : := <capitale> {<lettre>}{(” ” | ”-”) <capitale> {<lettre>}}
<nom-famille> : := <capitale> {<lettre>} {(” ” | ”-”) <capitale> {<lettre>}
<capitale > : := ”A” | ... | ”Z”
<lettre> : := ”a” | ... | ”z”
<numéro-portable> : := [”+” <chiffre> [<chiffre>]] <chiffre> {<chiffre>}
<chiffre> : := ”0” | ... | ”9”

Exemple 3.10. expression arithmétique (avec une version de l’EBNF sans < et >)

Expression : := Terme {(”+” | ” ?”) Terme }
Terme : := Facteur { (”*” | ”/”) Facteur }
Facteur : := Nombre | Variable | ”(” Expression ”)”
Nombre : := [”-”] Chiffre {Chiffre}
Chiffre : := ”0” | ”1” | ...... | ”9”
Variable : := Lettre { Lettre | Chiffre | ”” | ”$”
Lettre : :=”a”|”b”|......|”Z”

Exemple 3.11. BNF de la BNF

syntax : :={rule}
rule : :=identifier” : :=”expression
expression : :=term{”|”term}
term : :=factor{factor}
factor : :=identifier | quoted_symbol| ”(” expression”)” | ”[” expression”]” |”{” expression”}”
identifier : :=letter{letter|digit}
quoted_symbol : :=”””{any_character}”””

Exercices

1. Décrivez les langages engendrés par les grammaires ci-dessous
   (a) $G = (\{S\}, \{a\}, S, R)$ avec $R = \{S \rightarrow aaa, S \rightarrow aaaS\}$.
   (b) $G = (\{S\}, \{a, b, c\}, S, R)$ avec $R = \{S \rightarrow 0, S \rightarrow aSc, S \rightarrow bSc\}$.
   (c) $G = (\{S\}, \{a, b\}, S, R)$ avec $R = \{S \rightarrow \epsilon, S \rightarrow aSa, S \rightarrow bSb\}$.

2. On a une grammaire formelle G composée des symboles non terminaux $\{E, T, V, C\}$, des symboles terminaux $\{0, 1, p, q, r, s, \&, |, (,)\}$, du symbole initial E et des règles

$$
\begin{array}{lll}
E \rightarrow T & T \rightarrow V & V \rightarrow r \\
E \rightarrow T \ \& \ T & T \rightarrow C & C \rightarrow 0 \\
E \rightarrow T \mid T & V \rightarrow p & C \rightarrow 1 \\
T \rightarrow (E) & V \rightarrow q &
\end{array}
$$

   (a) Parmi les chaînes suivantes, dites lesquelles appartiennent au langage engendré par G et donnez leur arbre syntaxique.
   - 0
   - $p \mid 0$
   - $(p\ q)$
   - $(1 \ \& \ r)$
   - $p \ \& \ ((q \mid 1) \ \& \ 0)$
   - $p \mid q \mid r$
   - $(((())))$
   - $((p \ \& \ q) \mid 1) \mid (p \ \& \ (q \mid 0))$

   (b) Cette grammaire est-elle ambiguë ?

3. Définissez une grammaire formelle hors contexte pour chacun des langages ci dessous :
   (a) toutes les chaînes ne contenant que des a
   (b) les chaînes de $\{a, b\}^*$ qui ne contiennent qu'un seul b.
   (c) les chaînes de $\{a, b\}^*$ contenant exactement 6 b et un nombre quelconque de a.
   (d) les chaînes de $\{a, (,)\}^*$ dans lesquelles les parenthèses sont équilibrées par exemple la chaîne $aaa(a)aa(aaaa(aaa)aa(aa)a)$
   (e) les listes sur le vocabulaire $\{a, b, c\}$. Une liste est une expression qui commence par une parenthèse "(", contient une séquence (éventuellement vide) d'éléments séparés par des virgules et se termine par une parenthèse ")". Un élément est soit un symbole du vocabulaire, soit une liste.. Quelques exemples : $(a, b), (a, b, (a, c)c), ((())), (((a)), ((e)))$

   (f) même chose mais on admet qu'une liste peut commencer soit par l'une des parenthèses (, [, <, ou {, et se terminer par la fermeture correspondante :),], > ou }

   (g) les définitions de processus. Un processus est soit un nom (suite de lettres), soit une suite d'autres processus connectés par les opérateurs ■ (séquence), `||` (parallèlisme) ou `**` (alternative). On peut utiliser les parenthèses pour imbriquer les définitions. Par exemple : `metro ■ boulot ■ dodo`, `manger ■ (boire ** conduire)`.

4. Trouvez des expressions régulières pour représenter les langages suivants.

   (a) L'ensemble des dates comprises entre le 1.1.2000 et le 31.12.2022, en supposant que tous les mois ont 31 jours. On admettra donc la chaîne 31.4.2004.

   (b) L'ensemble des dates comprises entre le 1.1.2000 et le 31.12.2022, en tenant compte de la longueur de chaque mois, sauf pour février que l'on fixe à 29. Donc on admettra le 29.2.2001, mais pas le 31.04.2006.

   (c) L'ensemble des dates comprises entre le 1.1.2000 et le 31.12.2022, en tenant compte des années bissextiles pour février.

   (d) L'ensemble des noms complets de fichiers Windows (ou Linux). Par exemple : `/usr/documents/exercices/e1.doc` .

   (e) L'ensemble des URL admis par les navigateur web. Par exemple :
   `http://www.truc.com/bla/blo/blu.html`
   ou `http://www.truc.com/bla/blo/`
   ou `http://www.truc.ch:8181/bla/blo/blu.php?pos1=56`
   La définition précise de la syntaxe se trouve dans
   `http ://www.ietf.org/rfc/rfc3986.txt`