Deuxième partie

Analyse syntaxique

Plan

5 Analyse lexicale : expressions régulières et automates

L'analyse lexicale consiste à découper le texte d'un programme en une séquence de mots. On s'intéresse ici à deux questions :

1. *comment spécifier l'ensemble des mots utilisables dans un langage de programmation ?*
2. *en partant d'une telle spécification, comment réaliser un programme d'analyse efficace ?*

5.1 Expressions régulières

Avant de regarder l'analyse lexicale elle-même, nous allons nous intéresser à la manière de décrire l'ensemble des mots utilisables dans un programme écrit dans un langage donné.

Mots et langages formels. Un ***mot*** est une séquence de caractères, les caractères étant pris dans un ensemble A appelé ***alphabet***.

$$m = a_1 a_2 \dots a_n$$

L'ensemble des mots sur l'alphabet A est noté A^*. La ***longueur*** d'un mot est le nombre de caractères dans la séquence. Le ***mot vide*** est l'unique mot formé de zéro caractères, on le note ε. La ***concaténation*** de deux mots $m = a_1 \dots a_n$ et $m' = b_1 \dots b_k$, simplement notée mm', est le mot $a_1 \dots a_n b_1 \dots b_k$ formé en plaçant à la suite les caractères de m et ceux de m'. Dans ce chapitre, on appelle ***langage*** un ensemble de mots. Un langage L sur l'alphabet A est donc un sous-ensemble de A^*.

Dans le cadre d'un langage de programmation on considère différentes formes de mots, dont par exemple :

— des mots clés : `if`, `fun`, `while`...
— des opérateurs et autres symboles : `+`, `*`, `;`, `{`, `}`...
— des nombres : `123`, `123.45`...

Les symboles comme `+` ou `;` correspondent à des caractères isolés, et les mots-clés à des séquences prédéfinies de symboles. Les nombres introduisent quelques idées additionnelles, comme l'alternative entre plusieurs formats ou la répétition arbitraire de certains types de symboles (on ne fixe pas a priori de nombre maximal de chiffres dans l'écriture d'un entier !).

Expressions et langages associés. Une ***expression régulière*** décrit un ensemble de mots à l'aide des éléments que nous venons d'énumérer : caractères isolés, séquences, alternatives et répétitions. Voici les différentes formes possibles d'une telle expression.

e	$::=$	$\emptyset$	ensemble vide
	$\mid$	ε	mot vide
	$\mid$	a	caractère
	$\mid$	$e_1\ e_2$	séquence
	$\mid$	$e_1 \mid e_2$	alternative
	$\mid$	e^*	répétition

Notez qu'il s'agit d'une définition inductive !

À chaque expression régulière e on peut associer un langage $L(e)$, qui est l'ensemble des mots décrits par cette expression. En suivant la structure inductive des expressions régulières, on définit $L(e)$ par des équations récursives sur chaque forme d'expression régulière.

$$\begin{aligned}
L(\emptyset) &= \emptyset \\
L(\varepsilon) &= \{\varepsilon\} \\
L(a) &= \{a\} \\
L(e_1 e_2) &= \{m_1 m_2 \mid m_1 \in L(e_1) \wedge m_2 \in L(e_2)\} \\
L(e_1|e_2) &= L(e_1) \cup L(e_2) \\
L(e^*) &= \bigcup_{n\in\mathbb{N}} L(e^n)
\end{aligned}$$

Il faut bien distinguer dans cette définition le langage vide $\emptyset$ (contenant zéro mot) et le langage $\{\varepsilon\}$ contenant le mot vide (et donc contenant un mot). Le langage associé à l'étoile est défini à l'aide d'une expression auxiliaire e^n désignant n répétitions de l'expression e.

$$\begin{aligned}
e^0 &= \varepsilon \\
e^{n+1} &= e e^n
\end{aligned}$$

À noter : la répétition e^* désigne une répétition de e un nombre quelconque de fois, *qui peut être zéro.*

Exemples. Voici quelques expressions régulières sur l'alphabet $\{a, b\}$ et leurs interprétations :
- $(a|b)(a|b)(a|b)$: mots de 3 lettres
- $(a|b)^*a$: mots terminant par un a
- $(b|\varepsilon)(ab)^*(a|\varepsilon)$: mots alternant a et b

Exemples, dans l'autre sens :
- Constantes entières positives. Si l'on admet l'écriture de zéros inutiles à gauche, comme 00123 ou 000, on peut proposer l'expression régulière $(0|1|\dots|9)(0|1|\dots|9)^*$: on impose d'avoir au minimum un chiffre, suivi par une quantité arbitraire de chiffres supplémentaires. Si on veut en revanche interdire les zéros superflus, il faut alors imposer que tout nombre commence par un chiffre non nul, à part le nombre 0 lui-même. On obtient alors : $0|((1|2|\dots|9)(0|1|2|\dots|9)^*)$
- Nombres binaires à virgule (en autorisant les zéros superflus à gauche comme à droite) : $(\varepsilon|-)(0|1)(0|1)^*(.(0|1)^*|\varepsilon)$. Note : cette solution n'autorise pas la notation .1 souvent vue dans les langages de programmation pour 0.1.

 On sera attentif aux priorités dans l'écriture des expressions régulières. L'opération * a la priorité la plus importante, vient ensuite la séquence puis l'alternative. Ainsi l'expression régulière $.(0|1)^*|\varepsilon$ se parenthèse en $(.((0|1)^*))|\varepsilon$.

Expressions régulières étendues. On utilise couramment quelques formes additionnelles, qui sont des raccourcis pour certaines expressions.
- e^+ : répétition stricte (au moins une fois) $e^+ = ee^*$
- $e?$: option $e? = (e|\varepsilon)$
- $[a_1 \dots a_n]$: choix de caractères $[a_1 \dots a_n] = (a_1|\dots|a_n)$
- $[\hat{}a_1 \dots a_n]$: exclusion de caractères (admet n'importe quel caractère de l'alphabet hors de ceux énumérés)

 Les crochets carrés (avec ou sans exclusion), ne permettent de désigner que des ensembles de caractères, pas des mots en général.

Représentation en caml. Les expressions régulières étant des objets inductifs, on peut les représenter en caml à l'aide d'un type algébrique.

```
type regexp =
  | Vide
  | Epsilon
  | Caractere   of char
  | Sequence    of regexp * regexp
  | Alternative of regexp * regexp
  | Repetition  of regexp
```

On pourrait alors imaginer écrire une fonction récursive qui prend en paramètre une expression régulière de type `regexp` et qui renvoie l'ensemble des mots reconnus. À un détail près : la répétition génère un ensemble infini, que l'on ne peut pas représenter à l'aide d'une structure de données ordinaire. Nous allons prendre une approche différente, et nous concentrer sur une fonction qui teste si un mot appartient au langage d'une expression régulière donnée.

Ainsi, les expressions régulières sont des termes décrivant des ensembles de mots. La suite du chapitre s'intéressera principalement aux points suivants :
- quels ensembles de mots peuvent ou non être décrits par des expressions rationnelles ?
- quels outils algorithmiques permettent un test d'acceptation le plus efficace possible ?
- comment transformer cela en un outil d'analyse lexicale ?

Approfondissement : un algorithme de test d'appartenance d'un mot au langage d'une expression régulière. On dit qu'un mot m est ***accepté*** par une expression régulière e lorsque $m \in L(e)$. On peut caractériser simplement la propriété $m \in L(e)$ en suivant la structure inductive de e et en appliquant la définition de $L(e)$. On obtient cependant des conditions comme $m \in L(e_1e_2) \iff \exists m_1, m_2,\ (m = m_1m_2) \wedge (m_1 \in L(e_1)) \wedge (m_2 \in L(e_2))$ qui donnent une spécification mathématique tout à fait convenable, mais une complexité algorithmique déplorable : faut-il essayer toutes les manières de décomposer m en deux parties, et tester pour chacune $m_1 \in L(e_1)$ puis $m_2 \in L(e_2)$?

Pour obtenir un algorithme de test raisonnable, on propose de raisonner plutôt par récurrence sur le mot m. On se ramène alors à deux questions :

— $\varepsilon \in L(e)$: quelles expressions acceptent le mot vide ?
— $am \in L(e)$: que peut-on accepter après a pour former un mot accepté ? autrement dit, que reste-t-il de l'expression e après prise en compte du caractère a ?

Pour répondre à chacune de ces questions, on va cette fois bien raisonner sur la structure de l'expression rationnelle, en suivant la définition de $L(e)$.
Acceptation du mot vide : on donne une fonction accepte_eps telle que accepte_eps(e) indique si $\varepsilon \in L(e)$.

$$\begin{aligned}
\text{accepte_eps}(\emptyset) &= \text{Faux} \\
\text{accepte_eps}(\varepsilon) &= \text{Vrai} \\
\text{accepte_eps}(a) &= \text{Faux} \\
\text{accepte_eps}(e_1 e_2) &= \text{accepte_eps}(e_1) \wedge \text{accepte_eps}(e_2) \\
\text{accepte_eps}(e_1 | e_2) &= \text{accepte_eps}(e_1) \vee \text{accepte_eps}(e_2) \\
\text{accepte_eps}(e^*) &= \text{Vrai}
\end{aligned}$$

Résidus : étant donnés une expression régulière e et un caractère a, on définit une expression régulière $e' = \text{résidu}(e, a)$ représentant l'ensemble $\{ m \mid am \in L(e) \}$ des mots qui peuvent être lus après a pour former un mot de $L(e)$.

$$\begin{aligned}
\text{résidu}(\emptyset, a) &= \emptyset \\
\text{résidu}(\varepsilon, a) &= \emptyset \\
\text{résidu}(b, a) &= \begin{cases} \varepsilon & \text{si } b = a \\ \emptyset & \text{si } b \neq a \end{cases} \\
\text{résidu}(e_1 e_2, a) &= \begin{cases} \text{résidu}(e_1, a) e_2 & \text{si } \varepsilon \notin L(e_1) \\ \text{résidu}(e_1, a) e_2 \mid \text{résidu}(e_2, a) & \text{si } \varepsilon \in L(e_1) \end{cases} \\
\text{résidu}(e_1 | e_2, a) &= \text{résidu}(e_1, a) \mid \text{résidu}(e_2, a) \\
\text{résidu}(e^*) &= \text{résidu}(e, a) e^*
\end{aligned}$$

Ces fonctions peuvent être directement traduites en code caml, pour donner une fonction de test plus efficace que l'approche naïve.

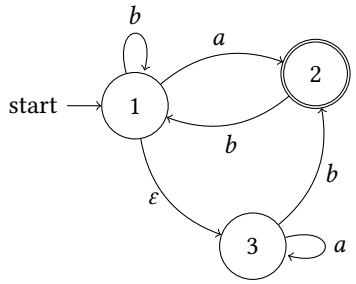
```
let rec accepte_eps = function
  | Vide               -> false
  | Epsilon            -> true
  | Caractere a        -> false
  | Sequence(e1, e2)   -> accepte_eps e1 && accepte_eps e2
  | Alternative(e1, e2) -> accepte_eps e1 || accepte_eps e2
  | Repetition(e)      -> true

let rec residu e a = match e with
  | Vide        -> Vide
  | Epsilon     -> Vide
  | Caractere b -> if b=a then Epsilon else Vide
  | Sequence(e1, e2) ->
     let e3 = Sequence(residu e1 a, e2) in
     if accepte_eps e1 then
       Alternative(e3, residu e2 a)
     else
       e3
  | Alternative(e1, e2) -> Alternative(residu e1 a, residu e2 a)
  | Repetition(e)       -> Sequence(residu e a, Repetition e)

let rec accepte m e = match m with
  | []    -> accepte_eps e
  | a::m' -> accepte m' (residu e a)
```

Nous verrons dans la suite du chapitre des techniques encore plus élaborées pour résoudre ce problème de l'acceptation.

5.2 Automates finis

Nous avons vu que les expressions régulières permettaient de décrire des ensembles de mots, mais n'étaient pas directement utilisables à des fins algorithmiques. Les *automates finis* que nous présentons ici sont des dispositifs algorithmiques dédiés à la manipulation de séquences de caractères, et particulièrement adaptés aux expressions régulières.

Automates. Un ***automate fini*** sur un alphabet A est un graphe orienté avec un nombre fini de sommets, dont chaque arc est étiqueté par un caractère de A ou par le mot vide ε. Dans l'étude des automates on s'intéresse aux chemins entre des sommets de départ et des sommets d'arrivée fixés, ou plus précisément aux caractères rencontrés le long de ces chemins. Dans les schémas, on indiquera les sommets de départ par une flèche entrante extérieure étiquetée par *start* (ci-dessous, le sommet numéro 1), et les sommets d'arrivée par un cercle double (ci-dessous, le sommet numéro 2).



Formellement, un ***automate fini*** sur un alphabet A est décrit par un quadruplet (Q, T, I, F) où :

- Q est un ensemble fini dont les éléments sont appelés des ***états***,
- T est un ensemble de triplets de $Q \times (A \cup \{\varepsilon\}) \times Q$ appelés ***transitions***,
- I est un ensemble d'états ***initiaux***, et
- F est un ensemble d'états ***acceptants***.

Une transition (q, a, q') d'un état de départ q vers un état d'arrivée q' étiquetée par la lettre a est encore notée $q \xrightarrow{a} q'$. On note de même $q \xrightarrow{\varepsilon} q'$ pour une transition étiquetée par le mot vide ε.

On dit qu'un mot m sur l'alphabet A est ***reconnu*** (ou ***accepté***) par un automate (Q, T, I, F) dès lors qu'il existe dans cet automate un chemin $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \dots q_{n-1} \xrightarrow{a_n} q_n$ qui :

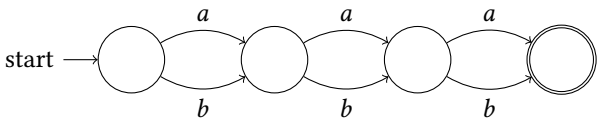
- part d'un des états initiaux : $q_0 \in I$,
- est étiqueté par les caractères de m, pris dans l'ordre : $m = a_1 a_2 \dots a_n$ et
- arriv à l'un des états acceptants : $q_n \in F$.

Note : dans la lecture des étiquettes du chemin, on ignore les ε (mais on fait bien la transition entre l'état de départ et l'état d'arrivée d'une éventuelle transition ε utilisée).

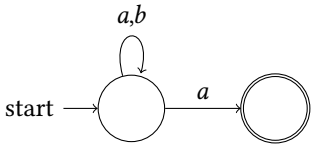
Le ***langage reconnu*** par un automate est l'ensemble de ses mots reconnus, c'est-à-dire l'ensemble des mots correspondant à tous les chemins possibles d'un état initial à un état acceptant.

Exemples. Voici quelques langages déjà évoqués, et des automates les reconnaissant.

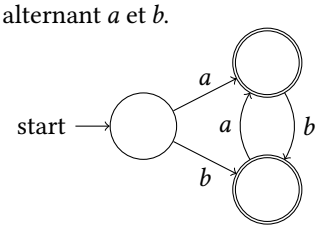
- Mots de 3 lettres sur l'alphabet $\{a, b\}$.



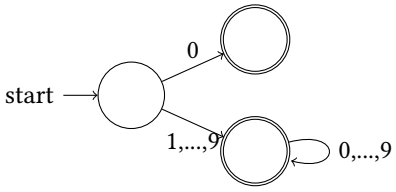
- Mots sur l'alphabet $\{a, b\}$ terminant par un a.



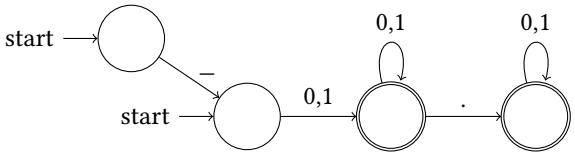
- Mots alternant a et b.



- Constantes entières positives, sans zéros superflus.



— Nombres binaires à virgule, positifs ou négatifs, avec zéros superflus possibles.



Automates et algorithmes. Un automate peut être vu comme un algorithme d'un type extrêmement contraint :

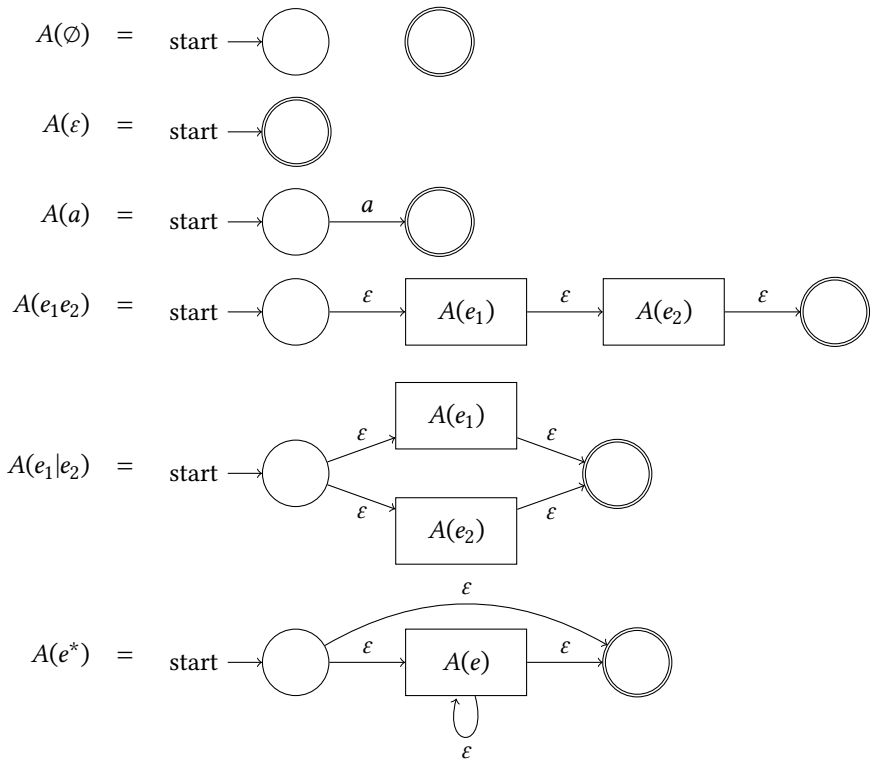
— il prend en entrée un mot,
— il renvoie comme résultat un booléen,
— il n'utilise qu'une quantité finie prédéterminée de mémoire,
— il lit son entrée de gauche à droite
— et, si possible, il répond sans avoir à faire de retour en arrière.

Les clés de lecture suivantes permettent de comprendre le lien entre un automate et un tel algorithme :

— les états de l'automate représentent les configurations possibles de la mémoire,
— une transition donne, en fonction d'une configuration de départ de la mémoire et d'un caractère lu dans l'entrée, une nouvelle configuration de la mémoire, autrement dit chaque transition décrit une modification à faire sur la mémoire en fonction de l'entrée,
— le langage reconnu par un automate est l'ensemble des entrées pour lesquelles l'algorithme renvoie Vrai.

Construction d'un automate, première approche. Partant de n'importe quelle expression régulière e, il est possible de construire un automate fini reconnaissant le même langage que e. Une approche particulièrement simple pour cela est la construction de Thompson, qui procède en suivant la structure récursive de l'expression régulière.

Voici une description synthétique de cette construction, où on note $A(e)$ l'automate produit pour l'expression régulière e. L'automate $A(e)$ a la particularité de posséder un unique état initial et un unique état acceptant (ce qui facilite la manipulation), et également de posséder de nombreuses transitions ε (ceci facilite la définition, mais ne sera pas forcément une bonne chose à terme).



Note : dans ces schémas, une transition allant vers la pastille représentant un automate $A(e)$ désigne une transition dont la cible est l'unique état inital de $A(e)$. Symétriquement, une transition partant de $A(e)$ part de son unique état acceptant.

Dans un tel automate, un même mot peut étiqueter de multiples chemins partant de l'état initial. Le test de reconnaissance par recherche de chemins, bien que possible, est donc potentiellement coûteux. On préfère se restreindre à des automates d'une forme plus simple, où les chemins sont uniques.

Automates déterministes. On se donne un ensemble de restrictions sur la forme des automates pour faciliter la reconnaissance d'un mot :

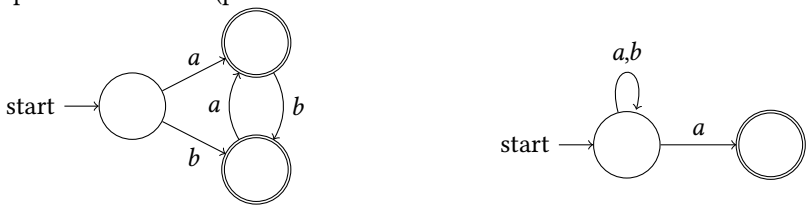
— un seul état initial,
— pas de transitions ε,
— partant de chaque état, au plus une transition par caractère.

Avec ces contraintes, pour un automate et un mot donnés, il y a au plus un chemin partant de l'état initial et correspondant à ce mot, que l'on peut construire ainsi :

1. partir de l'état initial,
2. pour chaque caractère a de l'entrée, suivre la transition correspondante si elle existe (sinon : échec),
3. une fois la lecture du mot terminée, celui-ci est reconnu si l'état d'arrivée est acceptant.

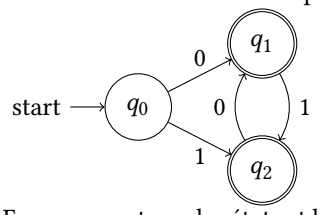
Ainsi, s'il existe bien un chemin pour un certain mot m et que cet unique chemin mène à un état acceptant, alors le mot m est reconnu. Sinon, c'est-à-dire s'il n'existe pas de chemin ou si l'unique chemin mène à un état qui n'est pas acceptant, alors le mot m n'est pas reconnu.

En reprenant ici deux exemples d'automates déjà présentés, celui de gauche est déterministe, mais pas celui de droite (partant de son état initial on trouve deux transitions avec l'étiquette a).



Représentation d'un automate déterministe par une table de transitions. On peut représenter les transitions d'un automate fini déterministe par un tableau à double entrée, avec une ligne par état et une colonne par caractère dans l'alphabet. S'il existe une transition $q \xrightarrow{a} q'$, on place q' dans la case croisant la ligne q et la colonne a.

Ainsi, les transitions de l'automate dessiné ci-dessous à gauche sont représentées par le tableau à droite. Les deux cases vides soulignent l'absence de transition étiquetée 0 depuis l'état q_1 et l'absence de transition étiquetée 1 depuis l'état q_2.



	0	1
q_0	q_1	q_2
q_1		q_2
q_2	q_1	

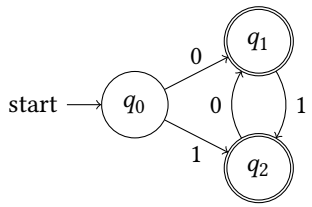
En supposant que les états et les caractères de l'alphabet sont numérotés, on peut représenter une telle table en caml à l'aide d'un tableau d'options.

```
type automate = {
  initial: int;
  transitions: (int option) array array;
  accepting: int list;
}

let a = {
  initial = 0;
  transitions = [| [| Some 1; Some 2 |];
                   [| None;   Some 2 |];
                   [| Some 1; None   |] |];
  accepting = [1; 2];
}
```

On peut alors écrire simplement une fonction de reconnaissance comme une boucle qui passe d'un état à l'autre en suivant les transitions. On suppose ici que les mots sont représentés par des listes d'entiers.

```
let reconnait (m: int list) (a: automate): bool =
  let rec loop m q = match m with
    | []    -> List.mem q a.accepting
    | b::m' -> match a.transitions.(q).(b) with
                 | None    -> false
                 | Some q' -> loop m' q'
  in
  loop m a.initial
```

Codage d'un automate par des fonctions récursives. Alternativement, on peut représenter un état d'un automate par une fonction prenant en entrée un mot et renvoyant **`true`** si le mot est reconnu depuis cet état et **`false`** sinon. Les fonctions représentant chaque état d'un automate donné forment alors un ensemble de fonctions mutuellement récursives. Si le mot reçu en entrée est vide, chacune de ces fonction renvoie directement **`true`** ou **`false`** selon qu'elle représente un état acceptant ou non. Sinon, on effectue la transition vers un nouvel état en faisant un appel récursif à la fonction correspondante.




```
let rec q0 = function
  | []   -> false
  | a::m -> if a=0 then q1 m else q2 m

and q1 = function
  | []   -> true
  | a::m -> if a=1 then q2 m else false

and q2 = function
```

```
    | []   -> true
    | a::m -> if a=0 then q1 l else false
```

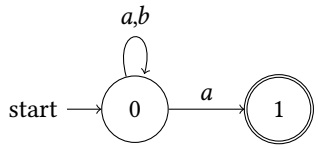
Déterminisation. Tout automate peut être transformé en un automate déterministe ***équivalent***, c'est-à-dire reconnaissant le même langage. Ce processus de transformation est appelé ***déterminisation***.

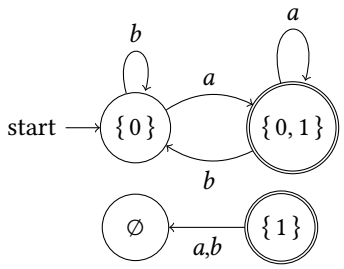
Lorsque l'on considère un automate non déterministe (Q, T, I, F), chaque mot d'entrée est susceptible de correspondre à plusieurs chemins. Ces chemins correspondentaux différents choix possible d'un état initial dans I, à l'emprunt d'éventuelles transitions ε, et aux choix faits entre plusieurs transitions depuis un état donné portant la même étiquette. L'idée directrice de la déterminisation consiste à suivre tous ces chemins à la fois. Plus précisément, il s'agit de suivre l'ensemble des états auxquels il est possible d'arriver après lecture d'un mot donné.

On construit pour cela un nouvel automate, dont chaque état est un sous-ensemble d'états de Q. Si $X \subseteq Q$ et $X' \subseteq Q$, on a une transition $X \xrightarrow{a} X'$ si et seulement si X' est l'ensemble des états atteignables à partir de X par des chemins dont le mot associé est a. C'est-à-dire que les états $q' \in Q$ sont exactement ceux pour lesquels il existe dans l'automate d'origine un chemin $q \xrightarrow{\varepsilon} \dots \xrightarrow{\varepsilon}\xrightarrow{a}\xrightarrow{\varepsilon} \dots \xrightarrow{\varepsilon} q'$ avec $q \in X$.

On prend comme unique état initial l'ensemble I lui-même, et on désigne comme acceptant tout ensemble ayant une intersection non vide avec F (autrement dit, l'ensemble des états acceptants est $\{ X \subseteq Q \mid X \cap F \neq \emptyset \}$).

Ainsi, l'automate non déterministe à gauche peut être déterminisé en l'automate ci-dessous à droite.





Remarquez dans l'automate déterminisé qu'il n'est pas possible d'atteindre les états $\emptyset$ et $\{ 1 \}$ depuis l'état initial. Ils sont inclus ici pour s'en tenir strictement à la définition.

À retenir de ce processus de déterminisation : il assure que tout langage pouvant être reconnu par un automate fini quelconque peut l'être en particulier par un automate fini déterministe. En revanche, le nombre d'états peut augmenter grandement : l'automate déterminisé peut contenir $2^{|Q|}$ états !

5.3 Approfondissement : théorème de Kleene

Les langages qui peuvent être décrits par une expression régulière sont exactement les langages qui peuvent être reconnus par un automate fini.

Ce théorème nous apprend que les expressions régulières et les automates finis sont deux facettes d'un même concept, l'une à vocation descriptive et l'autre à vocation algorithmique.

Nous avons déjà vu que de toute expression régulière on pouvait déduire un automate fini reconnaissant le même langage, par exemple en utilisant la construction de Thompson éventuellement suivie d'une déterminisation ou d'autres transformations. Pour obtenir le théorème de Kleene nous allons maintenant démontrer que la réciproque est vraie également.

Automate généralisé. Notre objectif est, partant d'un automate, de construire une expression régulière décrivant le même langage. On va le faire en transformant petit à petit notre automate de départ, en faisant progressivement apparaître des expressions régulières sur les transitions à la places des seuls caractères.

Un ***automate généralisé*** est un quadruplet (Q, T, I, F), où

- Q, I et F sont comme avant un ensemble d'états, et les sous-ensembles des états initiaux et des états acceptants,
- T est une relation de transition donnée par un ensemble de triplets $q \xrightarrow{e} q'$ d'un état de départ q, d'une expression régulière e et d'un état d'arrivée q'.

Un mot m sur un alphabet A est ***reconnu*** par un tel automate s'il existe un chemin $q_0 \xrightarrow{e_1} q_1 \xrightarrow{e_2} q_2 \cdots q_{k-1} \xrightarrow{e_k} q_k$ qui :

- part d'un des états initiaux $q_0 \in I$ et arrive à l'un des états acceptants $q_k \in F$, et

— est tel que m puisse être découpé en k morceaux $m_1 m_2 \dots m_k$ de telle sorte que pour chaque i, le mot m_i soit dans le langage de l'expression régulière e_i.

Nous n'allons *pas* utiliser ces automates en pratique, car la reconnaissance y est basée sur un découpage non déterministe. Nous allons plutôt voir un automate généralisé comme une *structure de données* auxiliaire dans laquelle stocker les expressions régulières partielles générées à partir de notre automate de départ.

Algorithme d'élimination des états. Soit (Q, T, I, F) un automate (déterministe ou non, avec ou sans transitions ε).

1. Créer l'automate généralisé $(Q \cup \{q_i, q_f\}, T', q_i, \{q_f\})$ avec

$$T' = T \cup \{q_i \xrightarrow{\varepsilon} q \mid q \in I\} \cup \{q \xrightarrow{\varepsilon} q_f \mid q \in F\}$$

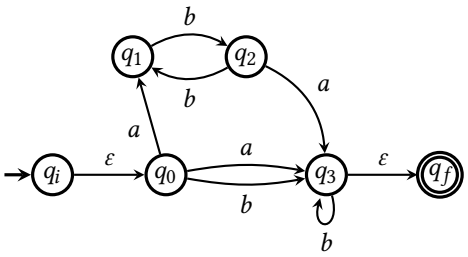
   Cet automate a un seul état initial q_i, sans transition entrante, et un seul état acceptant q_f, sans transition sortante.
2. Soit deux procédures auxiliaires :

   élimination des transitions : tant qu'il existe dans T deux transitions distinctes $q \xrightarrow{e_1} q'$ et $q \xrightarrow{e_2} q'$ entre les deux mêmes paires d'état, les supprimer et les remplacer par une unique transition $q \xrightarrow{e_1|e_2} q'$.

   élimination d'un état q : pour tous $p \xrightarrow{e_1} q$ et $q \xrightarrow{e_2} s$ dans T,
   - ajouter $p \xrightarrow{e_1 e^* e_2} s$ à T si $q \xrightarrow{e} q$ et y ajouter $p \xrightarrow{e_1 e_2} s$ sinon ;
   - supprimer $p \xrightarrow{e_1} q$ et $q \xrightarrow{e_2} s$ de T.

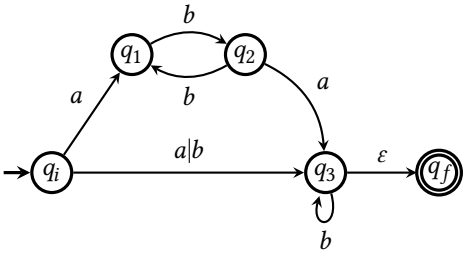
   Supprimer q de Q.
3. Boucle principale de l'algorithme. Pour chaque état $q \in Q$,
   - éliminer toutes les transitions possibles ;
   - éliminer l'état q.
4. Une fois l'automate réduit à $q_i \xrightarrow{e} q_f$, renvoyer e.

Exemple. Considérons l'automate A sur l'alphabet $\{a, b\}$, représenté par son graphe (complété avec q_i et q_f) :

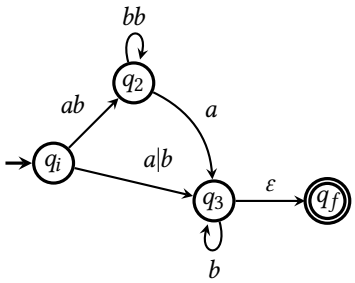


1. Élimination de q_0. On supprime toutes les transitions possibles, ici uniquement $q_0 \xrightarrow{a} q_3$ et $q_0 \xrightarrow{b} q_3$ remplacées par $q_0 \xrightarrow{a|b} q_3$.

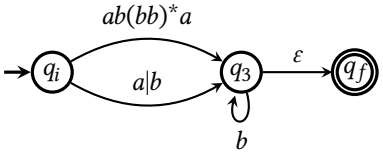
   On considère ensuite les paires $(q_i \xrightarrow{\varepsilon} q_0,\ q_0 \xrightarrow{a} q_1)$ et $(q_i \xrightarrow{\varepsilon} q_0,\ q_0 \xrightarrow{a|b} q_3)$. On crée $q_i \xrightarrow{a} q_1$ et $q_i \xrightarrow{a|b} q_3$ et on supprime q_0.



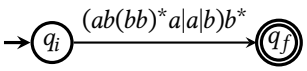
2. Élimination de q_1. Pas de transitions à éliminer. On considère $(q_i \xrightarrow{a} q_1,\ q_1 \xrightarrow{b} q_2)$ et $(q_2 \xrightarrow{b} q_1,\ q_1 \xrightarrow{b} q_2)$. On crée $q_i \xrightarrow{ab} q_2$ et $q_2 \xrightarrow{bb} q_2$ et on supprime q_1.



3. Élimination de q_2. Pas de transition à éliminer. On considère ($q_i \xrightarrow{ab} q_2$, $q_2 \xrightarrow{a} q_3$). On crée $q_i \xrightarrow{ab(bb)^*a} q_3$, en n'oubliant pas la boucle sur q_2 qui se transforme en $(bb)^*$.



4. Élimination de q_3. On élimine les transitions $q_i \xrightarrow{ab(bb)^*a} q_3$ et $q_i \xrightarrow{a|b} q_3$ que l'on remplace par $q_i \xrightarrow{ab(bb)^*a|a|b} q_3$. On considère cette transition et $q_3 \xrightarrow{\varepsilon} q_f$ pour obtenir $q_i \xrightarrow{(ab(bb)^*a|a|b)b^*\varepsilon} q_f$.
5. L'automate est réduit aux états q_i et q_f, l'expression recherchée est $(ab(bb)^*a|a|b)b^*$.



La taille des expressions régulières renvoyées par cet algorithme peut, dans le pire cas, être exponentielle en le nombre d'états de l'automate de départ (certains cas sont inévitables). Le choix de l'ordre des sommets peut grandement influencer la taille finale de l'expression. Dans l'exemple nous avons considéré dans l'ordre les sommets de q_0 à q_n, mais on pourrait choisir à chaque itération n'importe quel état autre que q_i et q_f. De nombreuses simplifications algébriques et heuristiques sont utilisées en pratique pour obtenir des expressions plus courtes et plus *lisibles*.

Cet algorithme est cependant suffisant pour achever la preuve du théorème de Kleene. Il suffit pour cela de justifier que les deux transformations préservent le langage de l'automate.

Élimination de transitions : pour tout mot m tel que $q \xrightarrow{m} q'$, s'il y a deux transitions $q \xrightarrow{e_1} q'$ et $q \xrightarrow{e_2} q'$ allant de q à q', alors soit $m \in L(e_1)$, soit $m \in L(e_2)$ et par définition $m \in L(e_1) \cup L(e_2) = L(e_1|e_2)$.

Élimination des états : soit $p \xrightarrow{e_1} q$, $q \xrightarrow{e_3} q$ et $q \xrightarrow{e_2} s$ trois transitions. Si $p \xrightarrow{m} s$ (en prenant ces transitions), c'est que $m = m_1 m_3 m_2$ avec $m_1 \in L(e_1)$, $m_2 \in L(e_2)$ et $m_3 \in (L(e_3))^* = L(e_3^*)$. Ainsi, $m \in L(e_1 e_3^* e_2)$.

Propriétés des langages reconnaissables. Les langages reconnaissables ont de bonnes propriétés.

— Par définition, l'union de langages reconnaissables L_1 et L_2 est un langage reconnaissable. Il en est de même de l'intersection. On peut montrer ce résultat en partant des deux automates (Q_i, T_i, I_i, F_i) et en construisant l'automate produit dont les états sont des couples d'états (s_1, s_2) et dont les transitions sont $(s_1, s_2) \xrightarrow{a} (s_1', s_2')$ si et seulement si $s_1 \xrightarrow{a} s_1'$ et $s_2 \xrightarrow{a} s_2'$. On laisse en exercice la précision des états initiaux et terminaux et la justification du fait que le langage reconnu est l'intersection $L_1 \cap L_2$.

— Le complémentaire d'un langage reconnaissable L est reconnaissable. Pour montrer cela, on part d'un automate (Q, T, i, F) déterministe et complet qui reconnaît le langage L et il suffit juste de modifier l'ensemble des états terminaux en prenant le complémentaire de l'ensemble F.

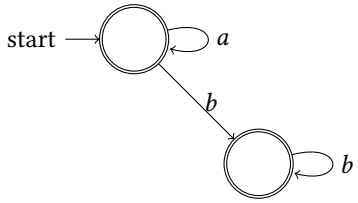
— Il est possible de décider si un langage reconnaissable est vide, il suffit de démontrer qu'il n'existe pas de chemin d'un état initial à un état final. L'automate étant un graphe fini, on peut calculer la composante connexe d'un état initial et vérifier s'il y a ou non des états terminaux dans cette composante.

5.4 Approfondissement : langages non reconnaissables

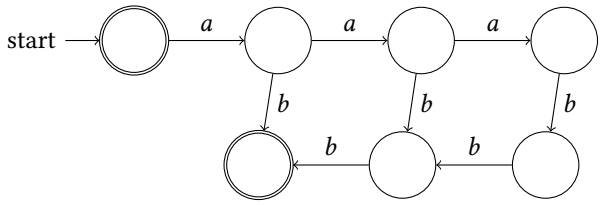
Les automates représentent des algorithmes travaillant avec une mémoire bornée. Nous allons voir ce que cela implique concernant les langages qui peuvent ou non être reconnus par un automate fini (déterministe ou non).

Considérons quelques langages sur l'alphabet $\{a, b\}$.

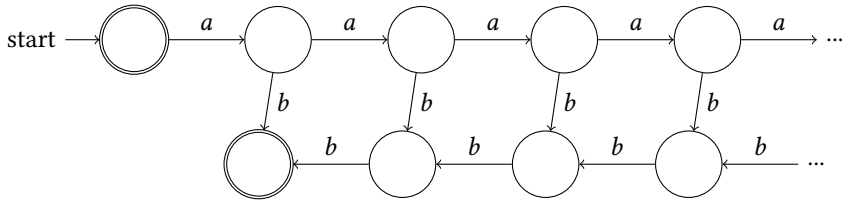
— $L_1 = \{ a^n b^m \mid n \in \mathbb{N} \wedge m \in \mathbb{N} \}$
Le langage L_1 est aussi décrit par l'expression régulière a^*b^* et est reconnu par l'automate



— $L_2 = \{ a^n b^n \mid n \leqslant 3 \}$
Le langage L_2 est aussi décrit par l'expression régulière $\varepsilon|ab|aabb|aaabbb$ et est reconnu par l'automate



— $L_3 = \{ a^n b^n \mid n \in \mathbb{N} \}$
La stratégie utilisée pour reconnaître le langage L_2 ne fonctionne pas ici : pour accepter de lire un nombre arbitrairement grand de a, il faudrait un automate s'étendant infiniment loin vers la droite.

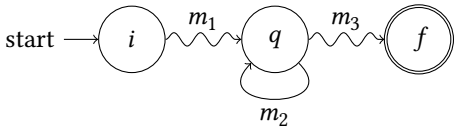


De manière générale, s'il faut des états différents pour distinguer chaque nombre possible de a lus (pour ensuite compter le bon nombre de b) alors on a besoin d'un nombre infini d'états, ce qui n'est pas permis avec un automate *fini*.

Conséquences de la finitude d'un automate. Revenons sur ce qui caractérise un automate fini.

— Le nombre fini d'états correspond à une mémoire bornée, impliquant que l'on ne peut pas tout retenir à propos du mot lu jusque là.
— Les règles de transition donnent une destination en fonction de l'état courant et d'un caractère lu (éventuellement plusieurs destinations en cas d'automate non déterministe).
— Revenir à un état déjà visité, c'est comme oublier tout ce qui a été lu depuis la première visite de cet état. C'est même ignorer le fait que cet état est vu une deuxième fois, ou une troisième, ou une quatrième...

Considérons donc un automate (Q, T, I, F) à N états, et intéressons-nous à un mot m de longueur $l \geqslant N$ reconnu par cet automate. Un chemin étiqueté par ce mot passe donc $l + 1$ états. Comme $l + 1 > N$, par le lemme des tiroirs il existe un état qui est vu au moins deux fois dans ce chemin. Autrement dit, le chemin étiqueté par m peut être schématisé ainsi, avec avec $m = m_1 m_2 m_3$.



L'automate ne distingue donc d'aucune manière les mots suivants :

— $m_1 m_3$
— $m_1 m_2 m_3$
— $m_1 m_2 m_2 m_3$
— $m_1 m_2 m_2 m_2 m_3$
— ...

Si l'un de ces mots est accepté, tous les autres le sont également, puisque lire le mot m_2 à partir de l'état q ramène à ce même état et donc ne change en rien à la suite du calcul de reconnaissance.

Lemmes de l'étoile. La remarque précédente peut être traduite en deux lemmes, appelés lemmes de l'étoile.

Version faible. *Soit L un langage reconnaissable. Il existe un entier N tel que tout mot $m \in L$ de longueur $l \geqslant N$ puisse être décomposé en $m_1 m_2 m_3$ avec $m_2 \neq \varepsilon$ et $L(m_1 m_2^* m_3) \subseteq L$.*

Version forte. *Soit L un langage reconnaissable. Il existe un entier N tel que pour tout mot $m_0 m m_4 \in L$ avec m de longueur $l \geqslant N$, le mot m peut être décomposé en $m_1 m_2 m_3$ avec $m_2 \neq \varepsilon$ et $L(m_0 m_1 m_2^* m_3 m_4) \subseteq L$.*

La preuve découle du paragraphe précédent, avec N le nombre d'états d'un automate reconnaissant le langage L.

Les lemmes de l'étoile indiquent que dans un langage reconnaissable, tout mot long contient une partie répétable à l'infini. La particularité de la version forte est qu'elle permet de maîtriser approximativement l'endroit où placer cette boucle. Ces lemmes peuvent être utilisés pour montrer que certains langages *ne sont pas reconnaissables.*

Un langage non reconnaissable. *Montrons que le langage $L_3 = \{ a^n b^n \mid n \in \mathbb{N} \}$ n'est pas reconnaissable. L'idée est, en raisonnant par l'absurde, de supposer que L_3 soit reconnaissable et d'utiliser le lemme de l'étoile pour déduire qu'un certain mot $a^{k_1} b^{k_2}$ avec $k_1 \neq k_2$ appartienne aussi nécessairement à L_3.*

Supposons que L_3 soit reconnaissable. Par le lemme de l'étoile fort, il existe un entier $N \in \mathbb{N}$ tel que pour tout mot $m_0 m m_4 \in L_3$ avec m de longueur $l \geqslant N$, le mot m peut être décomposé en $m_1 m_2 m_3$ avec $m_2 \neq \varepsilon$ et $L(m_0 m_1 m_2^* m_3 m_4) \subseteq L_3$.

Considérons alors le mot $a^N b^N$. Il peut être décomposé en $a^N b^N = m_0 m m_4$ avec $m_0 = \varepsilon$, $m = a^N$ et $m_4 = b^N$. Le mot $m = a^N$ a la longueur N (qui est bien $\geqslant N$), on peut donc le décomposer en trois parties $m_1 m_2 m_3$, avec $m_2 \neq \varepsilon$ et $L(m_1 m_2^* m_3 b^N) \subseteq L_3$. Autrement dit, on peut décomposer a^N en $a^{n_1} a^{n_2} a^{n_3}$ avec $n_1 + n_2 + n_3 = N$, $n_2 \neq 0$ et pour tout $k \in \mathbb{N}$, $a^{n_1} a^{k n_2} a^{n_3} b^N \in L_3$, c'est-à-dire $a^{n_1 + k n_2 + n_3} b^N \in L_3$.

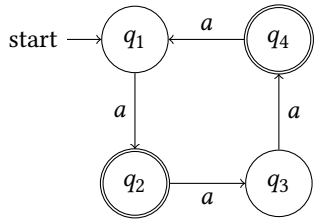
En particulier, en prenant $k = 0$ on aurait $a^{n_1} a^{n_3} b^N \in L_3$, c'est-à-dire $a^{N - n_2} b^N \in L_3$. Or $N - n_2 \neq N$: contradiction. Donc L_3 n'est pas reconnaissable par un automate fini.

5.5 Approfondissement : minimisation

On cherche à construire un automate le plus petit possible pour un langage reconnaisable L donné.

Précision : on veut un automate déterministe et ***complet***, c'est-à-dire pour lequel depuis chaque état il existe exactement une transition pour chaque caractère de l'alphabet. Pour tout état q et tout caractère a on pourra donc noter $q.a$ l'unique état atteint en faisant une transition a depuis q. On étend même cette notation à $q.m$: la lecture de n'importe quel mot m à partir de l'état q.

Équivalence de Nérode. Deux états q et q' d'un automate sont ***équivalents*** si les mêmes mots permettent d'aller de q ou q' à un état acceptant.



Mots qui permettent d'aller à un état acceptant :
— depuis q_1 : *a*, *aaa*, *aaaaa*, ...
— depuis q_2 : ε, *aa*, *aaaa*, ...

— depuis q_3 : a, aaa, $aaaaa$, ...
— depuis q_4 : ε, aa, $aaaa$, ...

L'état q_1 est donc équivalent à l'état q_3 : ils sont caractérisés par les mots $a(aa)^*$. De même, l'état q_2 est équivalent à l'état q_4, caractérisés par les mots $(aa)^*$.

Formellement, étant donné un automate fini déterministe complet (Q, T, i, F) on définit le langage $L(q)$ d'un état q comme l'ensemble des mots étiquetant un chemin de q à un état acceptant.

$$L(q) \quad = \quad \{m \mid q.m \in F\}$$

On définit alors l'équivalence par

$$q_1 \sim q_2 \quad \text{ssi} \quad L(q_1) = L(q_2)$$

Cette relation d'équivalence est compatible avec la fonction de transition : si $q_1 \sim q_2$ alors pour tout caractère a on a encore $q_1.a \sim q_2.a$.

Preuve. Supposons $q_1 \sim q_2$. Soit a un caractère et $m \in L(q_1.a)$. Par définition, $(q_1.a).m \in F$. Donc $q_1.(am) \in F$, c'est-à-dire $am \in L(q_1)$. Par équivalence entre q_1 et q_2 on a donc $am \in L(q_2)$, c'est-à-dire $q_2.(am) \in F$. On en déduit finalement $(q_2.a).m \in F$, autrement dit $m \in L(q_2.a)$. Donc $q_1.a \sim q_2.a$.

Automate quotient. Partant d'un automate fini (Q, T, i, F) déterministe et complet, on obtient un ***automate quotient*** en fusionnant les états équivalents.

Pour tout état $q \in Q$, notons $[q]$ la classe d'équivalence de q. L'automate quotient $(Q_\sim, T_\sim, i_\sim, F_\sim)$ est un automate fini déterministe complet défini par les éléments suivants :
— les états sont les classes d'équivalence : $Q_\sim = \{[q] \mid q \in Q\}$,
— la fonction de transition est donnée par : $[q].a = [q.a]$ (ce qui est bien défini grâce à la propriété de compatibilité),
— l'état initial est la classe de i : $i_\sim = [i]$,
— les états acceptants sont les classes des états acceptants : $F_\sim = \{[f] \mid f \in F\}$.

Le point clé pour définir concrètement cet automate quotient est la caractérisation de la relation d'équivalence de Nérode $\sim$. On calcule cette relation à l'aide d'approximations successives $(\sim_n)_{n\in\mathbb{N}}$, où $\sim_n$ caractérise les états qui sont équivalents vis-à-vis des mots de longueur $\leqslant n$. Autrement dit, en notant $A^{\leqslant n}$ l'ensemble de mots de longueur $\leqslant n$ sur l'alphabet A :

$$q_1 \sim_n q_2 \quad \text{ssi} \quad L(q_1) \cap A^{\leqslant n} = L(q_2) \cap A^{\leqslant n}$$

Ces approximations successives sont calculées par récurrence sur n :
— $q_1 \sim_0 q_2$ ssi $q_1 \in F \wedge q_2 \in F$ ou $q_1 \notin F \wedge q_2 \notin F$,
— $q_1 \sim_{n+1} q_2$ ssi $q_1 \sim_n q_2$ et $\forall a,\ q_1.a \sim_n q_2.a$.

On arrête ce processus dès lors que l'on trouve un N tel que $\sim_N = \sim_{N+1}$. En effet, puisque chaque approximation successive est calculée uniquement en fonction de la précédente, on a alors $\sim_N = \sim_{N+1} = \sim_{N+2} = \sim_{N+3} = \ldots$

Reste à démontrer qu'un tel N existe bien. Remarquons d'abord que $\sim_0$ définit uniquement deux classes d'équivalence : F et $Q \setminus F$. Lors du passage d'un $\sim_n$ à un $\sim_{n+1}$ chaque classe d'équivalence de niveau $n+1$ est égale à ou incluse dans une classe d'équivalence de niveau n. Si $\sim_n \neq \sim_{n+1}$ alors au moins une classe d'équivalence de $\sim_n$ est donc scindée en plusieurs classes dans $\sim_{n+1}$, et $\sim_{n+1}$ contient strictement plus de classes d'équivalence que $\sim_n$. Or la relation d'équivalence de Nérode réelle $\sim$ contient au plus une classe d'équivalence par état de Q. Donc après $|Q|$ étapes (ou même $|Q| - 1$ étapes) au maximum il ne sera plus possible de scinder à nouveau des classes d'équivalence et on arrivera bien au point fixe.

Langages résiduels et minimalité de l'automate quotient. Partant d'un automate fini (Q, T, i, F) déterministe complet reconnaissant un langage L, la construction de l'automate quotient donne un moyen algorithmique de construire un automate plus petit reconnaissant le même langage. Nous allons maintenant démontrer que cet automate quotient est bien *le plus petit* automate déterministe complet reconnaissant ce langage L.

Étant donnés un langage L sur un alphabet A et un mot u sur A, le ***langage résiduel*** (ou simplement ***résidu***) de L après u est l'ensemble des mots complétant u en un mot de L :

$$u^{-1}L \quad = \quad \{v \mid uv \in L\}$$

Par exemple, en posant $L = \{\varepsilon, ab, (ab)^2, (ab)^3\}$ et $u = aba$ on a $u^{-1}L = \{b, bab\}$.

Si le langage L est reconnu par un automate déterministe complet (Q, T, i, F), alors la définition du langage résiduel est équivalente à

$$u^{-1}L \quad = \quad L(i.u)$$

Ainsi chaque langage résiduel est le langage de l'un des états de l'automate. Par conséquent, le nombre de langages résiduels de L est nécessairement inférieur ou égal au nombre d'états de tout automate déterministe complet reconnaissant L. Autrement dit, le nombre de langages résiduels de L donne une borne inférieure sur le nombre d'état d'un automate déterministe complet reconnaissant L.

Question en passant : qu'en déduire d'un langage L qui admettrait une infinité de résiduels différents ?

On peut même caractériser à l'aide des langages résiduels un automate déterministe complet de taille minimale, appelé ***automate des résiduels***. Prenons un langage reconnaissable L, doté d'un ensemble fini $\{R_1, \ldots, R_k\}$ de résiduels. Définition de l'automate des résiduels :

- les états sont les résiduels R_1, ..., R_k,
- la fonction de transition est définie par $R.a = a^{-1}R$,
- l'état initial est $\varepsilon^{-1}L$, c'est-à-dire L,
- les états acceptants sont les R_i tels que $\varepsilon \in R_i$.

Il ne reste donc qu'à justifier que, quel que soit l'automate (Q, T, i, F) déterministe complet reconnaissant le langage L, son quotient est précisément l'automate des résiduels de L. Pour cela, remarquons qu'il y a correspondance entre une classe d'équivalence $[q]$ et un langage résiduel $L(q)$, et que cette correspondance est compatible avec les fonctions de transition de l'automate quotient et de l'automate des résiduels.

Pour une classe d'équivalence $[q]$ et un caractère a on a dans l'automate quotient une transition

$$[q].a \quad = \quad [q.a]$$

Or, partant du résiduel correspondant $L(q)$ on peut calculer ainsi la transition équivalente dans l'automate des résiduels

$$\begin{array}{rcl} L(q).a & = & a^{-1}L(q) \\ & = & \{m \mid am \in L(q)\} \\ & = & \{m \mid q.am \in F\} \\ & = & \{m \mid (q.a).m \in F\} \\ & = & L(q.a) \end{array}$$

Finalement, pour tout langage reconnaissable L il existe un automate fini déterministe complet de taille minimale, qui est unique modulo renommage des états. Cet automate minimal peut être obtenu par des fusions d'états à partir de n'importe quel automate déterministe complet reconnaissant L. Faire ce calcul est un moyen algorithmique de tester l'égalité entre deux langages.

5.6 Approfondissement : de l'automate à l'analyseur lexical

Les automates finis, dont nous avons étudié la théorie, donnent une contrepartie algorithmique aux expressions régulières. Nous allons maintenant utiliser ces concepts pour construire un analyseur lexical, c'est-à-dire un programme extrayant d'une chaîne de caractères représentant le code source d'un programme la séquence des mots qui le constituent.

Un analyseur lexical est essentiellement un automate fini qui reconnaît la réunion de toutes les expressions régulières définissant les mots, ou *lexèmes*, d'un langage de programmation. Nous allons cependant voir petit à petit qu'un certain nombre de subtilités viennent s'ajouter à cette théorie. Pour illustrer toutes ces petites différences, nous allons partir du code simple de reconnaissance d'un mot par un automate et le transformer petit à petit.

Définition d'un automate et d'une fonction de reconnaissance. On se donne un type caml pour représenter un automate fini déterministe `a` dont les états sont numérotés. L'état initial `a.initial` est donné par son numéro, et le statut acceptant ou non de chaque état est consigné dans un tableau de booléens `a.accepte` tel que `a.accepte.(etat)` vaut **`true`** si l'état `etat` de l'automate `a` est acceptant. La table à double entrée des transitions est réalisée par un tableau qui à chaque état source associe une liste de paires associant un caractère lu à l'état cible.

```
type automate = {
  initial: int;
  trans:   (char * int) list array;
  accepte: bool array;
}
```

On définit une fonction de transition qui, prenant un état de départ et un caractère lu, renvoie l'état cible. On ajoute un état puits avec le numéro `-1`, qui est cible de toute transition inexistante.

```
let transition autom etat car =
  try List.assoc car autom.trans.(etat) with Not_found -> -1
```

On en déduit la fonction de reconnaissance suivante, qui renvoie **`true`** si un préfixe de l'entrée est reconnu, et **`false`** sinon. Cette fonction prend le mot d'entrée sous la forme d'une chaîne de caractères. Elle est construite à l'aide d'une boucle principale `scan`, qui effectue la reconnaissance à partir d'une position courante `pos` dans l'entrée et d'un état courant `etat` dans l'automate. Cette fonction `scan` commence par calculer l'état cible `etat'` obtenu après lecture du caractere courant `entree.[pos]`. On arrête l'analyse s'il n'y a pas d'état suivant possible (réponse **`false`**), ou si l'état suivant est acceptant (réponse **`true`**). Dans les autres cas l'analyse continue à partir du nouvel état et de la position suivante.

```
let analyseur (autom: automate) (entree: string): bool =
  let n = String.length entree in
  let rec scan (pos: int) (etat: int): bool =
    let etat' =
      if pos = n then -1
      else transition autom etat entree.[pos]
    in
    if etat' >= 0 then
      if autom.accepte.(etat') then true
      else scan (pos+1) etat'
    else false
  in
  scan 0 autom.initial
```

Notez qu'une telle fonction d'analyse ne peut pas reconnaître le mot vide, mais c'est tout à fait adapté dans notre cas.

Voici comment définir directement dans cette représentation un automate reconnaissant :

— les symboles arithmétiques + et * et les parenthèses,
— les nombres binaires positifs ou négatifs,
— les séquences d'espaces,
— les commentaires délimités par `(*` et `*)`, sans imbrication.

```
let a = {
  initial = 0;
  trans = [|
      (* 0 *) [('+', 1); ('*', 2); ('0', 3); ('1', 3);
                ('-', 4); ('(', 5); (')', 9); (' ', 10); ];
      (* 1 *) [];
      (* 2 *) [];
      (* 3 *) [('0', 3); ('1', 3)];
      (* 4 *) [('0', 3); ('1', 3)];
      (* 5 *) [('*', 6)];
      (* 6 *) [('+', 6); ('*', 7); ('0', 6); ('1', 6);
                ('-', 6); ('(', 6); (' ', 6); (')', 6); ];
      (* 7 *) [('+', 6); ('*', 7); ('0', 6); ('1', 6);
                ('-', 6); ('(', 6); (' ', 6); (')', 8); ];
      (* 8 *) [];
      (* 9 *) [];
      (* 10 *) [(' ', 10)]
    |];
  accepte = [| false; true; true; true; false; true;
               false; false; true; true; true |]
}
```

Test dans la boucle d'interaction caml :

```
# analyseur a "(1␣+␣10(*␣11␣*␣1␣*))*␣-101";;
- : bool = true
```

L'analyse lexicale diffère cependant de la simple reconnaissance d'un mot par un automate car :
- il ne faut pas seulement dire qu'un mot a été reconnu mais identifier le lexème,
- il faut décomposer l'entrée en tous les mots qui la composent et tout ça en gérant de possibles ambiguïtés dans l'identification des lexèmes.

Nous allons raffiner notre fonction de reconnaissance pour obtenir ces nouveaux effets.

Renvoyer le lexème reconnu. Pour identifier quel lexème est reconnu, nous allons déjà devoir ajouter des informations aux états acceptants de l'automate.

Le tableau de booléens `accepte` laisse donc la place à un nouveau tableau `mots`, qui pour chaque état indique soit `Some m` avec `m` un lexème si l'état est acceptant et reconnaît le lexème `m`, soit `None` si l'état n'est pas acceptant. Le type `'mot` des lexèmes est laissé libre, il sera défini conjointement à l'automate lui-même.

```
type 'mot automate = {
  initial: int;
  trans:   (char * int) list array;
  mots:    'mot option array;
}
```

On adapte également la fonction d'analyse lexicale pour qu'elle renvoie l'identification du lexème reconnu plutôt qu'un simple booléen. Cette fonction déclenchera en revanche une exception si aucun lexème n'est reconnu.

```
let analyseur (autom: 'mot automate) (entree: string): 'mot =
  let n = String.length entree in
  let rec scan (pos: int) (etat: int): 'mot =
    let etat' =
      if pos = n then -1
      else transition autom etat entree.[pos]
    in
    if etat' >= 0 then
      match autom.mots.(etat') with
        | None -> scan (pos+1) etat'
        | Some(mot) -> mot
    else failwith "échec"
  in
  scan 0 autom.initial
```

Pour adapter notre automate exemple, on commence par définir un type pour représenter nos lexèmes en caml.

```
type mot = NOMBRE | PLUS | FOIS | LPAR | RPAR | BLANC
```

On inclut alors dans la définition de l'automate a le tableau de mots suivants.

```
let a = {
  initial = 0;
  trans = ...;
  mots = [| None; Some PLUS; Some FOIS; Some NOMBRE; None; Some LPAR;
            None; None; Some BLANC; Some RPAR; Some BLANC |]
}
```

Test :

```
# analyseur a "(1␣+␣10(*␣11␣*␣1␣*))*␣-101";;
- : mot = LPAR
```

Fonction de reconnaissance avec point de départ variable. Voici une nouvelle adaptation permettant d'appeler une fonction de reconnaissance plusieurs fois, à plusieurs positions de l'entrée. Ainsi, au lieu d'analyser le premier lexème de l'entrée, un appel `analyseur autom entree` renvoie une fonction de type `int -> 'mot * int`, qui prend en paramètre une position de départ et renvoie
- la nature du lexème reconnu à partir de cette position, et

— la position atteinte après la reconnaissance du lexème.

On se donne ainsi la possibilité d'appeler à nouveau la fonction, en recommançant après les mots qui ont déjà été reconnus.

```
let analyseur autom entree =
  let n = String.length entree in
  let rec scan (pos: int) (etat: int) =
    let etat' =
      if pos = n then -1
      else transition autom etat entree.[pos]
    in
    if etat' >= 0 then
      match autom.mots.(etat') with
        | None -> scan (pos+1) etat'
        | Some(mot) -> mot, pos+1
    else failwith "échec"
  in
  fun depart -> scan depart autom.initial
```

Test enchaînés :

```
# let prochain_mot = analyseur a "(1␣+␣10(*␣11␣*␣1␣*))*␣-101";;
# prochain_mot 0;;
- : mot * int = (LPAR, 1)
# prochain_mot 5;;
- : mot * int = (NOMBRE, 6)
# prochain_mot 7;;
- : mot * int = (LPAR, 8)
```

Renvoyer la chaîne en plus de l'identification du lexème. Certains lexèmes, comme `NOMBRE` ici, sont censés être accompagnés d'un contenu. On va donc renvoyer la chaîne reconnue à côté de l'identification du lexème et de la position atteinte. Pour pouvoir identifier cette chaîne, on mémorise la position de départ de la reconnaissance sous le nom `depart`, et on fait de cette information un nouveau paramètre de la fonction `scan`.

```
let analyseur autom entree =
  let n = String.length entree in
  let rec scan (depart: int) (pos: int) (etat: int) =
    let etat' =
      if pos = n then -1
      else transition autom etat entree.[pos]
    in
    if etat' >= 0 then
      match autom.mots.(etat') with
        | None -> scan depart (pos+1) etat'
        | Some(mot) ->
           mot, String.sub entree depart (pos+1-depart), pos+1
    else failwith "échec"
  in
  fun depart -> scan depart depart autom.initial
```

Tests :

```
# let prochain_mot = analyseur a "(1␣+␣10(*␣11␣*␣1␣*))*␣-101";;
# prochain_mot 0;;
- : mot * int = (LPAR, "(", 1)
# prochain_mot 5;;
- : mot * int = (NOMBRE, "1", 6)
# prochain_mot 7;;
- : mot * int = (LPAR, "(", 8)
```

Analyseur avec reprise automatique. Pour que l'analyse reprenne à chaque nouvel appel à la position qui avait été atteinte après lecture du dernier lexème, sans besoin de passer à la main le nouvel indice, on ajoute un état mutable à notre fonction. On introduit pour cela une référence `depart` donnant la position à laquelle l'analyse doit commencer, et on met cette référence à jour à la fin de chaque appel.

La fonction renvoyée par l'appel `analyseur autom entree` ne prend donc plus de position de départ en paramètre, et ne renvoie plus de position d'arrivée. Son type devient simplement `unit -> mot * string`.

```
let analyseur autom entree =
  let n = String.length entree in
  let depart = ref 0 in
  let rec scan (pos: int) (etat: int) =
    let etat' =
      if pos = n then -1
      else transition autom etat entree.[pos]
    in
    if etat' >= 0 then
      match autom.mots.(etat') with
        | None -> scan (pos+1) etat'
        | Some(mot) ->
          let s = String.sub entree !depart (pos+1 - !depart) in
          depart := pos+1;
          mot, s
    else failwith "échec"
  in
  fun () -> scan !depart autom.initial
```

Tests :

```
# let prochain_mot = analyseur a "(1␣+␣10(*␣11␣*␣1␣*))*␣-101";;
# prochain_mot ();;
- : mot * string = (LPAR, "(")
# prochain_mot ();;
- : mot * string = (NOMBRE, "1")
# prochain_mot ();;
- : mot * string = (BLANC, "␣")
```

Reconnaissance du mot le plus long. Un analyseur lexical n'est pas censé s'arrêter dès qu'il trouve un mot reconnaissable dans l'entrée. Une telle stratégie conduit à des aberrations comme le test

```
# let prochain_mot = analyseur a "(1␣+␣10(*␣11␣*␣1␣*))*␣-101";;
# prochain_mot 5;;
- : mot * int = (NOMBRE, "1", 6)
```

vu plus haut, qui dans son analyse d'une séquence commençant par "10" reconnaît un nombre formé du seul premier caractère (puisqu'un chiffre seul suffit à définir un nombre !). On s'attendrait dans ce cas précis à ce que l'intégralité du nombre soit prise en compte, et on souhaiterait donc plutôt l'issue suivante :

```
# prochain_mot 5;;
- : mot * int = (NOMBRE, "10", 7)
```

La règle pour un analyseur lexical consiste à toujours reconnaître la plus longue séquence reconnaissable. Ainsi, en présence d'une chaîne `"1234"` on reconnaîtra bien le nombre `1234` et non le simple chiffre `1`, et en présence d'une chaîne `funambule` on reconnaîtra toute cette chaîne (un identifiant) plutôt que la simple chaîne `f` (un autre identifiant) ou encore la chaîne intermédiaire **`fun`** (qui serait un mot-clé en caml).

Pour réaliser cette reconnaissance de la chaîne la plus longue, on change de stratégie dans le parcours de l'automate. Plutôt que de s'arrêter au premier état final rencontré, on continue la lecture jusqu'au premier blocage (manifesté dans notre convention par la rencontre de l'état "puits" de numéro `-1`).

Le blocage signifiant que plus aucune complétion du mot parcouru jusqu'ici ne pourra être reconnue, notre chaîne reconnaissable la plus longue a déjà été dépassée. On a alors deux situations possibles :

— Si aucun état acceptant n'avait été rencontré avant le blocage, alors il n'y a aucun motif reconnaissable : échec de l'analyse.

— Sinon, on va revenir au dernier état acceptant rencontré, et à la position correspondante de l'entrée. Ainsi, en notant

— s_1 le fragment de chaîne reconnu par le dernier état acceptant, et
— s_2 le fragment de chaîne lu entre le dernier état acceptant et le blocage

on reconnaît le lexème s_1 et on reprend l'analyse du prochain lexème au début de s_2.

Adaptation de l'analyseur pour reconnaître les mots les plus longs. La fonction `scan` prend en paramètre supplémentaire une option sur une paire d'un mot (identification du dernier lexème reconnu) et d'une position (position atteinte après reconnaissance de ce dernier lexème). Cette option vaut `None` tant que l'on n'a pas encore rencontré d'état acceptant.

```
let analyseur (autom: 'mot automate) (entree: string) =
  let n = String.length entree in
  let depart = ref 0 in
  let rec scan (pos: int) (etat: int)
               (dernier_mot: ('mot * int) option) =
    let etat' =
      if pos = n then -1
      else transition autom etat entree.[pos]
    in
    if etat' >= 0 then
      (* Tant que l'état visité n'est pas l'état puits, on poursuit
         l'analyse. En revanche, si l'état est acceptant on met à
         jour l'information [dernier_mot]. *)
      let dernier_mot = match autom.mots.(etat') with
        | None -> dernier_mot
        | Some(mot) -> Some(mot, pos+1)
      in
      scan (pos+1) etat' dernier_mot
    else match dernier_mot with
      (* Aboutir à l'état puits ou à la fin de la chaine n'est plus
         nécessairement un échec : c'est à ce moment que l'on
         consulte le dernier état acceptant rencontré et que l'on
         peut renvoyer un lexème (et mettre à jour la position de
         départ pour la prochaine analyse). *)
      | None -> failwith "échec"
      | Some(mot, pos) ->
        let s = String.sub entree !depart (pos - !depart) in
        depart := pos;
        mot, s
  in
  fun () -> scan !depart autom.initial None
```

Tests :

```
# let prochain_mot = analyseur a "10(*␣11␣*␣1␣*)*␣-101";;
# prochain_mot();;
- : mot * string = (NOMBRE, "10")
# prochain_mot();;
- : mot * string = (BLANC, "(*␣11␣*␣1␣*)")
# prochain_mot();;
- : mot * string = (FOIS, "*")
```

Au-delà des lexèmes : des actions. En pratique, le champ d'action d'un analyseur lexical peut aller au-delà de la simple identification d'un lexème, et la reconnaissance d'un mot peut entraîner des actions arbitrairement riches.

On pourrait ainsi avoir une version à nouveau étendue de la définition d'un automate, où le tableau des mots reconnus laisserait à son tour la place à un tableau d'actions à effectuer lorsqu'un mot est reconnu. Une action est représentée dans ce tableau par une fonction caml à appeler lorsqu'un mot est reconnu. Dans la version ci-dessous, cette fonction prend en paramètre le mot reconnu lui-même.

On donne un nouveau type reflétant ce nouvel enrichissement de l'automate, avec des actions produisant un résultat de type `'res` laissé libre (il sera défini conjointement à l'automate lui-même).

```
  type 'res automate = {
    initial: int;
    trans:   (char * int) list array;
    actions: (string -> 'res) option array;
  }
```

Principale modification de l'analyseur par rapport à la version précédente : au lieu de renvoyer l'identification du lexème et la chaîne reconnue, on renvoie le résultat de l'application de l'action à la chaîne reconnue.

```
let analyseur (autom: 'res automate) (entree: string) =
  let n = String.length entree in
  let depart = ref 0 in
  let rec scan (pos: int) (etat: int)
               (derniere_action: ((string -> 'res) * int) option) =
    let etat' =
      if pos = n then -1
      else transition autom etat entree.[pos]
    in
    if etat' >= 0 then
      let derniere_action = match autom.actions.(etat') with
        | None -> derniere_action
        | Some(a) -> Some(a, pos+1)
      in
      scan (pos+1) etat' derniere_action
    else match derniere_action with
      | None -> failwith "échec"
      | Some(a, pos) ->
         let s = String.sub entree !depart (pos - !depart) in
         depart := pos;
         a s
  in
  fun () -> scan !depart autom.initial None
```

Le format des actions donne plus de souplesse dans la représentation des lexèmes. Plutôt que d'avoir systématiquement une paire d'une étiquette d'identification et de la chaîne lue, on peut utiliser pour notre exemple un type caml un peu plus intégré comme le suivant.

```
type mot =
  NOMBRE of int | PLUS | FOIS | LPAR | RPAR | BLANC of string
```

On conserve dans tous les cas une identification du lexème reconnu, mais on n'inclut la chaîne que lorsqu'elle est utile, éventuellement après application de quelques conversions pour la présenter sous le format le plus utile, comme ici pour `NOMBRE`.

```
  let a = {
    initial = 0;
    trans = ...;
    actions = [|
        (* 0 *) None;
        (* 1 *) Some (fun _ -> PLUS);
        (* 2 *) Some (fun _ -> FOIS);
        (* 3 *) Some (fun s -> NOMBRE(int_of_string s));
        (* 4 *) None;
        (* 5 *) Some (fun _ -> LPAR);
        (* 6 *) None;
        (* 7 *) None;
        (* 8 *) Some (fun s -> BLANC s);
        (* 9 *) Some (fun _ -> RPAR);
        (* 10 *) Some (fun s -> BLANC s)
      |]
  }
```

Tests :

```
# let prochain_mot = analyseur a "(1␣+␣10(*␣11␣*␣1␣*))*␣-101";;
# prochain_mot();;
- : mot = LPAR
# prochain_mot();;
- : mot = NOMBRE 1
# prochain_mot();;
- : mot = BLANC "␣"
```

Notez que, l'action effectuée étant une fonction arbitraire, elle peut également produire divers effets de bord. C'est d'ailleurs une possibilité que nous utiliserons naturellement et abondamment en pratique, par l'intermédiaire des outils de la famille lex (chapitre 7).

6 Analyse syntaxique : analyse ascendante

6.1 Analyse ascendante avec automate et pile

L'analyse descendante formait l'arbre de dérivation en partant de la racine, pour essayer de produire une phrase correspondant à la séquence d'entrée. L'***analyse ascendante***, ou ***bottom-up***, travaille à l'inverse à partir des feuilles de l'arbre. Il s'agit de considérer chaque lexème de la séquence d'entrée comme une feuille, et d'opérer des regroupements une fois que l'on a lu un fragment correspondant à une règle. L'analyse est réussie si l'intégralité de l'entrée a pu être regroupée en un seul arbre, dont la racine est étiquetée par le symbole de départ.

On suit donc les principes suivants.

— Lecture de l'entrée un mot à la fois, de gauche à droite.

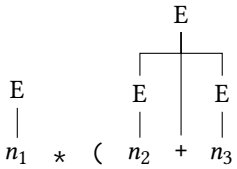
— Analyse de la phrase de bas en haut : les mots lus sont interprétés comme des arbres de dérivation triviaux, et sont regroupés à mesure que la lecture découvre des groupes entiers.

Sur la phrase n_1 * (n_2 + n_3) et avec la grammaire naïve des expressions arithmétiques, voici les regroupements qui peuvent être faits en fonction des différents niveaux d'avancement de la lecture.

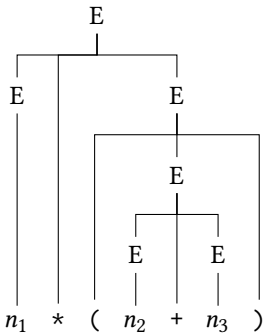
— Après lecture jusqu'à n_1 on identifie une première expression.

— Après lecture jusqu'à n_2 on a identifié une séquence comportant une expression, les symboles * et (et une deuxième expression.

— Après lecture jusqu'à n_3 on a à nouveau une séquence comportant une expression, les symboles * et (et une deuxième expression, mais la deuxième sous-expression dénote un fragment plus grand de l'entrée.



— Après lecture jusqu'à) on peut finalement regrouper tous les éléments lus en une unique expression.



En décomposant ce processus d'analyse, on isole une structure de données :

— une ***pile*** contenant les différents fragments lus ou reconstruits

et deux opérations élémentaires, entre lesquelles on va alterner :

— ***progression*** : placer le prochain mot sur la pile, (aussi appelé ***décalage***, ou en anglais ***shift***)

— ***réduction*** : identifier au sommet de la pile une séquence β apparaissant dans une règle $X \prec \beta$ de la grammaire, la retirer de la pile et la remplacer par X (en anglais ***reduce***).

On peut ainsi détailler l'analyse précédente par un tableau comme le suivant où apparaissent à chaque étape le contenu de la pile (une séquence de $(T \cup N)^*$), le fragment de phrase restant à analyser, et l'opération effectuée.

Pile	**Reste**	**Action**
$\emptyset$	$n_1 * (n_2 + n_3)$	progression
n_1	$* (n_2 + n_3)$	réd. $E \prec n$
E	$* (n_2 + n_3)$	progression
$E *$	$(n_2 + n_3)$	progression
$E * ($	$n_2 + n_3)$	progression
$E * (n_2$	$+ n_3)$	réd. $E \prec n$
$E * (E$	$+ n_3)$	progression
$E * (E +$	$n_3)$	progression
$E * (E + n_3$	$)$	réd. $E \prec n$
$E * (E + E$	$)$	réd. $E \prec E+E$
$E * (E$	$)$	progression
$E * (E)$	$\emptyset$	réd. $E \prec (E)$
$E * E$	$\emptyset$	réd. $E \prec E*E$
E	$\emptyset$	succès

Reste à définir des critères pour choisir efficacement et intelligemment quelle opération effectuer et quand.

Algorithme efficace. À chaque étape de l'analyse d'une phrase, le choix de l'opération est fait en fonction du contenu de la pile, et du prochain mot. Pour éviter un travail d'analyse lourd à chaque nouvelle opération, on précalcule une table indiquant l'opération à effectuer en fonction de la forme de la pile et du prochain mot. Cette table est construite une fois pour toutes à partir des règles de la grammaire, et peut ensuite être utilisée à chaque nouvelle phrase analysée.

On peut par exemple fixer dans un premier temps les décisions suivantes pour l'analyse des expressions arithmétiques :

— avec l'une des séquences n, $($ E $)$ ou E $*$ E au sommet de la pile, on réduit,
— avec la séquence E $+$ E au sommet de la pile on réduit également, sauf dans le cas où le prochain symbole est $*$ (pour gérer la priorité usuelle de la multiplication sur l'addition),
— dans tous les autres cas, on progresse.

Sous la forme d'une table, cela donnerait donc :

Sommet de pile	**Prochain mot**	**Action**
n	quelconque	réduction $E \prec n$
$($ E $)$	quelconque	réduction $E \prec (E)$
$E * E$	quelconque	réduction $E \prec E*E$
$E + E$	$*$	progression
	autre que $*$	réduction $E \prec E+E$
autres cas	quelconque	progression

Notez qu'une telle table, que nous devons explicitement construire et stocker en mémoire, ne peut pas être infinie. Nous ne tiendrons donc pas compte de toute la pile, qui peut être arbitrairement grande, mais seulement de quelques éléments pris à son sommet. Ce nombre dépend notamment de la taille des règles de la grammaire, et ne dépasse jamais 3 dans notre exemple.

Même parmi les motifs de trois éléments, tous n'ont pas besoin d'être pris en compte.

— D'une part, il peut suffire de moins de trois éléments pour prendre une décision. Par exemple, si on a au sommet de la pile un symbole n, il n'est pas utile de regarder ce qui se trouve dessous pour prendre la décision de réduire $E \prec n$.
— D'autre part, certaines séquences comme $($ $+$ ou $+$ $*$ ou encore E E ne peuvent exister dans une expression arithmétique bien formée. Autrement dit, avec un symbole $($ au sommet de la pile, il est inutile de progresser si le prochain symbole est $+$: on peut plutôt immédiatement interrompre l'analyse et signaler une erreur de syntaxe.

Finalement les motifs qui nous intéressent au sommet de la pile sont uniquement ceux qui correspondent à un préfixe de l'une des règles de la grammaire, soit dans notre exemple :

— les motifs n, $($ E $)$, E $*$ E et E $+$ E déjà identifiés,
— les motifs $($, $($ E, E, E $+$ et E $*$ qui correspondent à des versions incomplètes d'un ou plusieurs des motifs précédents,
— et la pile vide.

Chaque opération de progression ajoute un élément au sommet de la pile et fait donc passer de l'un à l'autre de ces motifs, du moins si le symbole empilé est bien cohérent avec l'une des règles de la grammaire.

On peut donc compléter notre tableau comme ci-dessous, en utilisant trois principes pour distinguer les moments où la progression est possible des moments où il faut échouer.

— une expression ne peut commencer que par *n* ou (,
— après (, + ou * on a le début d'une nouvelle expression,
— on ne peut fermer une parenthèse qu'après avoir identifié une expression suivant une parenthèse ouvrante.

On déclare également que l'analyse est complète et est un succès lorsque l'on a sur la pile une unique expression et que l'entrée a été intégralement consommée.

Sommet de pile	**Prochain mot**	**Action**
∅	*n* ou (	progression
	autres	échec
n	quelconque	réduction $E \prec n$
(	*n* ou (	progression
	autres	échec
(*E*	) ou + ou *	progression
	autres	échec
(*E*)	quelconque	réduction $E \prec (E)$
E	+ ou *	progression
	fin de l'entrée	succès
	autres	échec
E *	*n* ou (	progression
	autres	échec
E * *E*	quelconque	réduction $E \prec E*E$
E +	*n* ou (	progression
	autres	échec
E + *E*	*	progression
	autres	réduction $E \prec E+E$

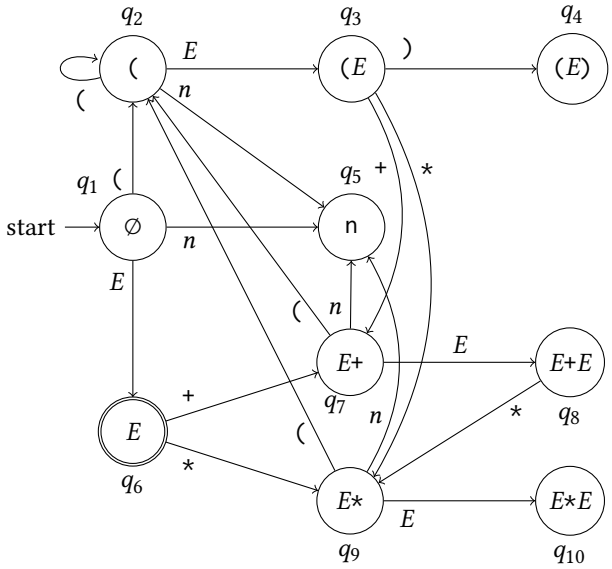
On peut écrire relativement facilement un programme appliquant les règles regroupées dans ce tableau.

```
type token = INT of int | PLUS | STAR | LPAR | RPAR | EOF
type expr  = Cst of int | Add of expr * expr | Mul of expr * expr
type fragment = T of token | E of expr

let parse input =
  let rec ps (p: fragment list *  token list) = match p with
    | ([E e], [EOF]) -> e

    | ([]   ,  (INT _ as t) :: l)
    | ([]   ,  (LPAR  as t) :: l) ->
    ps([T t],                  l)

    | (T (INT n) :: s,  l) ->
    ps(E (Cst n) :: s,  l)

    | (       T LPAR :: s,  (INT _ as t) :: l)
    | (       T LPAR :: s,  (LPAR  as t) :: l) ->
    ps(T t :: T LPAR :: s,                 l)

    | (       E e :: T LPAR :: s,  (RPAR as t) :: l)
    | (       E e :: T LPAR :: s,  (PLUS as t) :: l)
    | (       E e :: T LPAR :: s,  (STAR as t) :: l) ->
    ps(T t :: E e :: T LPAR :: s,                l)

    | (T RPAR :: E e :: T LPAR :: s,  l) ->
    ps(          E e           :: s,  l)
    | (       E e :: [],  (PLUS as t) :: l)
    | (       E e :: [],  (STAR as t) :: l) ->
    ps(T t :: E e :: [],                 l)
```

Ce code pourrait être rendu plus compact en factorisant certaines lignes à l'aide de l'opérateur |, et légèrement plus efficace en évitant de reconstruire des fragments de liste identiques à ceux déjà présents. Il ne s'agit cependant que d'une étape intermédiaire, avant une amélioration bien plus fondamentale.

```
  | (       T (PLUS as o) :: E e :: s,  (INT _ as t) :: l)
  | (       T (PLUS as o) :: E e :: s,  (LPAR  as t) :: l)
  | (       T (STAR as o) :: E e :: s,  (INT _ as t) :: l)
  | (       T (STAR as o) :: E e :: s,  (LPAR  as t) :: l) ->
  ps(T t :: T o           :: E e :: s,                  l)

  | (E e2 :: T STAR :: E e1 :: s,  l) ->
  ps(   E (Mul (e1, e2))    :: s,  l)

  | (          E e1 :: T PLUS :: E e2 :: s,  STAR :: l) ->
  ps(T STAR :: E e1 :: T PLUS :: E e2 :: s,          l)

  | (E e2 :: T PLUS :: E e1 :: s,  l) ->
  ps(   E (Add (e1, e2))    :: s,  l)

  | _ -> failwith "syntax error"
in
ps ([], input)
```

Modélisation de l'algorithme par un automate. Nous avons donc identifié un nombre fini de motifs intéressants, que nous pouvons considérer comme un nombre fini d'*états d'un automate*. Les transitions de cet automate sont alors étiquetées par les différents éléments qui peuvent se trouver sur la pile, c'est-à-dire des symboles terminaux ou non terminaux.



L'unique état acceptant est celui qui décrit une pile formée d'une unique expression.

Le passage d'un état à l'autre est clair lors d'une opération de progression. Par exemple, partant de l'état q_3 correspondant au motif (E, si le prochain mot est) alors on progresse et on arrive dans l'état q_4 correspondant au motif (E).

Le cas d'une opération de réduction est plus délicat. Partant de l'état q_4 du motif (E) on va systématiquement faire la réduction $E \prec (E)$. Dans quel état arrivons nous alors ? Cela dépend en réalité de ce qui se trouve dans la pile au-delà du motif (E) que nous avions identifié. Voici trois possibilités (ce ne sont pas les seules !) :

Sommet de pile avant	Sommet de pile après
((E)	(E
E + (E)	E + E
E * (E)	E * E

Dans les trois cas présentés dans ce tableau, nous étions *avant le début de l'analyse* du motif (E) dans l'un des états q_2, q_7 ou q_8 (motif (ou E + ou E *). Par exemple, dans la troisième ligne nous avons mis en suspens la reconnaissance d'une opération de multiplication le temps de compléter l'analyse de son deuxième opérande, une expression entre parenthèses potentiellement complexe. L'état obtenu après réduction est l'un des états q_3, q_8 ou q_{10} (motif (E ou E + E ou E * E), chacun accessible par une transition E à partir de l'un des états de départ identifiés. Ce retour à un état précédent correspond, après reconstruction d'un fragment de phrase, à un retour au contexte d'analyse qui englobait ce fragment.

Automate à pile. Du fait de ces retours en arrière, nous ne pouvons pas remplacer toute la pile des éléments déjà analysés par un unique état, comme nous le faisions pour l'analyse lexicale. Ici à la place, nous allons utiliser une pile d'états, qui nous permettra après chaque réduction d'aller consulter l'état dans lequel nous étions préalablement à l'analyse du motif qui vient d'être réduit, pour correctement calculer le nouvel état de la pile.

Nous construisons donc un automate dont :

— les états correspondent à des motifs du sommet de pile,
— les transitions sont étiquetées par des symboles terminaux ou non terminaux,
— certains états sont associés à des opérations de réduction,

et nous l'utilisons conjointement avec une pile des états rencontrés qui correspondent à des motifs dont l'analyse est encore en cours.

Étant donné un tel automate, une pile $q_0 \dots q_n$ d'états, et un prochain mot a, on avance dans l'analyse comme suit :

— s'il existe une transition depuis l'état q_n pour le symbole terminal a vers un état q_{n+1}, alors on consomme l'entrée a et on empile l'état q_{n+1},
— si l'état q_n est associé à une opération de réduction $X \prec \beta$, alors on dépile $|\beta|$ éléments de la pile (pour retrouver l'état q_i datant d'avant le début de la lecture de la séquence β), et on empile l'état q' cible de l'unique transition $q_i \xrightarrow{X} q'$,
— s'il n'y a ni prochain mot ni réduction possible, et que la pile contient un unique symbole non terminal, alors on a fini l'analyse,
— dans les autres cas l'analyse s'arrête et échoue.

On peut alors reprendre notre tableau d'analyse précédent, en utilisant une pile d'états plutôt qu'une pile d'éléments. On laisse ici la pile d'éléments uniquement à titre de comparaison : seule la pile d'états est utilisée par l'algorithme d'analyse.

Pile	**Pile d'états**	**Reste**	**Action**
$\emptyset$	q_1	n_1 * (n_2 + n_3)	progression
n_1	$q_1\ q_5$	* (n_2 + n_3)	réd. $E \prec n$
E	$q_1\ q_6$	* (n_2 + n_3)	progression
E *	$q_1\ q_6\ q_9$	(n_2 + n_3)	progression
E * (	$q_1\ q_6\ q_9\ q_2$	n_2 + n_3)	progression
E * (n_2	$q_1\ q_6\ q_9\ q_2\ q_5$	+ n_3)	réd. $E \prec n$
E * (E	$q_1\ q_6\ q_9\ q_2\ q_3$	+ n_3)	progression
E * (E +	$q_1\ q_6\ q_9\ q_2\ q_3\ q_7$	n_3)	progression
E * (E + n_3	$q_1\ q_6\ q_9\ q_2\ q_3\ q_7\ q_5$	)	réd. $E \prec n$
E * (E + E	$q_1\ q_6\ q_9\ q_2\ q_3\ q_7\ q_8$	)	réd. $E \prec E$+E
E * (E	$q_1\ q_6\ q_9\ q_2\ q_3$	)	progression
E * (E)	$q_1\ q_6\ q_9\ q_2\ q_3\ q_4$	$\emptyset$	réd. $E \prec (E)$
E * E	$q_1\ q_6\ q_9\ q_{10}$	$\emptyset$	réd. $E \prec E$*E
E	$q_1\ q_6$	$\emptyset$	succès

Tables d'analyse. Pour représenter l'automate et les différentes actions de progression ou de réduction associées à ses états, on utilise traditionnellement deux tables.

— La ***table d'action*** donne l'action à effectuer en fonction de l'état courant et du prochain mot (un symbole terminal, ou l'absence de prochain mot si l'on a déjà atteint la fin de la phrase). L'action est à choisir parmi les suivantes :
  — progresser, et la table donne le prochain état (correspond aux transitions étiquetées par un symbole terminal),
  — réduire, et la table précise la règle à utiliser (correspond aux état décrivant un motif complet),
  — finir l'analyse (avec succès)
  — échouer
— La ***table de saut*** donne l'état cible en fonction d'un état précédent et d'un symbole non terminal qui vient d'être reconnu (correspond aux transitions étiquetées par un symbole non terminal).

À partir de ces tables, vous pouvez tenter d'écrire une nouvelle version de l'analyseur ! Solution en fin de chapitre.

Dans notre exemple.

	Actions						**Sauts**
	n	(	)	+	*	#	E
q_1	q_5	q_2					q_6
q_2	q_5	q_2					q_3
q_3			q_4	q_7	q_9		
q_4	$E \prec (E)$						
q_5	$E \prec n$						
q_6				q_7	q_9	succès	
q_7	q_5	q_2					q_8
q_8	$E \prec E+E$				q_9	$E \prec E+E$	
q_9	q_5	q_2					q_{10}
q_{10}	$E \prec E*E$						

6.2 Construction d'automates d'analyse ascendante

Nous allons maintenant voir comment, partant d'une grammaire, nous pouvons construire de manière systématique des automates tels que celui vu précédemment pour les expressions arithmétiques. Ces automates font partie d'une famille baptisée ***LR*** (***L**eft-to-right scanning, **R**ightmost derivation*) et qui contient toute une hiérarchie d'automates de plus en plus précis mais aussi de plus en plus difficiles à construire.

Construction d'un automate LR(0). L'automate le plus simple de la famille LR est appelé LR(0). Chacun de ses états représente un niveau d'avancement dans la reconnaissance du membre droit d'une règle de la grammaire (T, N, S, R) considérée. Un état a donc la forme

$$[X \prec \alpha \bullet \beta]$$

où

- $X \in N$ est un symbole non terminal,
- α et β sont deux séquences de symboles (terminaux ou non),
- $X \prec \alpha\beta$ est une règle.

La signification d'un tel état est : « nous sommes en train de chercher à reconnaître la séquence $\alpha\beta$ pour la regrouper en un fragment X, nous avons déjà reconnu la partie α et il reste à reconnaître la partie β ». Dans le cas particulier d'un état $[X \prec \alpha\bullet]$ où β est la séquence vide, nous avons reconnu l'intégralité de la séquence pouvant être regroupée en X.

Pour simplifier le traitement de certains cas limites, on ajoute à notre grammaire un symbole spécial # représentant la fin de l'entrée, et on cherche à reconnaître une phrase de la forme S #. On peut voir cela comme le remplacement du symbole de départ S par un nouveau symbole de départ $S_\#$ auquel est associé une unique règle $S_\# \prec S$ #.

L'automate LR(0) non déterministe est défini par les éléments suivants :

- les états sont tous les triplets $[X \prec \alpha \bullet \beta]$ où $X \in N$, $\alpha, \beta \in (T \cup N)^*$ et $X \prec \alpha\beta \in R$,
- l'unique état initial est $[S_\# \prec \bullet S\#]$,
- l'unique état acceptant est $[S_\# \prec S \bullet \#]$,
- les transitions sont toutes celles qui peuvent être formées de l'une des trois manières suivantes :
  - $[Y \prec \alpha \bullet a\beta] \xrightarrow{a} [Y \prec \alpha a \bullet \beta]$, avec a symbole terminal,
  - $[Y \prec \alpha \bullet X\beta] \xrightarrow{X} [Y \prec \alpha X \bullet \beta]$, avec X symbole non terminal,
  - $[Y \prec \alpha \bullet X\beta] \xrightarrow{\varepsilon} [X \prec \bullet\gamma]$, avec $X \prec \gamma$ une règle pour le symbole non terminal X.

Les transitions de la première forme correspondent à une action de progression, les transitions de la deuxième forme correspondent à un saut (c'est-à-dire à la prise en compte d'un groupe déjà analysé), et les actions de la troisième forme correspondent à la mise en suspens de l'analyse d'une séquence pour se concentrer sur l'un de ses composants.

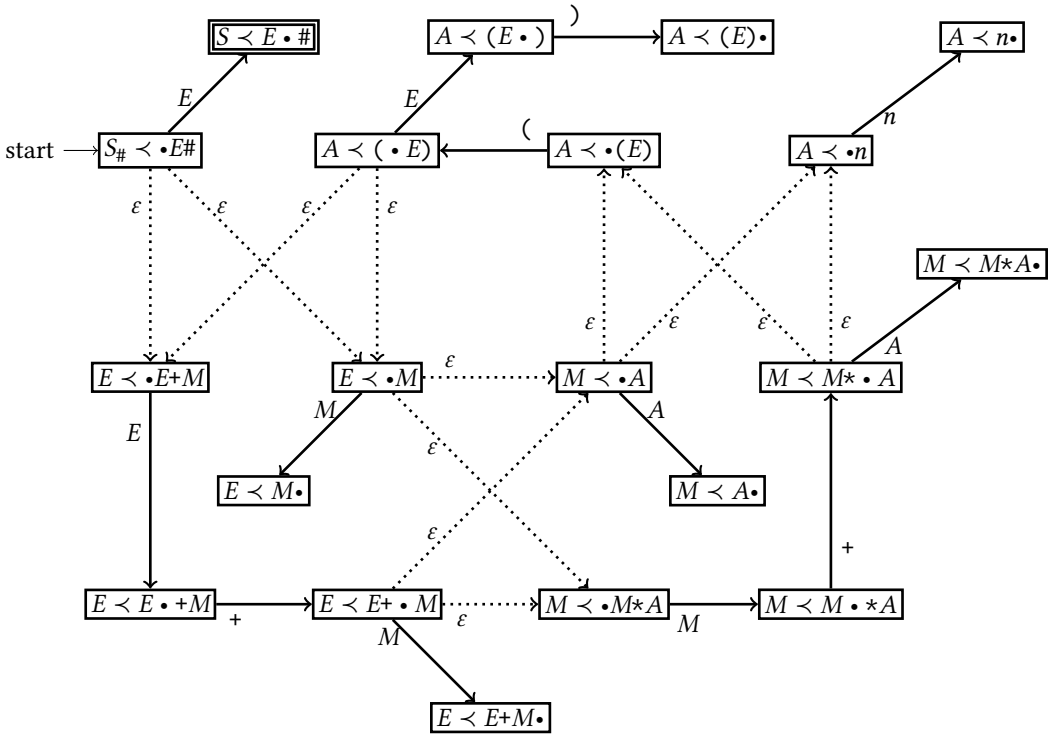
Exemple. En prenant la grammaire suivante des expressions arithmétiques

$$\begin{array}{rcl} E & ::= & E + M \\ & | & M \\ M & ::= & M * A \\ & | & A \\ A & ::= & n \\ & | & (E) \end{array}$$

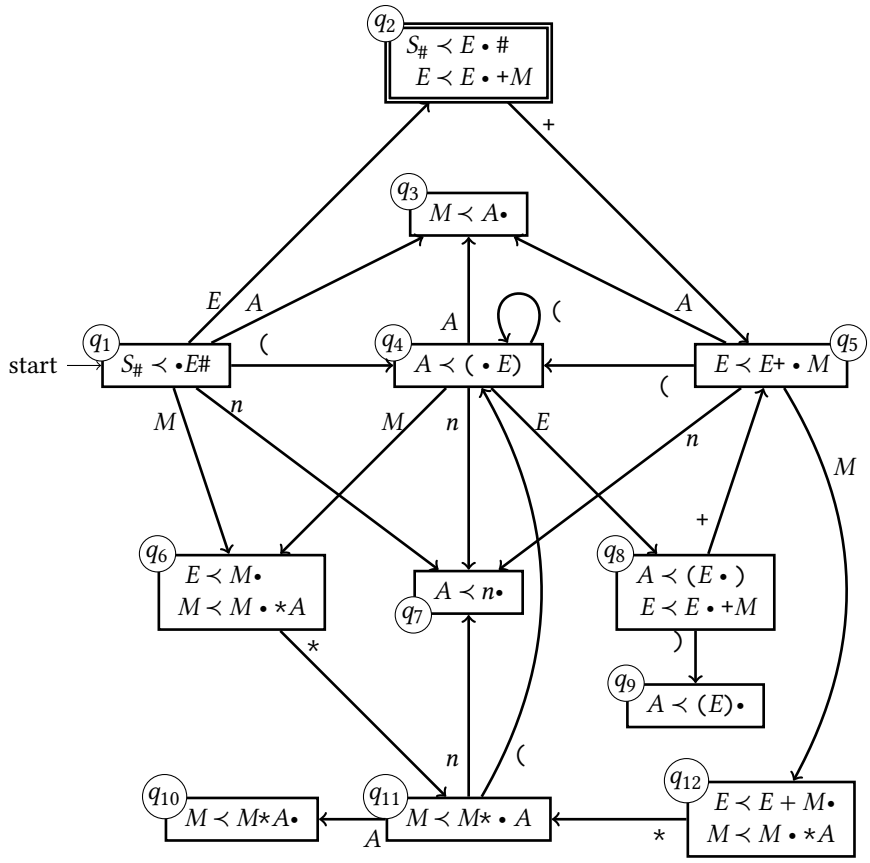
et en y ajoutant la règle de départ

$$S_\# \quad ::= \quad E\,\#$$

on obtiendrait ainsi l'automate ci-dessous. Note : les transitions ε sont affichées en pointillés sur ce dessin, pour les distinguer facilement des transitions normales.



Cet automate n'est pas déterministe du fait des transitions ε apportées par la troisième forme de transition. On peut en revanche le déterminiser en appliquant les techniques déjà vues, et former des états correspondant à des unions d'états de l'automate d'origine.



Dans ce dessin, on a gardé une écriture compacte des unions d'états, en n'indiquant que les éléments qui ne peuvent pas être directement déduits par des transitions ε. Ainsi par exemple, l'état q_{11} correspond à l'union d'états

$$\begin{aligned} M &\prec M\ast \bullet A \\ A &\prec \bullet n \\ A &\prec \bullet (E) \end{aligned}$$

car $A \prec \bullet n$ et $A \prec \bullet (E)$ peuvent chacun être atteint par une transition ε depuis $M \prec M\ast \bullet A$, et l'état q_1 correspond à l'union d'états

$$\begin{aligned} S_\# &\prec \bullet E\# \\ E &\prec \bullet E+M \\ E &\prec \bullet M \\ M &\prec \bullet M\ast A \\ M &\prec \bullet A \\ A &\prec \bullet n \\ A &\prec \bullet (E) \end{aligned}$$

où $M \prec \bullet A$ par exemple peut-être atteint en enchaînant deux transitions ε à partir de $S_\# \prec \bullet E\#$, et $A \prec \bullet (E)$ en poursuivant avec une troisième.

Notez que malgré cette représentation compacte, les états q_2, q_6, q_8 et q_{12} contiennent plusieurs éléments. Il s'agit de situations dans lesquelles la dernière transition pouvait s'appliquer à plus d'une ligne de l'état précédent. Par exemple, une transition E depuis l'état q_1 peut faire avancer aussi bien dans $[S_\# \prec E \bullet \#]$ que dans $[E \prec E \bullet +M]$.

Tables d'analyse. On déduit ensuite de l'automate déterministe obtenu des tables d'analyse. Pour la table d'actions :

- pour toute transition $q \xrightarrow{a} q'$ avec a un symbole terminal, on place dans la case (q, a) une action de progression, avec pour cible l'état q',
- pour tout état q contenant une ligne de la forme $[X \prec \alpha\bullet]$, on a sur toute la ligne q une action de réduction de la règle $X \prec \alpha$ (cette action vaut donc quelque soit le prochain mot),
- si un état q contient la ligne $[S_\# \prec S \bullet \#]$, on place dans la case $(q, \#)$ l'action « succès ».

Pour la table des sauts :

- pour toute transition $q \xrightarrow{X} q'$ avec X un symbole non terminal, on place dans la case (q, X) un saut vers q'.

En appliquant à notre exemple on obtient les tables suivantes.

	Actions						**Sauts**		
	n	(	)	+	$\ast$	#	E	M	A
q_1	q_7	q_4					q_2	q_6	q_3
q_2				q_5		ok			
q_3	$M \prec A$								
q_4	q_7	q_4					q_8	q_6	q_3
q_5	q_7	q_4						q_{12}	q_3
q_6					q_{11}				
	$E \prec M$								
q_7	$A \prec n$								
q_8			q_9	q_5					
q_9	$A \prec (E)$								
q_{10}	$M \prec M\ast A$								
q_{11}	q_7	q_4							q_{10}
q_{12}					q_{11}				
	$E \prec E+M$								

Conflits. La table précédente présente des particularités dans les lignes des états q_6 et q_{12}. Dans ces états, si le prochain symbole est $\ast$ alors la table d'actions permet deux actions différentes :

- progresser (avec transition vers l'état q_{11}),
- réduire la règle $E \prec M$ (état q_6) ou $E \prec E+M$ (état q_{12}).

Autrement dit, la table d'action n'est pas déterministe et ne donne pas un algorithme de reconnaissance comme attendu. On appelle cette ambiguïté entre plusieurs actions un ***conflit d'analyse***. On classe ces conflits en deux sortes, sur lesquelles nous reviendrons plus tard :

— les conflits entre progression et réduction (*shift/reduce*),
— les conflits entre plusieurs réductions (*reduce/reduce*) [8].

Il y a deux explications possibles à l'apparition d'un conflit d'analyse :

— soit la grammaire est ambiguë,
— soit la méthode LR(0) n'est pas suffisamment précise pour cette grammaire.

Selon la situation, nous avons plusieurs manières de remédier à ce problème et obtenir un algorithme d'analyse déterministe. Ces différentes techniques peuvent même être combinées.

— On peut utiliser une méthode plus fine que LR(0). La méthode « standard » est d'ailleurs justement un cran au-dessus dans la hiérarchie. À noter : même au niveau supérieur cela peut rester insuffisant, et de toute façon cela ne suffira pas à régler les conflits dans le cas d'une grammaire ambiguë.
— On peut modifier la grammaire pour qu'elle soit plus adaptée à l'outil d'analyse. Il s'agit donc de trouver une autre grammaire reconnaissant les mêmes structures de phrase. Il existe quelques techniques mais c'est globalement difficile. À garder en dernier recours.
— On peut retoucher manuellement la table d'actions pour ne laisser qu'un seul choix possible dans chaque case où apparaissait un conflit. En pratique, on fait souvent quelque chose qui revient à cela, en donnant des priorités aux symboles et aux règles de réduction. Voir chapitre suivant, où on verra un outil intégrant cette possibilité.

Ici en l'occurrence notre grammaire des expressions arithmétiques n'est pas ambiguë, et il va suffire d'une petite amélioration à la méthode LR(0) pour obtenir des tables déterministes.

6.3 Approfondissement : tables d'analyse et automates plus précis

Analyse SLR(1). Avec l'analyse LR(0) on permet la réduction dès que l'on se trouve dans un état de la forme $[X \prec \alpha\bullet]$, indépendamment du prochain symbole. Par exemple avec l'automate des expressions arithmétiques, dans l'état q_{12} :

$$\begin{aligned} E &\prec E{+}M\bullet \\ M &\prec M \bullet {\star} A \end{aligned}$$

on autorise la réduction $[E \prec E{+}M]$ quel que soit le prochain symbole, et la progression vers l'état q_{11} $[M \prec M\star \bullet A]$ seulement si le prochain symbole est $\star$. Le conflit n'apparaît donc que lorsque le prochain symbole est $\star$. Une question se pose alors : est-il raisonnable de faire une réduction dans ce cas ? Autrement dit : après réduction du motif $E{+}M$, nous allons laisser au sommet de la pile un symbole E (à strictement parler, un état décrivant la présence de ce symbole E). Partant de cet état, saurons-nous progresser avec le prochain symbole $\star$?

Cette question peut être reformulée ainsi : avec notre grammaire, est-il possible de dériver une phrase contenant le motif $E\ \star$? Ou autrement dit, le symbole terminal $\star$ fait-il partie des possibles ***suivants*** du symbole non terminal E ? On peut remarquer que dans la grammaire, la seule règle faisant intervenir le symbole $\star$ est la règle

$$M \prec M \star A$$

Pour une formalisation des notions de *suivant*, *premier* et *annulable*, voir section d'approf. « construction d'analyseurs » (chapitre 3, pages 40–42).

Pour former une séquence $E\star$ il faudrait donc être capable de dériver à partir de M une phrase qui terminerait par E. Or seulement trois règles font intervenir E :

$$\begin{aligned} S_{\#} &\prec E\,\# \\ E &\prec E + M \\ A &\prec (\ E\ ) \end{aligned}$$

Dans ces règles, E est suivi de #, de + ou de), mais n'est jamais ni suivi de $\star$, ni le dernier élément de la phrase. Donc, lorsque le prochain symbole est $\star$ la réduction de $E{+}M$ en E est une impasse et il vaut mieux favoriser la progression.

L'analyse SLR(1) (***S**imple LR(1)*) précise les tables LR(0) en y ajoutant cet unique critère : dans la table d'actions, pour l'état q et le prochain mot a, on n'autorise la réduction $[X \prec \beta]$ que si $[X \prec \beta\bullet] \in q$ *et* $a \in \mathsf{Suivants}(X)$.

8. *Question : pourquoi n'y a-t-il pas de conflit* shift/shift *?*

On complète dont la construction de l'automate d'une analyse des annulables, des premiers et des suivants de la grammaire. Pour notre exemple :

— aucun symbole n'est annulable,

— le calcul des premiers se déroule ainsi :

	E	M	A
0.	$\emptyset$	$\emptyset$	$\emptyset$
1.	$\emptyset$	$\emptyset$	$(, n$
2.	$\emptyset$	$(, n$	$(, n$
3.	$(, n$	$(, n$	$(, n$
4.	$(, n$	$(, n$	$(, n$

— et le calcul des suivants donne

	E	M	A
0.	$\emptyset$	$\emptyset$	$\emptyset$
1.	$\#, +,)$	$\star$	$\emptyset$
2.	$\#, +,)$	$\star, \#, +,)$	$\star$
3.	$\#, +,)$	$\star, \#, +,)$	$\star, \#, +,)$
4.	$\#, +,)$	$\star, \#, +,)$	$\star, \#, +,)$

Ainsi en particulier, le symbole $\star$ n'est pas dans les suivants de E, donc l'analyse SLR(1) n'autorise pas la réduction de $E \prec E{+}M$ lorsque le prochain mot est $\star$. Alors les deux conflits d'analyse LR(0) disparaissent.

Analyse LR(1). L'analyse LR(0) est basée sur un automate dont les états sont directement basés sur les règles de la grammaire. Elle ne regarde le prochain mot que pour décider d'une progression. L'analyse SLR(1) utilise le même automate mais considère le prochain mot y compris pour décider d'une réduction. L'analyse LR(1) généralise la prise en compte du prochain mot, et l'intégre aux états de l'automate eux-mêmes. Autrement dit LR(1) utilise un nouvel automate, dont chaque état correspond à une paire d'un état de l'automate LR(0) et d'un prochain mot. On peut donc voir l'automate LR(1) comme une version plus précise de l'automate LR(0) : un même état LR(0) peut être subdivisé en plusieurs états LR(1) en fonction des différents mots suivants possibles, et chaque état LR(1) pourra préconiser des actions différentes de celles de ses cousins grâce à cette connaissance du prochain mot attendu, et limiter ainsi les conflits.

L'automate LR(1) non déterministe est défini par les éléments suivants.

— Les états sont des quadruplets $[X \prec \alpha \bullet \beta, a]$ avec $X \in N$, $\alpha, \beta \in (T \cup N)^*$, $X \prec \alpha\beta \in R$ et $a \in \mathsf{Suivants}(X)$. Un tel état s'interprète comme « nous sommes en train de chercher à reconnaître la séquence $\alpha\beta$ pour la regrouper en un fragment X qui devra être suivi du symbole a, nous avons déjà reconnu la partie α et il reste à reconnaître la partie β (et vérifier la présence de a ensuite) ».

— L'unique état initial est $[S_\# \prec \bullet S, \#]$.

— L'unique état acceptant est $[S_\# \prec S \bullet, \#]$.

— Les transitions sont toutes celles qui peuvent être formées de l'une des trois manières suivants :

  — $[Y \prec \alpha \bullet a\beta, b] \xrightarrow{a} [Y \prec \alpha a \bullet \beta, b]$ avec a symbole terminal,

  — $[Y \prec \alpha \bullet X\beta, b] \xrightarrow{X} [Y \prec \alpha X \bullet \beta, b]$ avec X symbole non terminal,

  — $[Y \prec \alpha \bullet X\beta, b] \xrightarrow{\varepsilon} [X \prec \bullet\gamma, c]$ avec $X \prec \gamma$ une règle et $c \in \mathsf{Premiers}(\beta b)$.

Enfin, la table d'actions nous donne une réduction dans l'état q pour le mot suivant a si $[X \prec \alpha\bullet, a] \in q$.

Finalement, les analyses LR(1) apportent une meilleure prise en compte du mot suivant, qui se traduit par un automate avec plus d'états mais moins de conflits. Le calcul des tables est plus complexe, mais peut tout à fait être pris en charge par un outil. Et c'est exactement cette analyse que fait l'outil que nous verrons au prochain chapitre.

6.4 Approfondissement : code final d'un analyseur LR

On reprend l'analyseur donné à la section 6.1, mais cette fois en se basant sur l'automate et les tables d'analyse, plutôt que sur l'analyse manuelle de séries de lexèmes. Dans un premier temps, on écrit une simple fonction de validation, qui renvoie **`true`** si l'entrée est syntaxiquement correcte, et **`false`** sinon. La fonction récursive principale `vl` fait l'essentiel de l'analyse, et délègue le calcul des sauts à la fonction `jump_e`. La liste `qs` est la pile des états, et la liste `ts` est la liste des lexèmes restant à lire.

```
type token = INT of int | PLUS | STAR | LPAR | RPAR | EOF

let validate input =
  let rec vl ((qs: int list), (ts: token list)) = match qs, ts with
    | (1::_, INT n::ts) -> vl(5::qs, ts)
    | (1::_, LPAR::ts ) -> vl(2::qs, ts)

    | (2::_, INT n::ts) -> vl(5::qs, ts)
    | (2::_, LPAR::ts ) -> vl(2::qs, ts)

    | (3::_, RPAR::ts) -> vl(4::qs, ts)
    | (3::_, PLUS::ts) -> vl(7::qs, ts)
    | (3::_, STAR::ts) -> vl(9::qs, ts)

    | (4::_::_::q::qs, ts) -> vl(jump_e q::q::qs, ts)

    | (5::q::qs, ts) -> vl(jump_e q::q::qs, ts)

    | (6::_, PLUS::ts) -> vl(7::qs, ts)
    | (6::_, STAR::ts) -> vl(9::qs, ts)
    | (6::1::[], [EOF]) -> true

    | (7::_, INT n::ts) -> vl(5::qs, ts)
    | (7::_, LPAR::ts ) -> vl(2::qs, ts)

    | (8::_          , STAR::ts) -> vl(9::qs          , ts)
    | (8::_::_::q::qs, ts      ) -> vl(jump_e q::q::qs, ts)

    | (9::_, INT n::ts) -> vl(5::qs, ts)
    | (9::_, LPAR::ts ) -> vl(2::qs, ts)

    | (10::_::_::q::qs, ts) -> vl(jump_e q::q::qs, ts)

    | _ -> false
  and jump_e = function
    | 1 -> 6
    | 2 -> 3
    | 7 -> 8
    | 9 -> 10
    | _ -> assert false
  in
  vl([1], input)
```

Notez que chaque action de progression empile un nouvel état sur la pile `qs`, alors qu'à l'inverse les actions de réductions se caractérisent par un retour en arrière de plusieurs crans dans cette pile.

Voici une extension du code précédent, où on reconstruit également l'arbre de syntaxe de l'expression reconnue. La fonction récursive principale ps prend comme paramètre supplémentaire une liste es représentant la pile des dernières expressions reconstruites. Contrairement à ce qui était fait dans la section 6.1, cette pile ne contient aucun lexème, et son contenu ne participe jamais au choix de la prochaine étape (la pile qs joue déjà ce rôle) : on se contente de prélever des éléments lorsque nécessaire, ou d'en ajouter de nouveaux lorsqu'une nouvelle expression est formée.

```
type token = INT of int | PLUS | STAR | LPAR | RPAR | EOF
type expr = Cst of int | Add of expr * expr | Mul of expr * expr

let parse input =
  let rec ps ((qs: int list), (ts: token list), (es: expr list)) =
    match qs, ts, es with
    | (1::_, INT n::ts, es) -> ps(6::qs, ts, Cst n::es)
    | (1::_, LPAR::ts , es) -> ps(2::qs, ts, es)

    | (2::_, INT n::ts, es) -> ps(3::qs, ts, Cst n::es)
    | (2::_, LPAR::ts , es) -> ps(2::qs, ts, es)

    | (3::_, RPAR::ts, es) -> ps(4::qs, ts, es)
    | (3::_, PLUS::ts, es) -> ps(7::qs, ts, es)
    | (3::_, STAR::ts, es) -> ps(9::qs, ts, es)

    | (4::_::_::q::qs, ts, es) -> ps(jump_e q::q::qs, ts, es)

    | (6::_, PLUS::ts, es) -> ps(7::qs, ts, es)
    | (6::_, STAR::ts, es) -> ps(9::qs, ts, es)
    | (6::1::[], [EOF], [e]) -> e

    | (7::_, INT n::ts, es) -> ps(8::qs, ts, Cst n::es)
    | (7::_, LPAR::ts , es) -> ps(2::qs, ts, es)

    | (8::_, STAR::ts, es) -> ps(9::qs, ts, es)
    | (8::_::_::q::qs, ts, e2::e1::es) ->
       ps(jump_e q::q::qs, ts, Add(e1, e2)::es)

    | (9::_, INT n::ts, es) -> ps(10::qs, ts, Cst n::es)
    | (9::_, LPAR::ts , es) -> ps( 2::qs, ts, es)

    | (10::_::_::q::qs, ts, e2::e1::es) ->
       ps(jump_e q::q::qs, ts, Mul(e1, e2)::es)

    | _ -> failwith "syntax␣error"
  and jump_e = function
    | 1 -> 6
    | 2 -> 3
    | 7 -> 8
    | 9 -> 10
    | _ -> assert false
  in
  ps([1], input, [])
```

Remarquez au passage que ce code contient également un raccourci : on a supprimé l'état numéro 5, et *inliné* son comportement dans les états 1, 2, 7 et 9.

Défi : écrivez l'analyseur correspondant à l'analyse SLR(1) de la grammaire donnée aux sections 6.2 et 6.3.

7 Outillage pour l'analyse syntaxique

Ce chapitre présente deux outils qui automatisent la création de programmes d'analyse lexicale et d'analyse grammaticale à l'aide des techniques des deux chapitres précédents.

7.1 Génération d'analyseurs lexicaux avec ocamllex

L'écriture de l'analyseur lexical reconnaissant un ensemble de lexèmes donné peut être en partie automatisée. C'est l'objet des outils de la famille LEX, dont le représentant en caml est `ocamllex`.

Principe : on écrit dans un fichier `.mll` l'ensemble des expressions régulières à reconnaître et les traitements associés, puis l'utilitaire `ocamllex` traduit ce fichier en un programme caml réalisant l'analyse. Le cœur du fichier `.mll` est constitué des expressions régulières à reconnaître et des traitements associés, sous la forme suivante rappelant une définition de fonction par cas :

Donc : `ocamllex` peut être vu comme un *compilateur*, d'un langage dédié à la description d'expressions régulières vers Caml.

```
    | expression_régulière_1
        { traitement_1 }

    | expression_régulière_2
        { traitement_2 }

    |  ...
```

Cette partie va être traduite en un automate et des fonctions de reconnaissance. Le fichier `.mll` contient également un prélude et un épilogue, dans lesquels on peut inclure du code caml arbitraire destiné à être copié au début ou à la fin du fichier `.ml` produit.

Dans cette section, on présente l'outil `ocamllex` sur l'exemple d'un analyseur qui extrait les lexèmes d'un fichier et copie à la volée le contenu des commentaires dans un fichier de documentation. Note : l'intégralité des extraits de code de cette séquence, pris dans l'ordre, forment un fichier `.mll` complet générant un programme autonome.

Prélude d'un fichier `.mll`. Le prélude est le bon endroit pour définir des variables globales, des structures de données, des fonctions auxiliaires qui seront utilisées lors des traitements associés aux règles.

C'est une zone entre accolades au début du fichier. On y écrit du code Caml habituel, qui sera repris tel quel dans le fichier produit. On y trouve typiquement les mêmes choses qui garnissent généralement le début d'un fichier `.ml`, et par exemple :

— des importations de modules

```
{
  open Lexing
  open Printf
```

— des définitions de types et d'exceptions

```
  type token =
    | IDENT of string
    | INT   of int
    | FLOAT of float
    | PLUS
    | PRINT
  exception Eof
```

— des déclarations de variables globales

```
  let lines = ref 0
  let file = Sys.argv.(1)
  let cout = open_out (file ^ ".doc")
```

— des définitions de fonctions auxiliaires

```
  let print s = fprintf cout s
}
```

Note : on a défini ici avec `token` un type pour les lexèmes à reconnaître. Cependant, dans le cas d'une utilisation de cet analyseur lexical dans un développement plus grand, cette définition sera plutôt placée dans un autre fichier et importée avec une directive **open**.

Définition d'expressions régulières auxiliaires. La zone principale est placée immédiatement après le prélude et définit les règles de reconnaissance. La syntaxe est spécifique à `ocamllex`. Souvent, les règles de reconnaissance elle-mêmes sont précédées d'une première partie dans laquelle on peut définir un certain nombre de raccourcis pour des expressions régulières avec la syntaxe

```
let nom = expression_régulière
```

Les expressions régulières sont construites avec la syntaxe suivante :

`_`	n'importe quel caractère
`'a'`	caractère spécifique
`"abc"`	chaîne de caractères spécifique
`[...]`	alternative parmi un ensemble de caractères
`[^...]`	alternative parmi le complément d'un ensemble de caractères
r_1 `|` r_2	alternative
r_1 r_2	concaténation
`r*`	étoile (répétition de `r`, éventuellement vide)
`r+`	répétition non vide
`r?`	présence optionnelle de `r`
`eof`	fin de l'entrée

Lors de la définition d'un ensemble de caractères, on peut énumérer les caractères en les séparant par une espace `['a' 'b' 'c']` ou sélectionner toute une plage en utilisant un tiret `['a'-'c']`. Ces deux versions peuvent être combinées.

Expression régulière reconnaissant un chiffre

```
let digit = ['0'-'9']
```

Expression régulière reconnaissant une lettre, miniscule ou majuscule

```
let alpha = ['a'-'z' 'A'-'Z']
```

Expression régulière reconnaissant un identifiant de caml : une suite non vide pouvant contenir des chiffres, des lettres et le caractère `'_'`, et commençant par une lettre.

```
let ident = alpha (digit | alpha | '_')*
```

Expression régulière reconnaissant la partie décimale d'un nombre : un point et une suite éventuellement vide de chiffres.

```
let decimals = '.' digit*
```

Expression régulière reconnaissant un exposant : la lettre `e`, minuscule ou majuscule, suivie d'un nombre entier positif ou négatif. L'indication du signe de l'exposant est optionnelle.

```
let exponent = ['e' 'E'] ['+' '-']? digit+
```

Expression régulière reconnaissant un nombre flottant, c'est-à-dire un nombre comportant une partie décimale et/ou un exposant.

```
let fpnumber = digit+ (decimals | decimals? exponent)
```

Définition d'une fonction d'analyse. Dans le cœur de la zone principale on définit les règles de reconnaissance proprement dites et les traitements associés. Les règles de reconnaissance sont regroupées sous un nom de fonction introduit par :

```
rule nom_de_la_fonction = parse
```

Après utilisation de ocamllex, le fichier `.ml` produit définira une fonction du nom correspondant, de type `Lexing.lexbuf -> res` où `Lexing.lexbuf` fait référence à la structure `lexbuf` définie dans le module `Lexing`, qui décrit une entrée en cours de lecture, et où `res` est un type de retour dépendant des traitements réalisés.

Commençons par une simple fonction de parcours de texte, utilisée à l'intérieur des commentaires pour copier leur contenu dans un fichier. Cette fonction ne renvoie pas de résultat, et s'arrête à la première occurrence de la chaîne `"*/"`. La fonction compte également le nombre de lignes analysées en mettant à jour la référence `lines`. Le fichier `.ml` généré par ocamllex contiendra une définition de fonction répondant à la signature `scan_text : Lexing.lexbuf -> unit`.

```
rule scan_text = parse
```

Reconnaissance de l'expression régulière `"*/"`. Traitement associé : expression caml de type `unit` entre accolades. Ici il s'agit de ne rien faire (on a atteint notre signal de fin), on utilise la valeur caml `() : unit`. On traite exactement de même la fin de fichier avec l'expression régulière `eof`.

```
  | "*/"  { () }
  | eof   { () }
```

Reconnaissance de toute séquence de caractères autres que le retour à la ligne ou l'étoile. Dans le traitement associé, on commence par récupérer la chaîne reconnue à l'aide de la fonction `lexeme: lexbuf -> string` fournie par le module `Lexing`, appliquée à la variable prédéfinie `lexbuf: lexbuf` désignant l'entrée en cours de lecture. Cette chaîne est copiée dans le fichier à l'aide de notre fonction auxiliaire **`print`** définie dans le prélude. Enfin, on poursuit l'analyse sur la suite du texte. Pour cela on applique la fonction `scan_text` en train d'être définie (elle est donc récursive, mais nous n'avons pas besoin de le préciser dans le code) à l'entrée en cours de lecture représentée par la variable `lexbuf`. À noter : les informations de l'entrée ont bien été mises à jour pour que la lecture passe au caractère suivant.

```
  | [^ '\n' '*']*  {
      let s = lexeme lexbuf in
      print "%s" s;
      scan_text lexbuf
    }
```

Note : lorsque l'expression régulière contient un indicateur de répétition la fonction produite va chercher à reconnaître un fragment de l'entrée le plus long possible. Ainsi la règle précédente va reconnaître la plus grande séquence de caractères non interrompue par `'\n'` ou `'*'`, ou autrement dit tous les caractères jusqu'à la prochaine occurrence de `'\n'` ou `'*'`.

On traite à part la reconnaissance du caractère `'*'` pour éviter que la règle précédente ne capture l'étoile d'une combinaison `"*/"`. La règle du plus long lexème reconnu fait en outre que la séquence `"*/"` pour laquelle nous avons déjà donné une règle sera prioritaire sur l'étoile seule. On traite également à part le cas du retour à la ligne pour y inclure un incrément du compteur de lignes. Dans chacun de ces deux cas en revanche on reprend l'essentiel du traitement précédent : copier le caractère dans le fichier puis poursuivre l'analyse.

```
  | '*'   { print "%c" '*';
            scan_text lexbuf }
  | '\n'  { lines := !lines + 1; print "%c" '\n';
            scan_text lexbuf }
```

Fonctions d'analyse multiples. Il est possible de définir avec ocamllex plusieurs fonctions mutuellement récursives, correspondant à la construction habituelle de caml suivante :

```
let rec f x = ... and g y = ...
```

L'analyse d'un texte peut alors faire appel alternativement à plusieurs fonctions de reconnaissance correspondant chacune à un mode particulier. Notez que cette capacité d'ocamllex n'est pas systématiquement présente dans les autres outils de la famille LEX.

Dans notre cas, l'objectif de la première fonction `scan_text` était de traiter les commentaires en recopiant leur contenu dans un fichier de documentation et en maintenant à jour un compteur de lignes. Nous allons maintenant définir la fonction principale `scan_token` dont l'objectif est de renvoyer le prochain lexème reconnu dans l'entrée. Cette fonction `scan_token` fera appel à `scan_text` pour traiter tout commentaire trouvé dans l'entrée.

La fonction `scan_token` devant produire un lexème, son type sera

```
scan_token: Lexing.lexbuf -> token
```

Cette fonction ne produit qu'un lexème à la fois et n'a pas elle-même le rôle de produire toute la séquence : c'est une fonction englobante qui appellera `scan_token` autant que nécessaire pour obtenir de nouveaux lexèmes (dans un cas d'application typique il s'agira de la fonction principale d'analyse syntaxique).

Définition d'une fonction supplémentaire, liée avec **`and`**

```
and scan_token = parse
```

Dans la recherche des lexèmes, on ignore les espaces, tabulations et sauts de ligne (appelés collectivement les *blancs*. Cela revient, lorsqu'un tel caractère est reconnu, à reprendre l'analyse à partir du caractère suivant.

```
    | [' ' '\t' '\n']*  { scan_token lexbuf }
```

Un commentaire peut également être vus comme une forme plus élaborée de caractère blanc. De même que dans le cas précédent on va donc reprendre la recherche du prochain lexème après la fin du commentaire. Nouveauté ici, pour reconnaître le commentaire lui-même on fait appel à notre fonction précédente `scan_text`, qui va copier le contenu du commentaire dans notre fichier de documentation avant de rendre la main une fois la fin du commentaire atteinte.

```
    | "/**"  { scan_text lexbuf;
              scan_token lexbuf }
```

Reconnaissance de lexèmes : voici par exemple pour l'opérateur `+`, le mot-clé **`print`**, ou un identifiant. Dans chacun de ces cas on renvoie un valeur du type `token` défini plus haut, qui identifie le lexème reconnu.

```
    | '+'         { PLUS }
    | "print"     { PRINT }
    | ident as s  { IDENT s }
```

Dans le dernier cas, on a utilisé la notation **`as`** pour nommer `s` la chaîne de caractère reconnue. On peut ainsi faire référence à cette chaîne reconnue dans le traitement `IDENT s` sans faire appel à `lexeme lexbuf`.

Priorités des règles. Concernant les priorités dans la sélection des règles, l'analyse cherche toujours à reconnaître une chaîne la plus longue possible. En conséquence, si plusieurs règles sont susceptibles de reconnaître un préfixe de la chaîne analysée, on appliquera la règle permettant de reconnaître le plus long préfixe.

Ainsi, en supposant que la chaîne analysée est `print_int 3` la règle du mot-clé `"print"` permet de reconnaître la séquence **`print`** et la règle des identifiants permet de reconnaître la séquence `print_int`. C'est la deuxième qui est sélectionnée, et donc aussi le deuxième traitement `IDENT s` qui est appliqué.

Lorsque deux règles permettent de reconnaître la même plus longue séquence, l'égalité est résolue en sélectionnant la règle qui apparaît avant l'autre dans le fichier `.mll`. Ainsi, en supposant que la chaîne analysée est **`print`** `3` les deux règles précédentes permettent de reconnaître la même séquence **`print`**, et on sélectionne donc celle correspondant au mot-clé `"print"` (première règle donnée) et non à un identifiant.

Le même phénomène apparaît dans la gestion des nombres entiers ou décimaux ci-dessous, où on donne deux règles : la première reconnaissant un entier (lexème `INT`) et la seconde reconnaissant un nombre flottant (lexème `FLOAT`).

```
    | digit+ as n    { INT (int_of_string n) }
    | fpnumber as n  { FLOAT (float_of_string n) }
```

Notez que dans un cas comme dans l'autre, le « nombre » `n` reconnu est donné par une chaîne de caractères, qui doit encore être traduite en une valeur de type `int` ou `float`.

On conclut cette fonction de reconnaissance principale en déclenchant une erreur lorsqu'aucun motif n'est reconnu, et en levant une exception spécifique lorsque la fin du fichier est atteinte.

```
    | _ as c  { failwith (sprintf "Caractère␣non␣reconnu␣:␣%c" c) }
    | eof     { raise Eof }
```

Épilogue d'un fichier `.mll`. On peut conclure un fichier `ocamllex` par une zone libre qui, comme le prélude, contient du code caml arbitraire qui sera intégré tel quel au fichier `.ml` produit, mais cette fois à la fin. Ce code peut donc faire référence à tout ce qui a été défini dans le prélude, ainsi qu'aux fonctions de reconnaissance définies dans la partie principale. À nouveau, cette zone est délimitée par des accolades.

Cette zone est le bon endroit où placer ce qui correspondrait à une fonction `main`, dans le cas où on utilise `ocamllex` pour produire un programme complet et autonome. Exemple : ici, on

produit un programme qui affiche les lexèmes sur la sortie standard et copie les commentaires dans un fichier `.doc`.

On fournit donc pour commencer une fonction convertissant un lexème en une chaîne de caractères.

```
{
  let rec token_to_string = function
    | IDENT s -> sprintf "IDENT␣%s" s
    | INT i   -> sprintf "INT␣%i" i
    | FLOAT f -> sprintf "FLOAT␣%f" f
    | PLUS    -> "PLUS"
    | PRINT   -> "PRINT"
```

Puis on donne le code du programme principal, qui ouvre le fichier à analyser, initialise la structure `lexbuf` qui sera utilisée par les fonctions d'analyse, puis lance une boucle infinie de lecture des lexèmes, qui ne sera interrompue que par l'exception `Eof` levée à la fin du fichier (ou éventuellement par une erreur si le fichier contient des parties non reconnues).

```
  let () =
    let cin = open_in file in
    try
      let lexbuf = Lexing.from_channel cin in
      while true do
        let tok = scan_token lexbuf in
        printf "%s\n" (token_to_string tok)
      done
    with
      | Eof ->
        close_in cin;
        close_out cout;
        printf "Nombre␣de␣lignes␣de␣commentaires␣:␣%d\n" !lines
}
```

Notez que cette boucle infinie n'existe pas dans l'analyseur lexical utilisé par un compilateur. Dans ce cas en effet le rythme est donné par la fonction d'analyse syntaxique qui demande les lexèmes un à un à mesure des besoin de son analyse, comme nous le verrons dans la suite du chapitre.

Discussion efficacité. En extrapolant ce qui est fait dans cet exemple, on aurait tendance à définir une nouvelle règle pour chaque mot-clé du langage. Cependant, cela engendrerait lors de l'utilisation du générateur `ocamllex` la création d'un automate inutilement gros.

Une optimisation simple consiste à avoir une seule règle reconnaissant toute séquence ayant la forme d'un identifiant (les mots-clés ayant aussi cette forme), puis à tester dans le traitement associé si la séquence reconnue appartient à la liste des mots-clés pour renvoyer le bon lexème.

Si l'on souhaite en plus que l'analyseur ne soit pas sensible à la casse, c'est-à-dire qu'il oublie toutes les alternances entre lettres majuscules et minuscules dans les mots-clés ou les identifiants, mieux vaut de même s'en remettre à un traitement a posteriori, fait une fois qu'une séquence de caractères générale a été identifiée.

7.2 Génération d'analyseurs syntaxiques avec menhir

La construction des tables d'analyse ascendante d'une grammaire et la programmation de l'analyseur correspondant sont nettement plus complexes que dans le cas de l'analyse descendante. Cependant, de même qu'on l'a vu à la section précédente pour l'analyse lexicale, cette tâche peut être automatisée.

C'est l'objet des outils de la famille YACC, représentée en Caml par `ocamlyacc` et `menhir`. On s'intéresse ici à `menhir`, qui est plus moderne et plus puissant que l'outil d'origine `ocamlyacc`.

Principe : on décrit dans un fichier `.mly` l'ensemble des règles de la grammaire à reconnaître, en leur associant des traitements à effectuer pour produire l'arbre de syntaxe abstraite du programme analysé. L'utilitaire `menhir` traduit alors ce fichier en un programme Caml réalisant une analyse ascendante d'un texte en suivant la grammaire fournie.

Le programme Caml obtenu prend en entrée le texte à analyser, mais aussi une fonction d'analyse lexicale fournissant à la demande le prochain lexème. Cette fonction d'analyse lexicale est celle qui a été générée par `ocamllex` à partir de la description des lexèmes.

Donc : `menhir`, similairement à `ocamllex`, peut être vu comme un compilateur, d'un langage dédié à la description de grammaires vers Caml. D'ailleurs, *YACC* signifie *Yet Another Compiler Compiler.*

L'outil `menhir` émet également un avertissement et un diagnostic en cas de conflit dans l'analyse. Ce point sera abordé dans la prochaine section.

Structure d'un fichier `.mly`. Le cœur du fichier `.mly` est constitué des règles de la grammaire à reconnaître et des traitements associés sous une forme proche de celle vue pour `ocamllex`, et cette partie sera traduite en un code Caml. Le fichier commence et termine de même par un prélude et un épilogue contenant des fragments de code Caml intégrés respectivement au début et à la fin du fichier `.ml` produit (cette fois, ces zones sont délimitées par des accolades précédées du symbole `%`).

Prélude d'un fichier `.mly`. Cette zone est toujours le bon endroit pour définir le contexte et les éléments auxiliaires utilisés lors des traitements. On y écrit du code Caml qui sera repris tel quel dans le fichier final.

Pour un exemple minimaliste, on peut y inclure par exemple la définition d'un AST pour des expressions et des programmes.

Note : en conditions réelles, la syntaxe abstraite serait plutôt définie dans un module dédié et le prélude du fichier `.mly` ne ferait que l'importer.

```
%{
  type expression =
    | Cst of int
    | Add of expression * expression
    | Mul of expression * expression
  type program =
      { code : expression }
%}
```

Après ce prélude délimité par `%{` et `%}` commence la partie centrale du fichier. La syntaxe de la partie centrale est cette fois spécifique à `ocamlyacc` ou `menhir` (`menhir` reconnaît la syntaxe `ocamlyacc`, mais introduit aussi de nouveaux éléments).

Déclaration des symboles de la grammaire. Les lexèmes manipulés, qui sont aussi les symboles terminaux de la grammaire, sont déclarés en tête de la partie principale, sous la forme

```
%token nom_du_lexme
```

pour les lexèmes ordinaires sans contenu et

```
%token <type_du_contenu> nom_du_lexeme
```

pour les lexèmes portant une valeur. On peut déclarer plusieurs lexèmes par ligne.

```
(* Constantes entières *)
%token <int> INT
(* Quelques symboles arithmétiques *)
%token LPAR RPAR PLUS STAR
(* Fin de fichier *)
%token EOF
```

Le symbole non terminal de départ de la grammaire est déclaré avec

```
%start nom_du_symbole
```

Cette déclaration est obligatoirement associée à une déclaration de type, indiquant le type de la valeur produite par l'analyse syntaxique

```
%type <type_du_resultat> nom_du_symbole
```

La déclaration des types des autres symboles non terminaux est optionnelle.

Le programme Caml généré par `menhir`/`ocamlyacc` à partir de cette grammaire exportera notamment une fonction portant le nom de ce symbole de départ, et de type

```
(Lexing.lexbuf -> token) -> Lexing.lexbuf -> type_du_resultat
```

Cette fonction prendra donc en paramètre une fonction d'analyse lexicale fournissant à chaque appel le prochain lexème, ainsi que l'entrée à lire (au format `Lexing.lexbuf` qui était utilisé pour l'analyse lexicale). Le résultat aura le type déclaré pour le symbole de départ.

On peut donc déclarer deux symboles non terminaux `prog` et `expr`, et préciser que le symbole de départ est `prog` à l'aide des déclarations suivantes.

```
%type <program> prog
%type <expression> expr
%start prog
```

Le programme généré fournira donc une fonction avec la signature suivante.

```
prog: (Lexing.lexbuf -> token) -> Lexing.lexbuf -> program
```

Notez qu'il n'était pas indispensable de déclarer le symbole `expr`, fournir les règles associées dans la partie suivante est suffisant.

Définition des règles de la grammaire et des traitements associés. On marque la fin de la déclaration des symboles et le début de la définition des règles par une ligne

```
%%
```

On peut ensuite introduire chaque symbole non terminal par une simple ligne

```
nom_du_symbole:
```

suivie des règles associées. Comme avec `ocamllex`, une règle contient un motif, puis entre accolades un traitement associé à l'utilisation de cette règle. Comme pour les traitements de l'analyse lexicale, on inclut ici du code caml arbitraire, avec un type de retour correspondant au type déclaré pour le symbole non terminal en cours de définition. On peut faire référence aux valeurs correspondant aux symboles de la production avec la notation

```
$numero_du_symbole
```

La numérotation prend en compte tous les symboles de la production, qu'ils soient terminaux ou non terminaux, et progresse de gauche à droite.

On peut ainsi définir l'unique règle `prog` $\prec$ `expr EOF` associée au symbole de départ `prog` comme suit. Dans le traitement associé, on récupère la valeur de l'expression dénotée par le symbole non terminal `expr` en première position et on la place dans une structure représentant un programme.

```
prog:
| expr EOF  { {code = $1} }
;
```

Notez que les deux paires d'accolades ici ont des rôles distincts : la paire extérieure délimite le traitement associé à la règle `expr EOF`, tandis que la paire intérieure est la syntaxe d'une structure caml. Le point-virgule (optionnel) marque la fin des règles associées au symbole non terminal `prog`.

L'outil `menhir` propose certaines facilités pour la manipulation des fragments mentionnés dans les règles. Plutôt que de faire référence aux éléments par un numéro (ce qui est la version traditionnelle dans les outils de la famille YACC), on peut associer des noms à certains éléments avec la notation

```
nom = symbole
```

Voici par exemple comment nommer une expression identifiée entre deux parenthèses. Le traitement associé renvoie simplement l'expression trouvée, puisque les parenthèses elles-mêmes n'apparaissent pas dans la syntaxe asbtraite.

```
expr:
| LPAR e=expr RPAR  { e }
```

Notez que cette référence à un élément d'une règle permet aussi bien de récupérer un morceau de programme déjà reconstruit, dans le cas précédent d'un symbole non terminal, qu'une donnée simplement associée à un symbole terminal comme l'entier associé à un symbole `INT` (ceci vaut aussi bien pour la référence via un nom que pour la référence via le numéro).

```
| i=INT  { Cst i }
```

Les différents éléments d'une règle peuvent optionnellement être séparés par des point-virgules si cela facilite la lecture. Cela ne change rien à la signification de la règle. Ainsi les deux déclarations suivantes donnent deux variantes d'écriture pour une signification essentiellement identique.

```
  | e1=expr; PLUS; e2=expr  { Add(e1, e2) }
  | expr STAR expr          { Mul($1, $3) }
  ;

  %%
```

La syntaxe de `menhir` offre également quelques éléments spéciaux pour écrire des règles avec des éléments répétés ou optionnels, que nous aborderons plus tard.

La zone de description des règles termine comme elle a commencé par une ligne contenant uniquement les symboles `%%`.

Épilogue d'un fichier `.mly`. Il s'agit à nouveau d'une zone de code caml arbitraire dont le contenu sera placé à la fin du fichier `.ml` généré. Ce code peut faire référence à la fonction produite pour le symbole de départ. Comme le prélude, l'épilogue est délimité par `%{` et `%}`.

Et maintenant... Si vous compilez avec menhir les déclarations prises en exemple dans cette section, vous obtiendrez un avertissement inquiétant :

```
Warning: 2 states have shift/reduce conflicts.
Warning: 4 shift/reduce conflicts were arbitrarily resolved.
```

L'analyse ascendante réalisée par `menhir` est susceptible de buter sur des conflits. Nous allons voir que ces conflits sont souvent relativement faciles à régler.

7.3 Conflits et priorités

Nous avons vu de nombreuses grammaires pour notre exemple des expressions arithmétiques, avec des caractéristiques différentes. Retenons les suivantes :

— Une grammaire un peu tordue, compatible avec l'analyse ascendante SLR(1).

$$\begin{array}{rcl} E & ::= & E + M \\ & | & M \\ M & ::= & M * A \\ & | & A \\ A & ::= & n \\ & | & (\ E\ ) \end{array}$$

Elle est a fortiori compatible avec l'analyse ascendante LR(1) faite par `menhir`.

— Une grammaire encore un peu plus tordue, mais cette fois compatible avec les techniques d'analyse descendante.

$$\begin{array}{rcl} E & ::= & M\,E' \\ E' & ::= & +\ E \\ & | & \varepsilon \\ M & ::= & A\,M' \\ M' & ::= & *\ M \\ & | & \varepsilon \\ A & ::= & n \\ & | & (\ E\ ) \end{array}$$

Elle permet notamment l'écriture directe d'un analyseur récursif descendant.

— Une grammaire naïve et naturelle.

$$\begin{array}{rcl} E & ::= & n \\ & | & E\ +\ E \\ & | & E\ *\ E \\ & | & (\ E\ ) \end{array}$$

Cette grammaire est ambiguë et génère des conflits quelle que soit la technique d'analyse. Mais n'est-ce pas celle-ci que vous souhaiteriez utiliser en pratique ?

Nous allons voir qu'il est possible d'adjoindre aux grammaires une notion de priorité entre opérateurs, qui reflèterait dans notre exemple les conventions d'écriture des mathématiques, et notamment la priorité de la multiplication sur l'addition. Cela permet de régler les conflits et d'autoriser l'analyse avec des outils comme `menhir`, *sans torturer la grammaire.*

Détection d'un conflit. Considérons la grammaire simplifiée des expressions arithmétiques avec uniquement l'addition. On y explicite en revanche le symbole de fin d'entrée pour bien suivre la construction de l'automate associé.

$$
\begin{array}{lcl}
S & ::= & E\,\# \\
E & ::= & n \\
 & | & E\ \texttt{+}\ E \\
 & | & \texttt{(}\ E\ \texttt{)}
\end{array}
$$

Traduisons cette grammaire en `menhir`, avec le fichier minimal suivant (ici on ne reconstruit aucun arbre de syntaxe, on a simplement l'ossature de la grammaire).

```
%token INT PLUS LPAR RPAR EOF
%start prog
%type <unit> prog
%%

prog:
| expr EOF {}

expr:
| INT            {}
| expr PLUS expr {}
| LPAR expr RPAR {}
```

Le bilan donné par `menhir` est le suivant.

```
Warning : one state has shift/reduce conflicts.
Warning : one shift/reduce conflict was arbitrarily resolved.
```

Ce bilan mentionne un *état*, qui est un état de l'automate d'analyse LR(1), dans lequel apparaît un conflit entre une opération de progression et une opération de réduction. Pour en savoir plus sur ce conflit, on peut consulter l'automate d'analyse LR(1) construit par `menhir`. On peut observer cet automate dans un fichier `.automaton` produit par menhir lorsque l'outil est utilisé avec l'option `-v` (*verbose*).

Dans notre exemple, on y trouve par exemple la description suivante de l'un des états.

```
State 4:
## Known stack suffix:
## expr PLUS
## LR(1) items:
expr -> expr PLUS . expr [ PLUS RPAR EOF ]
## Transitions:
-- On LPAR shift to state 1
-- On INT shift to state 2
-- On expr shift to state 5
## Reductions:
```

On y voit notamment un numéro pour l'état, un ensemble d'éléments nécessairement présents au sommet de la pile et une règle de réduction $E \prec E$+E en cours de reconnaissance. Le point entre le symbole `PLUS` et la deuxième occurrence de `expr` correspond au niveau de progression dans la règle (c'est le symbole • des états des automates LR) et les trois symboles entre crochets `[ PLUS RPAR EOF ]` correspondent aux symboles suivants possibles (il s'agit de l'ajout de LR(1) par rapport à LR(0)). La description termine par la liste des transitions et des réductions possibles en fonction du prochain lexème de l'entrée.

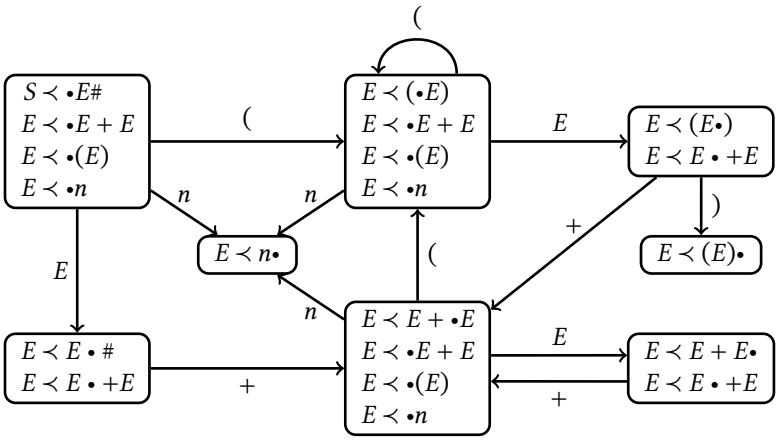
Par comparaison, voici le descriptif donné pour l'état comportant un conflit.

```
State 5:
## Known stack suffix:
## expr PLUS expr
## LR(1) items:
expr -> expr . PLUS expr [ PLUS RPAR EOF ]
expr -> expr PLUS expr . [ PLUS RPAR EOF ]
## Transitions:
-- On PLUS shift to state 4
```

```
## Reductions:
-- On PLUS RPAR EOF
--   reduce production expr -> expr PLUS expr
** Conflict on PLUS
```

Lorsque le prochain lexème de l'entrée est `PLUS`, on y voit deux opérations possibles : une transition vers l'état 4 (opération de progression) ou une réduction de la règle $E \prec E{+}E$.

Voici une vision plus complète de cet automate (on ne fait apparaître ici que les informations LR(0), qui sont suffisantes pour cet exemple exemple précis). Le conflit apparaît dans l'état en base à droite, dans lequel il est possible de progresser avec le symbole `+`, mais aussi de réduire la règle $E \prec E{+}E$.



Analyse d'un conflit. En plus de cette description de l'automate, `menhir` produit un fichier `.conflicts` donnant des détails sur les enjeux de chacun des conflits détectés. Détaillons le contenu de ce fichier pour notre exemple. On a d'abord une identification de l'état, du prochain lexème de l'entrée pour lequel le conflit existe, et d'un état de la pile correspondant à cet état (ce dernier point peut être vu comme un chemin dans l'automate).

```
** Conflict (shift/reduce) in state 5.
** Token involved: PLUS
** This state is reached from prog after reading:

expr PLUS expr
```

Le rapport décrit ensuite les arbres de dérivation correspondant à l'un ou l'autre choix parmi les différentes opérations possibles. Ces arbres ont d'abord un préfixe commun décrit par

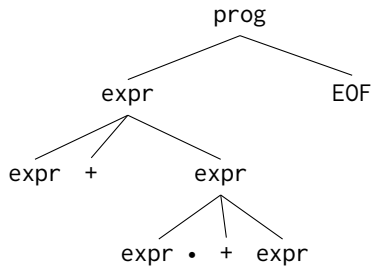
```
** The derivations that appear below have the following common
** factor: (The question mark symbol (?) represents the spot where
** the derivations begin to differ.)

prog
expr EOF
(?)
```

que nous pouvons traduire par le schéma

Les deux actions possibles (progression ou réduction), correspondent alors à deux formes différentes du sous-arbre noté par un point d'interrogation. Le rapport détaille d'abord le cas de la progression.

```
** In state 5, looking ahead at PLUS, shifting is permitted
** because of the following sub-derivation:

expr PLUS expr
          expr . PLUS expr
```

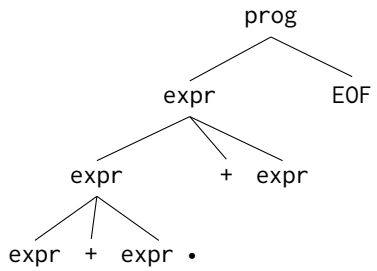
Dans ce cas le prochain symbole `PLUS` est interprété comme faisant partie de l'opérande droit de la première opération. Ceci correspondrait à un arbre de dérivation complété ainsi, où le point • désigne le stade de progression dans la lecture de l'entrée.



Enfin, on a une description de l'action de réduction.

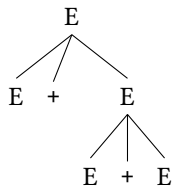
```
** In state 5, looking ahead at PLUS, reducing production
** expr -> expr PLUS expr
** is permitted because of the following sub-derivation:

expr PLUS expr // lookahead token appears
expr PLUS expr .
```

Cette fois, on considère que le motif $E+E$ déjà complété est une expression à part entière, formant le premier opérande de l'opération d'addition associée au prochain lexème. Ceci correspond à l'arbre suivant.

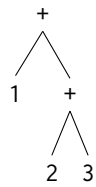


L'heure du choix. Bilan de ce rapport : à un tel stade de l'analyse de notre entrée, nous avons un choix entre progresser avec le symbole +, ou réduire avec la règle $E \prec E{+}E$.

— Progresser mène à un arbre de dérivation

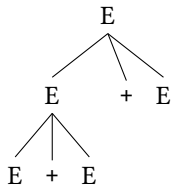


Sur l'entrée concrète `1+2+3` nous aurions l'arbre de syntaxe

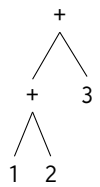


correspondant au parenthésage `1+(2+3)`.

— Réduire mène à un arbre de dérivation



Sur l'entrée concrète `1+2+3` nous aurions l'arbre de syntaxe



correspondant au parenthésage `(1+2)+3`.

Il faut alors déterminer laquelle de ces deux interprétations est « la bonne », puis indiquer en conséquence à l'outil s'il doit choisir de progresser ou de réduire dans cette situation.

Associativités. En l'occurrence, on va opter pour un parenthésage implicite à gauche, c'est-à-dire favoriser `(1+2)+3`. On déclare pour cela l'opérateur `PLUS` comme « associatif à gauche ». Il suffit pour cela d'inclure après la déclaration des symboles terminaux la précision.

```
%left PLUS
```

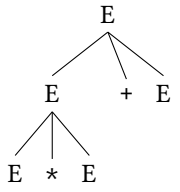
Conflits avec plusieurs symboles. D'autres conflits plus variés vont intervenir lorsque nous aurons plusieurs symboles dans la grammaire. En ajoutant la multiplication `*`, représentée avec un symbole terminal `STAR` dans `menhir`, on obtient trois nouveaux conflits :

— réduire `expr STAR expr` ou progresser avec `STAR`,
— réduire `expr STAR expr` ou progresser avec `PLUS`,
— réduire `expr PLUS expr` ou progresser avec `STAR`.

Le premier cas est similaire au précédent, et est relatif à l'associativité de la multiplication (on déclarera donc encore la multiplication comme associative à gauche). Les deux autres en revanche dépendent des priorités relatives données aux deux opérations d'addition et de multiplication.

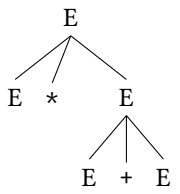
Si on a reconnu `expr STAR expr` est que le prochain symbole est `PLUS` on peut :

— soit réduire et s'orienter vers un arbre de dérivation de la forme



c'est-à-dire interpréter `2*3+4` comme `(2*3)+4`,

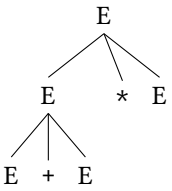
— soit progresser et s'orienter vers un arbre de dérivation de la forme



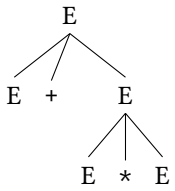
c'est-à-dire interpréter `2*3+4` comme `2*(3+4)`.

La solution cohérente avec les conventions mathématiques usuelles est la première : il faut donc dans ce cas favoriser la réduction.

À l'inverse, si on a reconnu `expr PLUS expr` et que le prochain symbole est `STAR` :
— on peut réduire et s'orienter vers un arbre de dérivation de la forme



c'est-à-dire interpréter `2+3*4` comme `(2+3)*4`,
— ou on peut progresser et s'orienter vers un arbre de dérivation de la forme



c'est-à-dire interpréter `2+3*4` comme `2+(3*4)`.
La solution cohérente avec les conventions mathématiques usuelles est cette fois la seconde : il faut donc dans ce cas favoriser la progression.

Déclaration de priorités. Point commun à ces deux conflits : on veut favoriser l'opération relative à l'opérateur `STAR` par rapport à l'opération relative à l'opérateur `PLUS`. Autrement dit, l'opérateur de multiplication doit être *plus prioritaire* que l'opérateur d'addition.

On déclare cela en `menhir` en plaçant les informations d'associativité de `PLUS` et `STAR` sur deux lignes différentes. La règle est la suivante : on donne les informations d'associativité par ordre de priorité, en commençant par les opérateurs les moins prioritaires. Nous avons donc ici les deux lignes

```
%left PLUS
%left STAR
```

Ces déclarations concernent toutes les utilisations des symboles concernés. Il est courant qu'ajouter un seul symbole à une liste de priorité déjà établie règle plusieurs conflits à la fois. Cependant, on a aussi parfois des situations dans lesquelles un même symbole peut apparaître dans plusieurs règles, et avec des priorités différentes.

Le symbole `-` par exemple peut apparaître dans deux situations :
— en tant qu'opérateur binaire, pour l'opération de soustraction, comme dans `1 - 2`,
— en tant qu'opérateur unaire, pour l'opération « opposé », comme dans `-1`.
Dans l'expression `-2 * 3 - 4 * 5` ce symbole apparaît ainsi deux fois, une fois dans chacune des deux situations. Le parenthésage implicite conventionnel est alors `((-2)*3) - (4*5)`. Autrement dit, le symbole `-` est plus prioritaire que la multiplication lorsqu'il représente une opération unaire, et moins prioritaire que la multiplication lorsqu'il représente une opération binaire.

Pour résoudre cela, on peut introduire dans `menhir` deux symboles : un symbole terminal normal `MINUS` désignant `-`, et un symbole terminal fantôme `U_MINUS` utilisé seulement pour marquer la priorité du `-` unaire. La déclaration complète pour cette solution est comme suit, avec utilisation du symbole `MINUS` dans les règles, et ajout d'une déclaration de priorité indiquant que la dernière règle prend la priorité du symbole `U_MINUS` plutôt que celle du symbole `MINUS` apparaissant dans la règle.

```
%token PLUS STAR MINUS U_MINUS

%left PLUS MINUS
%left STAR
%nonassoc U_MINUS
%%

expr:
| e1=expr STAR  e2=expr { Mul(e1, e2) }
| e1=expr MINUS e2=expr { Sub(e1, e2) }
| MINUS e=expr          { Opp(e)      } %prec U_MINUS
```

Notez que le symbole `U_MINUS` est déclaré comme « non-associatif » plutôt qu'associatif à gauche comme les autres, puisque l'associativité n'a pas de sens pour une telle opération unaire. Pour compléter, si l'on avait voulu un opérateur associatif à droite, on aurait pu utiliser la directive `%right`.

Interprétation des priorités. Formellement, les règles pour choisir entre progression et réduction en utilisant les priorités des symboles sont les suivantes. D'abord, chaque opération se voit affecter une priorité :

— pour une opération de progression sur un symbole $a \in T$, la priorité est celle de ce symbole,
— pour une opération de réduction d'une règle $X \prec \beta$, la priorité est celle du symbole terminal le plus à droite dans la séquence β.

La résolution est alors séparée en deux cas :

1. si les priorités sont différentes, alors on réalise l'opération la plus prioritaire,
2. sinon on utilise l'associativité : on favorise la réduction pour des opérateurs associatifs à gauche, et la progression pour des opérateurs associatifs à droite.

Notez que deux opérateurs ont la même priorité s'ils sont sur la même ligne. Dans ce cas ils ont aussi la même associativité, puisqu'ils suivent la même indication `%left` ou `%right`.

Bilan. En utilisant ces notions de priorité, on peut régler de nombreux conflits, et même des ambiguïtés de la grammaire, et cela sans défigurer la grammaire elle-même. Pour choisir entre les différentes possibilités en revanche, il faut analyser les différents arbres de dérivation correspondants pour trouver ceux qui correspondent à l'interprétation voulue.

7.4 Approfondissement : analyse syntaxique de IMP

Illustrons l'utilisation conjointe de `ocamllex` et `menhir` en construisant un nouvel analyseur syntaxique pour le langage IMP. On organise ce programme en quatre parties :

— un module `Imp` (fichier `imp.ml`) définit l'AST et d'éventuelles fonctions auxiliaires de manipulation de la syntaxe abstraite,
— un module `Impparser` réalisé avec `menhir` (fichier `impparser.mly`) définit les lexèmes et l'analyseur syntaxique,
— un module `Implexer` réalisé avec `ocamllex` (fichier `implexer.mll`) définit l'analyseur lexical,
— un module principal `Impc` (fichier `impc.ml`) définit le programme principal, qui analyse un fichier fourni en entrée.

On conserve la même syntaxe concrète qu'au chapitre 3, avec quelques variations dans la liste des opérateurs et les possibilités nouvelles d'inclure des commentaires ou d'omettre la branche secondaire **`else`** dans une instruction de branchement.

Syntaxe abstraite. Définition de l'AST (fichier `imp.ml`).

```
type bop = Add | Sub | Mul | Lt | Le
type expr =
  | Cst of int
  | Var of string
  | Bop of bop * expr * expr
type instr =
  | Print of expr
  | Set   of string * expr
  | While of expr * instr list
  | If    of expr * instr list * instr list
type prog = instr list
```

Analyse syntaxique. Définition de l'analyseur syntaxique avec menhir (fichier `impparser.mly`). On a un prélude minimal, qui se contente de charger le module `Imp` de syntaxe abstraite et un module utilitaire `Lexing` lié à l'analyse lexicale, suivi immédiatement de la définition des lexèmes.

```
%{
  open Imp
  open Lexing
%}
%token <int> INT
%token <string> IDENT
%token PLUS MINUS STAR LT LE
%token LPAR RPAR BEGIN END SEMI
```

```
%token PRINT SET IF ELSE WHILE
%token EOF
```

La grammaire que l'on souhaiterait comporte trois symboles non terminaux : P pour un programme complet, I pour une instruction et E pour une expression. Les règles sont ensuite :

$$
\begin{array}{rcl}
P & ::= & I^\star \ \# \\
I & ::= & x \texttt{ := } E \texttt{ ;} \\
 & | & \texttt{while (} E \texttt{) \{ } I^\star \texttt{ \}} \\
 & | & \texttt{if (} E \texttt{) \{ } I^\star \texttt{ \} [else \{ } I^\star \texttt{ \}]?} \\
 & | & \texttt{print (} E \texttt{) ;} \\
E & ::= & n \\
 & | & x \\
 & | & \texttt{(} E \texttt{)} \\
 & | & E \ o \ E \qquad \text{avec } o \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{<}, \texttt{<=}\}
\end{array}
$$

où # désigne un symbole terminal spécial de fin de fichier, n une constante entière, x un identifiant de variable, $I^\star$ une succession d'un nombre quelconque d'instructions, et [else { $I^\star$ }]? la présence optionnelle d'une branche secondaire dans une conditionnelle. On souhaite en outre que les conventions de priorité usuelles soient respectées.

On peut déjà définir un symbole de départ `prog` et son type.

```
%start prog
%type <Imp.prog> prog
%%
```

Pour écrire la règle associée en revanche, il faut pouvoir reconnaître une séquence d'instructions $I^\star$. Une première possibilité consiste à introduire un symbole `seq` représentant une séquence, en suivant la même grammaire qu'au chapitre 3. Ces règles se traduisent directement en `menhir`, et on peut renvoyer la liste des instructions reconnues. La règle reconnaissant un programme entier s'ensuit immédiatement.

$$
\begin{array}{rcl}
S & ::= & \varepsilon \\
 & | & \text{I S}
\end{array}
$$

```
seq:
| i=instr s=seq { i :: s }
| (* empty *)   { []     }
;
```

À noter : pour une règle de la forme $S \prec \varepsilon$ produisant une séquence vide, il suffit de ne rien écrire dans le motif à réduire !

Avec ce symbole supplémentaire `seq`, on peut donner des règles de reconnaissance pour les instructions et pour un programme entier, qui suivent directement la grammaire. Seule variation : on prévoit deux versions de la règle pour `if`, avec ou sans branche `else`.

```
instr:
| PRINT LPAR e=expr RPAR SEMI            { Print(e)     }
| id=IDENT SET e=expr SEMI               { Set(id, e)   }
| WHILE LPAR e=expr RPAR BEGIN s=seq END { While(e, s)  }
| IF LPAR e=expr RPAR BEGIN s=seq END    { If(e, s, []) }
| IF LPAR e=expr RPAR BEGIN s1=seq END
                ELSE BEGIN s2=seq END   { If(e, s1, s2) }
;

prog:
| s=seq EOF { s }
```

Pour rapporter à l'utilisateur un minimum d'information en cas d'erreur de syntaxe dans le programme analysé, on ajoute à `prog` une règle d'erreur, qui récupère la position à laquelle l'analyse a échoué avec `$startpos` (une valeur spéciale de `menhir`) et qui traduit cette position en un numéro de ligne et un numéro de colonne (voir le module `Lexing` de caml pour le format des positions).

```
| error
    { let pos = $startpos in
      let message = Printf.sprintf "echec␣a␣la␣position␣%d,␣%d"
                      pos.pos_lnum (pos.pos_cnum - pos.pos_bol)
      in
      failwith message }
;
```

Les règles pour les expressions suivent à nouveau de près la grammaire. On utilise un symbole supplémentaire `bop` pour factoriser les différentes opérations binaires.

```
expr:
| n=INT                  { Cst(n)  }
| id=IDENT               { Var(id) }
| LPAR e=expr RPAR       { e       }
| e1=expr op=bop e2=expr { Bop(op, e1, e2) }
;
```

On complète le fichier `impparser.mly` avec la définition des opérateurs binaires.

```
%inline bop:
| PLUS  { Add }
| MINUS { Sub }
| STAR  { Mul }
| LT    { Lt  }
| LE    { Le  }
;
```

À noter la directive `%inline`, qui demande à `menhir` d'expanser le symbole non terminal `bop` à chaque endroit où il apparaît, c'est-à-dire de remplacer la règle

```
| e1=expr op=bop e2=expr  { Bop(op, e1, e2) }
```

par les cinq règles suivantes.

```
| e1=expr PLUS  e2=expr   { Bop(Add, e1, e2) }
| e1=expr MINUS e2=expr   { Bop(Sub, e1, e2) }
| e1=expr STAR  e2=expr   { Bop(Mul, e1, e2) }
| e1=expr LT    e2=expr   { Bop(Lt,  e1, e2) }
| e1=expr LE    e2=expr   { Bop(Le,  e1, e2) }
```

Question. Mais pourquoi faire cela alors que nous avions justement factorisé ? Vous pouvez essayer de compiler l'analyseur entier sans ce mot clé et d'interpréter ce qui se passe alors.

Ceci étant fait, reste à intercaler après la déclaration des lexèmes les déclarations des priorités des différents opérateurs pour assurer l'absence de conflits. On propose ici les suivantes.

```
%left LT LE
%left PLUS MINUS
%left STAR
```

Quelques bonus de `menhir`. L'analyseur que nous venons de construire peut être réalisé dans n'importe quel outil de la famille YACC. Avec `menhir`, on a en prime quelques raccourcis possibles.

Par exemple, `menhir` fournit une primitive `list` pour la reconnaissance d'une séquence, qui renvoie une liste caml. On peut se passer complètement de notre symbole `seq` et de ses règles, en remplaçant chacune de ses utilisations par `list(instr)`. Par exemple :

```
prog:
| s=list(instr) EOF  { s }
```

On a également sur le même modèle : `nonempty_list` et `separated_list`.

Autre exemple : `menhir` fournit également une primitive `option` pour la reconnaissance d'un élément optionnel. Cette primitive renvoie une option caml (`None` si l'élément optionnel n'est pas présent, `Some` *s* sinon). Cette primitive a également une variante `loption`, applicable lorsque l'élément optionnel est une liste, qui renvoie directement la liste reconnue, ou simplement `[]` si la liste optionnelle n'est pas présente. On pourrait donc introduire une nouvelle règle `else_branch` décrivant la branche secondaire d'un **`if`**, et avoir ensuite une seule règle pour **`if`**.

```
else_branch:
| ELSE BEGIN s=list(instr) END { s }
;

instr:
| IF LPAR c=expr RPAR BEGIN s1=list(instr) END
                      s2=loption(else_branch) { If(c, s1, s2) }
...
```

Pour compacter encore un peu, l'unique règle du symbole `else_branch` peut être placée directement en argument de la primitive `loption`, ce qui évite d'avoir à définir ce symbole supplémentaire.

```
instr:
| IF LPAR c=expr RPAR BEGIN s1=list(instr) END
    s2=loption(ELSE BEGIN s=list(instr) END {s}) { If(c, s1, s2) }
```

Analyse lexicale. On réalise un analyseur lexical avec ocamllex (fichier `implexer.mll`). Notez que le prélude charge le module `Impparser`, puisque c'est ce dernier qui définit les lexèmes que l'analyse lexicale doit produire.

```
{
  open Lexing
  open Impparser
```

Pour ne pas écrire une règle particulière pour chaque mot-clé du langage, on fait une table des mots clés et des lexèmes associés et on définit une fonction qui utilise cette table pour interpréter les mots-clés et les identifiants.

```
  let keyword_or_ident =
  let h = Hashtbl.create 17 in
  List.iter (fun (s, k) -> Hashtbl.add h s k)
    [ "print",  PRINT;
      "if",     IF;
      "else",   ELSE;
      "while",  WHILE;
    ] ;
  fun s ->
    try  Hashtbl.find h s
    with Not_found -> IDENT(s)
}
```

Le reste de l'analyseur est ensuite conforme à ce que nous avons pu voir au début du chapitre. On commence par définir des abbréviations pour reconaître les constantes et les identifiants.

```
let digit = ['0'-'9']
let number = ['-']? digit+
let alpha = ['a'-'z' 'A'-'Z']
let ident = ['a'-'z' '_'] (alpha | '_' | digit)*
```

On introduit ensuite les règles.

La fonction `new_line` est définie dans la bibliothèque `Lexing`. Appelée à chaque reconnaissance d'un caractère de retour à la ligne, elle permet d'obtenir des messages d'erreurs identifiant la ligne du fichier concernée.

```
rule token = parse
  | ['\n']              { new_line lexbuf; token lexbuf }
  | [' ' '\t' '\r']+    { token lexbuf                  }
  | "//" [^ '\n']* "\n" { new_line lexbuf; token lexbuf }
  | "/*"                { comment lexbuf; token lexbuf  }
  | number as n         { INT(int_of_string n)          }
  | ident as id         { keyword_or_ident id           }
  | ";"  { SEMI  }
  | ":=" { SET   }
  | "+"  { PLUS  }
  | "-"  { MINUS }
  | "*"  { STAR  }
  | "<"  { LT    }
  | "<=" { LE    }
  | "("  { LPAR  }
  | ")"  { RPAR  }
  | "{"  { BEGIN }
  | "}"  { END   }
  | _    { failwith ("unknown␣character␣:␣" ^ (lexeme lexbuf)) }
  | eof  { EOF   }

and comment = parse
  | "*/" { () }
  | _    { comment lexbuf }
  | eof  { failwith "unfinished␣comment" }
```

Programme principal. Enfin, le module principal (fichier `impc.ml`) récupère un nom de fichier en ligne de commande, et analyse ce fichier.

```
let () =
  let file = Sys.argv.(1) in
  let c = open_in file in
  let lexbuf = Lexing.from_channel c in
  let ast = Impparser.prog Implexer.token lexbuf in
  close_in c;
  ignore(ast);
  exit 0
```

Notez la fonction d'analyse syntaxique `Impparser.prog` prend en paramètre la fonction d'analyse lexicale `Implexer.token` : elle va l'utiliser pour aller consulter les prochains lexèmes à chaque fois que nécessaire. L'analyse lexicale produit un arbre de syntaxe abstraite dont on ne fait rien ici. Dans un vrai projet la ligne `ignore(ast);` a vocation à être remplacée par ce que l'on souhaite faire avec ce programme. Par exemple : l'interpréter, ou l'analyser, ou le compiler, etc.

7.5 Approfondissement : utilisation de LEX à d'autres fins que l'analyse lexicale

Au-delà de l'analyse lexicale, les outils de la famille LEX sont utiles pour réaliser tout programme analysant un texte (chaîne de caractères, fichier, flux...) sur la base d'expressions régulières, ou transformant un texte par une série de modifications locales relativement simples.

Nettoyage d'un texte. Par exemple : les 6 lignes suivantes définissent en `ocamllex` un programme complet, qui récupère un texte sur l'entrée standard et produit sur la sortie standard le même texte dans lequel les lignes vides consécutives sont ignorées.

```
rule scan = parse
  | '\n' '\n'+  { print_string "\n\n"; scan lexbuf }
  | _ as c      { print_char c; scan lexbuf }
  | eof         { () }

{ let _ = scan (Lexing.from_channel stdin) }
```

En supposant que ces 6 lignes forment un fichier `mbl.mll`, on fabrique l'exécutable avec les deux lignes de commande

```
# ocamllex mbl.mll
# ocamlopt -o mbl mbl.ml
```

et on l'utilise pour lire un fichier `infile` et écrire le résultat dans un fichier `outfile` avec la ligne de commande

```
# ./mbl < infile > outfile
```

Statistiques. Deuxième exemple simple : le programme défini par le code `ocamllex` suivant prend en paramètres sur la ligne de commande un mot et un nom de fichier, et affiche le nombre d'occurrences du mot dans le fichier.

```
{
  let word = Sys.argv.(1)
  let count = ref 0
}

rule scan = parse
  | ['a'-'z' 'A'-'Z']+ as w  { if word = w then incr count;
                               scan lexbuf }
  | _    { scan lexbuf }
  | eof  { () }

{
  let () = scan (Lexing.from_channel (open_in Sys.argv.(2)))
  let () = Printf.printf "%d␣occurrence(s)\n" !count
}
```

Embellisseur de code. Regardons maintenant un exemple plus élaboré (repris à Jean-Christophe Filliâtre) : un utilitaire `caml2html` qui produit un code html pour l'affichage dans un navigateur d'une version embellie d'un code source caml donné en entrée. On se donne les objectifs suivants :

- la commande `caml2html file.ml` produit un fichier `file.ml.html`
- les mots-clés apparaissent en violet, les commentaires en orange
- les lignes sont numérotées

On commence, dans le prélude, par vérifier les paramètres donnés sur la ligne de commande, ouvrir le fichier `.html` cible et définir une fonction auxiliaire écrivant dans ce fichier.

```
{
  let () =
    if Array.length Sys.argv <> 2
      || not (Sys.file_exists Sys.argv.(1))
    then begin
      Printf.eprintf "usage:␣caml2html␣file\n";
      exit 1
    end

  let file = Sys.argv.(1)
  let cout = open_out (file ^ ".html")
  let print s = Printf.fprintf cout s
```

On ajoute une référence et une fonction auxiliaire pour numéroter les lignes, qu'on appelle une première fois pour créer la première ligne.

```
  let count = ref 0
  let newline() = incr count; print "\n%3d:␣" !count
```

Enfin, on introduit une définition conjointe d'une table des mots-clés à colorer, et d'une fonction auxiliaire `is_keyword` identifiant ces mots-clés.

```
  let is_keyword =
    let ht = Hashtbl.create 64 in
    List.iter
      (fun s -> Hashtbl.add ht s ())
      [ "fun"; "let"; "rec"; "and"; "in"; "match"; "with";
        "begin"; "end"; (* à compléter *) ];
    fun s -> Hashtbl.mem ht s
}
```

Les fonctions **print**, `newline` et `is_keyword` pourront être appelées par les fonctions principales d'analyse.
Après ce prélude, on définit une unique expression régulière auxiliaire, pour les identifiants.

```
let ident =
  ['A'-'Z' 'a'-'z' '_'] ['A'-'Z' 'a'-'z' '0'-'9' '_']*
```

On peut ensuite passer à la fonction principale de traitement du texte, qui copie le contenu du fichier source en ajoutant les différents embellissements à l'aide de balises html.

```
rule scan = parse
```

Dans le cas d'un identificateur, on teste s'il s'agit d'un mot-clé à l'aide de notre fonction auxiliaire `is_keyword`. On l'affiche alors avec ou sans coloration selon son statut avant de poursuivre la lecture.

```
  | ident as s  {
      if is_keyword s then
        print "<font␣color=\"violet\">%s</font>" s
      else
        print "%s" s;
      scan lexbuf
    }
```

À chaque saut de ligne, on invoque la fonction `newline` pour déclencher le passage à la ligne et la numérotation.

```
  | "\n"  { newline(); scan lexbuf }
```

Lorsque s'ouvre un commentaire, on change de couleur et on invoque une autre fonction d'analyse `comment`. Lorsque cet autre analyseur rend la main, on revient à la couleur par défaut et on continue.

```
  | "(*"  {
      print "<font␣color=\"orange\">(*";
      comment lexbuf;
      print "</font>";
      scan lexbuf
    }
```

L'idée serait ensuite de copier tel quel dans le fichier cible tout caractère ne correspondant pas à l'un des cas particuliers déjà traités. Dans un tel analyseur, on a cependant encore besoin de quelques traitement particuliers, pour des caractères réservés de html qui doivent être échappés.

```
  | "<"  { print "&lt;"; scan lexbuf }
  | "&"  { print "&amp;"; scan lexbuf }
```

Les chaînes doivent de même être traitées à part, pour des raisons d'échappement comme d'habitude. La lecture d'un guillement appellerait donc un nouvel analyseur dédié à l'aide d'une ligne comme

```
  | '"'  { print "\""; string lexbuf; scan lexbuf }
```

Il faudrait alors écrire cet autre analyseur `string`. À noter que le traitement complet des chaînes demande aussi d'autres petits ajustements dans `scan` et dans `comment`. Ne pas oublier de traiter le cas de guillemets qui ne délimitent pas une chaîne de caractères. Ne pas oublier également que certains guillemets peuvent apparaître échappés dans le texte source.

Revenons à notre analyseur principal : une fois traités les cas particuliers, il ne reste plus qu'à effectivement copier tout caractère autre, et à traiter la fin du fichier.

```
  | _ as c  { print "%c" c; scan lexbuf }
  | eof     { () }
```

On peut maintenant définir l'analyseur auxiliaire `comment` pour les commentaires. La couleur ayant déjà été configurée, cet analyseur affiche tout tel quel jusqu'à la fin du commentaire, en prenant bien garde à continuer à tenir compte des changements de ligne.

```
and comment = parse
  | "*)"    { print "*)" }
  | eof     { () }
  | "\n"    { newline(); comment lexbuf }
  | _ as c  { print "%c" c; comment lexbuf }
```

On ajoute une dernière règle pour traiter correctement les commentaires imbriqués. À la lecture de la séquence `(*`, on l'affiche bien comme d'habitude, mais il faut ensuite faire en sorte que la prochaine occurrence de `*)` ne ferme que le commentaire interne, et n'arrête pas la coloration avant que le commentaire principal ait bien été terminé. On réalise ceci en enchaînement deux appels à notre fonction `comment` : le premier pour analyser le commentaire interne, le deuxième pour reprendre l'analyse du commentaire externe.

```
  | "(*"  { print "(*"; comment lexbuf; comment lexbuf }
```

L'épilogue enfin va écrire dans le fichier `.html` cible, d'abord l'entête html, puis la copie embellie du code du fichier source, et enfin la conclusion html.

```
{
  let _ =
    print "<html><head><title>%s</title></head><body>\n<pre>" file;
    newline();
    scan (Lexing.from_channel (open_in file));
    print "</pre>\n</body></html>\n";
    close_out cout
}
```

Troisième partie

Programmation impérative

Plan

Quatrième partie

Programmation objet

Plan