Langages, interprétation, compilation

Thibaut Balabonski @ Université Paris-Saclay
http://www.lri.fr/~blsk/CompilationLDD3/
v1.0, automne 2021
Première partie

Table des matières

1 À propos d'une calculatrice

Tour d'horizon où l'on réalise des fonctions d'interprétation et de compilation pour des expressions arithmétiques.

1.1 Panorama

Supposons que l'on s'intéresse à l'expression arithmétique

```
(1+23*456+78)*9
```

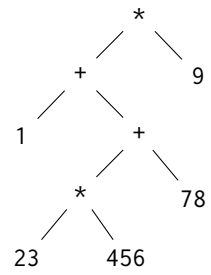
Une fois saisie sur le clavier d'une calculatrice ou d'un ordinateur, cette expression prend la forme d'une chaîne de caractères, formée de la suite de caractères suivante.

```
(, 1, +, 2, 3, *, 4, 5, 6, +, 7, 8, ), *, 9
```

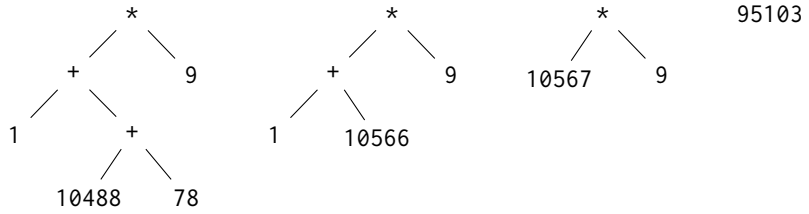
Cette séquence est plate, et ses éléments atomisés. Nous allons devoir l'analyser pour reconstruire sa structure et la manipuler réellement comme une expression arithmétique et non comme une simple suite de symboles. La première étape, appelée ***analyse lexicale***, consiste à regrouper les symboles formant ensemble un même élément. On obtient par exemple le découpage suivant.

```
(, 1, +, 23, *, 456, +, 78, ), *, 9
```

La deuxième étape, appelée ***analyse syntaxique***, consiste à associer correctement les différentes parties de l'expression aux bons opérateurs. Le résultat peut être présenté sous la forme d'un arbre.



Si l'on souhaite ***interpréter*** cette expression, c'est-à-dire son résultat, on peut alors effectuer les différentes opérations en partant des feuilles de l'arbre et en remontant vers la racine.



Pour tout autre objectif, d'analyse ou de compilation de notre expression, on peut de même travailler en se laissant guider par la structure de cet arbre.

1.2 Lexèmes et analyse lexicale

Faisons la liste des éléments pouvant apparaître dans l'écriture d'une expression arithmétique :

— des nombres (on prendra des entiers positifs),
— des opérateurs (on prendra `+` et `*`),
— des parenthèses ouvrantes ou fermantes.

On peut définir un type Caml permettant de décrire de tels éléments, qu'on appellera ici des ***mots*** (en jargon des ***lexèmes***, ou ***token*** en anglais).

```
type mot =
  | Nombre of int
  | Plus
  | Fois
  | ParO
  | ParF
```

L'analyse lexicale peut donc être vue comme une fonction prenant une chaîne de caractères et renvoyant une liste de mots.

```
let analyse_lex (e: string): mot list =
```

Pour itérer sur les différents caractères de la chaîne, on introduit une fonction auxiliaire `loop` prenant en paramètre l'indice `i` du caractère courant.

```
  let rec lex i =
    if i >= String.length e then
      []
```

Si l'indice `i` n'a pas dépassé le dernier caractère, on observe ce caractère courant. Dans le cas d'un caractère désignant à lui seul un mot, on combine ce mot et la liste obtenue en poursuivant la boucle à partir du caractère suivant.

```
    else match e.[i] with
      | '+' -> Plus :: (lex (i+1))
      | '*' -> Fois :: (lex (i+1))
      | '(' -> ParO :: (lex (i+1))
      | ')' -> ParF :: (lex (i+1))
```

Une espace est tout simplement ignorée : la boucle continue au caractère suivant et rien n'est ajouté à la liste des mots reconnus.

```
let analyse_lex (e: string): mot list =
  let rec lex i =
    if i >= String.length e then
      []
    else match e.[i] with
      | '+' -> Plus :: (lex (i+1))
      | '*' -> Fois :: (lex (i+1))
      | '(' -> ParO :: (lex (i+1))
      | ')' -> ParF :: (lex (i+1))
      | ' ' -> lex (i+1)
      | '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ->
        let i_fin = fin_nb i in
        let n = int_of_string (String.sub e i (i_fin-i)) in
        Nombre n :: lex i_fin
      | c -> failwith (Printf.sprintf "caractère␣inconnu␣:␣%c" c)
  and fin_nb i =
    if i >= String.length e then
      i
    else match e.[i] with
      | '0' | '1' | '2' | '3' | '4'| '5' | '6' | '7' | '8' | '9' ->
        fin_nb (i+1)
      | _ -> i
  in
  lex 0
```

Figure 1 – Analyse lexicale

```
      | ' ' -> lex (i+1)
```

Si le caractère courant est un chiffre, on va devoir lire une séquence de chiffres pour former un nombre. Pour gérer cela, on fait appel à une fonction auxiliaire `fin_nb` chargée de lire l'entrée jusqu'à la fin du nombre et de renvoyer l'indice `i_fin` du caractère situé immédiatement après. Le nombre lui-même est alors extrait de la chaîne, puis on reprend l'analyse à partir de l'indice `i_fin`.

```
      | '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ->
        let i_fin = fin_nb i in
        let n = int_of_string (String.sub e i (i_fin-i)) in
        Nombre n :: (lex i_fin)
```

Enfin, aucun autre caractère n'étant possible dans une expression arithmétique bien formée, on déclenche une erreur dans tout autre cas.

```
      | c -> failwith (Printf.sprintf "caractère␣inconnu␣:␣%c" c)
```

La deuxième fonction auxiliaire se contente de parcourir l'entrée tant qu'elle n'y lit que des chiffres. Elle renvoie l'indice courant lorsque la fin de l'entrée est atteinte ou lorsqu'elle lit un caractère autre qu'un chiffre.

```
  and fin_nb i =
    if i >= String.length e then
      i
    else match e.[i] with
      | '0' | '1' | '2' | '3' | '4'| '5' | '6' | '7' | '8' | '9' ->
        fin_nb (i+1)
      | _ -> i
```

Pour analyser la chaîne complète, il ne reste plus qu'à lancer notre boucle principale à partir du premier caractère.

```
  in
  lex 0
```

Le code complet est en figure 1.

Exercice 1.1. Étendre le type `mot` et l'analyseur pour permettre la reconnaissance :

1. d'autres opérateurs arithmétiques : `-`, `/`, `mod`, `**` ;
2. de nombres négatifs ;
3. de nombres à virgule.

Attention aux multiples utilisations de certains caractères.

Nous reviendrons sur l'analyse lexicale au chapitre 3. Parmi les questions qui se poseront : comment décrire les mots d'un langage ? quels algorithmes pour les reconnaître efficacement ? quels sortes de mots peuvent ou non être décrits et reconnus ? Ce dernier point recèle des ramifications théoriques surprenantes.

1.3 Arbres de syntaxe et interprétation

Revenons sur la structure d'une expression arithmétique : il ne s'agit pas tant d'une séquence linéaire de mots que d'une structure hiérarchique où chaque opérateur est associé à deux opérandes. Cette structure est non seulement hiérarchique mais aussi récursive, puisque chaque opérande est à nouveau une expression arithmétique obéissant à la même structure.

C'est ainsi que l'on représente une expression arithmétique sous la forme d'un arbre appelé ***arbre de syntaxe abstraite***. Chaque nœud de l'arbre est un symbole choisi parmi un ensemble prédéfini (ici, des symboles pour l'addition, la multiplication et les constantes entières positives). Notez que chaque nœud portant un nombre entier est une feuille de l'arbre (un nœud avec zéro fils), et que chaque nœud portant un symbole d'addition ou de multiplication a exactement deux fils. On dit que l'addition et la multiplication ont une ***arité*** de deux, et la constante entière une arité de zéro.

On définit en caml un type de données pour représenter ces arbres de syntaxe de la manière suivante.

```
type expr =
  | Cst of int
  | Add of expr * expr
  | Mul of expr * expr
```

Notez que pour garder un ensemble fini de symboles, on a introduit un seul constructeur `Cst` pour les constantes entières, auquel on associe un nombre entier. Ainsi l'expression $1+2\times3$ est représentée en caml par la valeur `Add(Cst 1, Mul(Cst 2, Cst 3))` (de type `expr`).

On définit naturellement des fonctions manipulant des expressions arithmétiques sous la forme de fonctions récursives sur ce type `expr`. Voici par exemple des fonctions `nb_cst` et `nb_op` comptant respectivement le nombre de constantes entières et le nombre d'opérateurs dans une expression donnée en paramètre.

```
let rec nb_cst = function
  | Cst _ -> 1
  | Add(e1, e2) -> nb_cst e1 + nb_cst e2
  | Mul(e1, e2) -> nb_cst e1 + nb_cst e2

let rec nb_op = function
  | Cst _ -> 0
  | Add(e1, e2) -> 1 + nb_cst e1 + nb_cst e2
  | Mul(e1, e2) -> 1 + nb_cst e1 + nb_cst e2
```

On peut de même définir une fonction `eval` prenant en paramètre une expression arithmétique (ou plus précisément une valeur caml de type `expr` représentant cet expression arithmétique) et renvoyant le résultat du calcul décrit par cette expression. Cette fonction particulièrement simple fait correspondre chaque symbole à sa signification arithmétique après avoir évalué récursivement les sous-expressions.

```
let rec eval = function
  | Cst n -> n
  | Add(e1, e2) -> eval e1 + eval e2
  | Mul(e1, e2) -> eval e1 * eval e2
```

Exercice 1.2. Étendre le type `expr` et la fonction `eval` pour intégrer les opérateurs arithmétiques et nombres de l'exercice 1.1.

Le chapitre 2 sera consacré au raisonnement sur les structures arborescentes telles que les arbres de syntaxe abstraite et à leur manipulation. Nous y programmerons des interprètes pour des langages de programmation et y verrons de nouvelles techniques de preuves.

1.4 Analyse syntaxique

Reste à savoir faire la transition entre la séquence de lexèmes produite par l'analyse lexicale et les arbres de syntaxe abstraite que l'on peut ensuite facilement manipuler à l'aide de fonctions récursives. Cette étape appelée ***analyse syntaxique*** doit regrouper les symboles formant ensemble des sous-expressions de l'expression principale, et combiner les sous-expressions avec les bons opérateurs.

On va utiliser ici l'algorithme de la gare de triage (*shunting yard*), qui lit la séquence de lexèmes de gauche à droite et stocke sur une pile les sous-expressions qui ont pu être formées avec les éléments déjà lus. Le principe est le suivant :

- les nombres sont directements transférés sur la pile des expressions, puisque chacun forme à lui seul une expression,
- les opérateurs sont placés en attente sur une pile auxiliaire d'opérateurs jusqu'à ce que leur opérande droit ait fini d'être analysé (comme on lit de gauche à droite, lorsque l'on rencontre un opérateur son opérande gauche a déjà été intégralement vu),
- lorsque l'on a au moins deux expressions et un opérateur au sommet de leurs piles respectives, ils sont regroupés pour former une seule expression, *sauf si le prochain opérateur de l'entrée est plus prioritaire que l'opérateur vu sur la pile d'opérateurs.*

En outre, une parenthèse ouvrante est placée sur la pile auxiliaire des opérateurs en attendant que la parenthèse fermante associée soit vue.

L'algorithme s'arrête lorsque l'entrée a été complètement lue, que la pile auxiliaire des opérateurs est vide et qu'il ne reste plus qu'une expression sur la pile principale des expressions, cette dernière étant le résultat de l'analyse.

Dans le code figure 2, chaque pile est représentée à l'aide d'une simple liste. Lors de l'analyse de l'expression `2*3+4*5`, les trois piles ont successivement les états suivants (en écriture simplifiée).

entree	ops	exprs
2 * 3 + 4 * 5	.	.
* 3 + 4 * 5	.	2
3 + 4 * 5	*	2
+ 4 * 5	*	3 2
+ 4 * 5	.	(2*3)
4 * 5	+	(2*3)
* 5	+	4 (2*3)
5	* +	4 (2*3)
.	* +	5 4 (2*3)
.	+	(4*5) (2*3)
.	.	((4*5)+(2*3))

Dans l'état

entree	ops	exprs
+ 4 * 5	*	3 2

on a deux expressions dans `exprs` et un opérateur de multiplication dans `ops`. La multiplication étant l'opération la plus prioritaire, on forme l'expression `2*3`. Dans l'état

entree	ops	exprs
* 5	+	4 (2*3)

en revanche, l'opération d'addition qui serait possible pour former l'expression `(4+(2*3))` n'est pas utilisée, l'opérateur d'addition au sommet de `ops` étant moins prioritaire que l'opérateur de multiplication en tête de `entree`.

Notez que dans le code de la figure 2, les premiers cas du **`match`** sont prioritaires sur les cas suivants.

```
let gare_de_triage l =
  let rec loop entree ops exprs =
    match entree, ops, exprs with
      | [], [], [e] -> e

      | Nombre n :: entree, _, _ ->
        loop entree ops (Cst n :: exprs)

      | ParO :: entree, _, _ ->
        loop entree (ParO :: ops) exprs

      | _, Fois :: ops, y :: x :: exprs ->
        loop entree ops (Mul(x, y) :: exprs)

      | Fois :: entree, _, _ ->
        loop entree (Fois :: ops) exprs

      | _, Plus :: ops, y :: x :: exprs ->
        loop entree ops (Add(x, y) :: exprs)

      | Plus :: entree, _, _ ->
        loop entree (Plus :: ops) exprs

      | ParF :: entree, ParO :: ops, _ ->
        loop entree ops exprs

      | _, _, _ -> failwith "expression␣mal␣formée"
  in
  (loop l [] [])
```

FIGURE 2 – Analyse syntaxique

Exercice 1.3. Étendre l'analyseur pour permettre la reconnaissance des nouvelles opérations introduites à l'exercice 1.1, avec les bonnes priorités.
Attention aux opérateurs comme - et /, qui ne sont pas associatifs.

Nous avons considéré ici les règles traditionnelles d'écriture d'une expression mathématique. Nous verrons au chapitre 4 comment décrire la syntaxe plus riche d'un langage de programmation, et les algorithmes et outils qui permettent de réaliser un analyseur syntaxique.

1.5 Compilation

Pour finir, nous allons traduire nos expressions arithmétiques (données par leur arbre de syntaxe abstraite) en séquences d'instructions élémentaires pour une machine virtuelle très simple utilisant un unique registre et stockant ses résultats intermédiaires sur une pile.

Imaginons donc une machine dotée de quatre instructions seulement, représentées par le type caml suivant.

```
type instr =
  | ICst of int (* charge une valeur entière *)
  | IPush (* stocke une valeur sur la pile *)
  | IAdd (* effectue une addition *)
  | IMul (* effectue une multiplication *)
```

La machine possède un unique registre de travail dans lequel elle stocke la valeur de l'expression qui vient d'être évaluée. En particulièer, `ICst(n)` stocke l'entier `n` dans ce registre. L'instruction `IPush` place la valeur stockée dans le registre au sommet de la pile. Les instructions `IAdd` et `IMul` effectuent respectivement une addition et une multiplication de la valeur stockée dans le registre et de la valeur placée au sommet de la pile (cette dernière étant au passage retirée de la pile), et leur résultat est stocké dans le registre.

Le résultat final de la machine est la valeur du registre une fois toutes les instructions exécutées. Voici une fonction simulant l'exécution d'une telle machine.

```
let exec p =
  let rec loop pile acc = function
    | [] -> acc
    | ICst n :: p -> loop pile n p
    | IPush :: p -> loop (acc :: pile) acc p
    | IAdd :: p -> loop (List.tl pile) (acc + List.hd pile) p
    | IMul :: p -> loop (List.tl pile) (acc * List.hd pile) p
  in
  loop [] 0 p
```

Notre objectif de *compilation* consiste alors à traduire une expression `e` (de type `expr`) en une liste d'instructions `l` (de type `instr list`), de sorte que l'exécution de la liste d'instruction `l` produise la valeur de l'expression d'origine.

Pour cela, on traduit d'abord une expression constante par une simple instruction `ICst`. Pour une addition ou une multiplication, on produit d'abord une liste d'instructions calculant la valeur de l'opérande gauche, on intercale ensuite une instruction `IPush` pour enregistrer cette valeur sur la pile avant d'évaluer l'opérande droit, et on termine par une instruction arithmétique combinant les deux valeurs obtenues.

```
let rec traduction = function
  | Cst n -> [ICst n]
  | Add(e1, e2) -> (traduction e1) @ [IPush] @ (traduction e2) @ [IAdd]
  | Mul(e1, e2) -> (traduction e1) @ [IPush] @ (traduction e2) @ [IMul]
```

Exercice 1.4. Étendre le type `instr` des instructions de la machine virtuelle pour permettre le traitement des nouveaux éléments introduits à l'exercice 1.1. Étendre en conséquence les fonctions `exec` et `traduction`.

Au chapitre 6 nous approfondirons les techniques de compilation d'un langage de programmation vers des instructions de bas niveau, qu'il s'agisse du code d'une machine virtuelle ou de code assembleur. Cette partie s'interfacera avec des questions d'architecture et de système.

2 Syntaxe abstraite et interprétation

Qu'est-ce qui, au-delà des détails de syntaxe ou d'implémentation, définit le cœur d'un langage de programmation ?

2.1 Syntaxe concrète et syntaxe abstraite

Notre calculatrice du chapitre précédent permettait d'écrire des expressions arithmétiques variées. Par exemple :

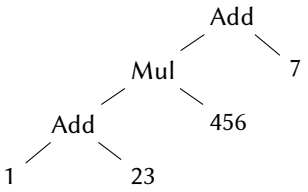
```
> (1+23)*456+7
> (1 + 23) * 456 + 7
> ( 1+23 ) *456 +7
```

Ces différentes écritures ont beau être différentes, elle représentent cependant la même expression arithmétique $(1 + 23) \times 456 + 7$.

Ainsi, on distingue deux niveaux de syntaxe pour un langage de programmation : la ***syntaxe concrète*** correspond à ce que le programmeur écrit, et la ***syntaxe abstraite*** à la structure du programme. La syntaxe concrète représente donc la partie visible du langage. La syntaxe abstraite est un outil central à la fois pour manipuler les programmes à l'intérieur du compilateur et pour raisonneer formellement sur les programmes.

Les deux niveaux de syntaxe sont généralement utilisés sous des formes différentes : la syntaxe concrète est utilisée comme un texte, une chaîne de caractères, tandis que la syntaxe abstraite est utilisée comme une représentation structurée, un arbre. En outre, la syntaxe abstraite tend à être plus centrée sur le cœur du langage et à s'abstraire de certains points annexes de l'écriture.

D'une part, certains éléments non-signifiants de la syntaxe concrète, comme les espaces ou les commentaires, n'apparaissent plus dans la représentation en syntaxe abstraite. De même, les parenthèses qui servaient à correcter associer les opérateurs à leurs opérandes ne sont plus utiles. Ainsi, les trois chaînes de caractères précédentes sont représentées par le même arbre de syntaxe abstraite arithmétique



D'autre part, la syntaxe concrète des langages de programmation offre souvent des écritures simplifiées, appelées *sucre syntaxique*, pour certaines opérations courantes. Ces formes simplifiées cependant ne viennent pas ajouter de nouvelles structures à la syntaxe abstraite, et sont simplement comprises comme une combinaison de certains éléments déjà présents dans le cœur du langage. Par exemple :

- en python l'instruction d'incrément `x += 1` est un raccourci d'écriture pour la simple instruction d'affectation `x = x + 1` et produit le même arbre de syntaxe abstraite ;
- en C, l'expression d'accès à une case de tableau `t[i]` n'est en réalité rien d'autre qu'une petite expression `*(t+i)` d'arithmétique de pointeur [1] ;
- en caml enfin, la définition d'une fonction avec **`let`** `f x = e` est en réalité une définition de variable ordinaire, à laquelle on affecte une fonction anonyme : **`let`** `f =` **`fun`** `x -> e`, et de même la définition d'une fonction à deux paramètres **`let`** `f x y = e` se décompose en **`let`** `f =` **`fun`** `x ->` **`fun`** `y -> e`.

2.2 Structure de terme et récurrence

Les arbres de syntaxe abstraite des programmes sont un cas particulier de la notion mathématique de *terme*, une structure récursive qui permet notamment de manipuler des objets à la structure arborescente, de définir des fonctions s'appliquant à de tels objets, ou encore de raisonner sur ces objets.

Signature et termes

Une ***signature*** est un ensemble Σ de symboles, dont chacun est associé à un nombre entier positif ou nul appelé ***arité***. L'ensemble $T(\Sigma)$ des ***termes*** sur la signature Σ est défini comme suit :

1. si c est un symbole de Σ d'arité 0, alors c est un terme,
2. si f est un symbole de Σ d'arité n et si t_1, ..., t_n sont n termes, alors f(t_1, ..., t_n) est un terme.

On demande en outre (mais on laisse ce point informel pour l'instant) que ces deux critères décrivent intégralement l'ensemble des termes, ou autrement dit qu'aucun objet autre que ceux construits par application répétée des deux critères n'est un terme.

Ainsi, les expressions arithmétiques du chapitre précédent peuvent être représentées par des termes sur la signature Σ_a contenant :

- un symbole n d'arité 0 pour chaque nombre entier $n \in \mathbb{N}$, et
- des symboles Add et Mul d'arité 2.

L'expression $1 + 2 \times 3$ est ainsi représentée par le terme Add(1, Mul(2, 3)). Vérifions au passage que Add(1, Mul(2, 3)) appartient bien à l'ensemble $T(\Sigma_a)$: en temps que symboles d'arité nulle, 2 et 3 sont des termes, en outre Mul est un symbole d'arité 2 et donc Mul(2, 3) est bien un terme. De plus, 1 est un terme et Add un symbole d'arité 2, donc Add(1, Mul(2, 3)) est bien un terme.

Les deux premiers critères de notre définition des termes sont appelés des propriétés de clôture. Ces propriétés ont la forme « sous certaines conditions sur ses constituants, un certain élément t appartient à $T(\Sigma)$ ». Notez que le premier critère est un *cas de base* : il est vrai sans condition pour tout symbole d'arité 0, tandis que le deuxième critère est un *cas récursif* : pour que f(t_1, ..., t_n) appartienne à $T(\Sigma)$ on demande que chacun des objets t_1 à t_n appartienne déjà à cet ensemble.

Définition de fonctions sur un ensemble de termes

L'ensemble $T(\Sigma)$ étant intégralement défini par des propriétés de clôture, ces même propriétés nous donnent un schéma pour définir des fonctions s'appliquant aux

1. Ce sucre syntaxique a des conséquences amusantes, puisqu'il permet d'écrire `2[t]` pour obtenir le même résultat que `t[2]`.

éléments de $\mathsf{T}(\Sigma)$. Pour définir une telle fonction $F : \mathsf{T}(\Sigma) \longrightarrow E$ il suffit de prévoir un cas par symbole de la signature Σ :

— pour chaque symbole c d'arité nulle, donner l'élément $e \in E$ tel que $F(\mathsf{c}) = e$;
— pour chaque symbole f d'arité n non nulle, donner une équation exprimant $F(\mathsf{f}(t_1, ..., t_n))$ en fonction des éléments t_1 à t_n, sachant que l'équation peut utiliser la valeur $F(t_i)$ de la fonction F pour les éléments t_i (on voit là à nouveau apparaître l'aspect récursif du cas des symboles d'arité non nulle).

Ainsi, on peut définir des fonctions nbCst, nbOp et eval sur nos termes représentant des expressions arithmétiques à l'aide des équations suivantes.

$$\begin{aligned}
\mathsf{nbCst}(\mathsf{n}) &= 1 \\
\mathsf{nbCst}(\mathsf{Add}(e_1, e_2)) &= \mathsf{nbCst}(e_1) + \mathsf{nbCst}(e_2) \\
\mathsf{nbCst}(\mathsf{Mul}(e_1, e_2)) &= \mathsf{nbCst}(e_1) + \mathsf{nbCst}(e_2)
\end{aligned}$$

$$\begin{aligned}
\mathsf{nbOp}(\mathsf{n}) &= 0 \\
\mathsf{nbOp}(\mathsf{Add}(e_1, e_2)) &= 1 + \mathsf{nbOp}(e_1) + \mathsf{nbOp}(e_2) \\
\mathsf{nbOp}(\mathsf{Mul}(e_1, e_2)) &= 1 + \mathsf{nbOp}(e_1) + \mathsf{nbOp}(e_2)
\end{aligned}$$

$$\begin{aligned}
\mathsf{eval}(\mathsf{n}) &= n \\
\mathsf{eval}(\mathsf{Add}(e_1, e_2)) &= \mathsf{eval}(e_1) + \mathsf{eval}(e_2) \\
\mathsf{eval}(\mathsf{Mul}(e_1, e_2)) &= \mathsf{eval}(e_1) \times \mathsf{eval}(e_2)
\end{aligned}$$

Principe de preuve par récurrence

Formellement, la définition des termes est ce qu'on appelle une *définition par clôture* : on fournit un certain nombre de propriétés de clôture et on définit l'ensemble $\mathsf{T}(\Sigma)$ comme *le plus petit* des ensembles E vérifiant toutes ces propriétés. Ainsi, si pour une certaine signature Σ, on note $\bar{\Sigma}$ l'ensemble des ensembles E vérifiant les propriétés :

1. pour tout $\mathsf{c} \in \Sigma$ d'arité 0 on a $\mathsf{c} \in E$,
2. pour tout $\mathsf{f} \in \Sigma$ d'arité n et tous $t_1 \in E$, ..., $t_n \in E$ on a $\mathsf{f}(t_1, ..., t_n) \in E$,

alors l'ensemble $\mathsf{T}(\Sigma)$ est défini comme le plus petit élément de $\bar{\Sigma}$.

Notez qu'ici, lorsque l'on écrit *le plus petit* à propos d'ensembles, le critère de comparaison est l'inclusion et pas la simple taille. Autrement dit, l'ensemble $\mathsf{T}(\Sigma)$ est caractérisé par le fait qu'il est *inclus* dans tout ensemble vérifiant les propriétés de clôture [2]. On peut déduire de cette caractéristique une technique de preuve permettant de démontrer la validité de certaines propriétés sur tous les termes de $\mathsf{T}(\Sigma)$: étant donnée une propriété P s'appliquant à des termes, si l'on peut justifier que l'ensemble $\{\ t \mid P(t) \text{ est vraie }\}$ appartient à $\bar{\Sigma}$ alors on en déduit que $\mathsf{T}(\Sigma)$ est inclus dans $\{\ t \mid P(t) \text{ est vraie }\}$, autrement dit que pour tout $t \in \mathsf{T}(\Sigma)$ la propriété $P(t)$ est vraie.

Finalement, étant donnée une propriété P parlant des termes de $\mathsf{T}(\Sigma)$, on démontre que cette propriété est vraie pour tous les termes en démontrant les choses suivantes :

1. pour tout $\mathsf{c} \in \Sigma$ d'arité 0, la propriété $P(\mathsf{c})$ est vraie, et
2. pour tous $\mathsf{f} \in \Sigma$ d'arité n et tous $t_1 \in \mathsf{T}(\Sigma)$, ..., $t_n \in \mathsf{T}(\Sigma)$ tels que $P(t_1)$, ..., $P(t_n)$ sont vraies, la propriété $P(\mathsf{f}(t_1, ..., t_n))$ est encore vraie.

Cette technique de preuve est une ***preuve par récurrence***, qu'on appelle plus précisément *preuve par récurrence sur la structure des termes*. Le premier point décrit ses ***cas de base*** (un par symbole d'arité nulle) et le deuxième point ses ***cas récursifs*** (un par symbole d'arité non nulle). Dans le deuxième point, les hypothèses $P(t_1)$ à $P(t_n)$ que l'on peut utiliser pour justifier $P(\mathsf{f}(t_1, ..., t_n))$ sont les ***hypothèses de récurrence***.

Montrons par exemple que dans toute expression arithmétique sur notre signature Σ_a, le nombre de constantes est de un supérieur au nombre d'opérateurs. Pour cela, notons $P(e)$ la propriété $\mathsf{nbCst}(e) = \mathsf{nbOp}(e) + 1$, et vérifions les cas de base et les cas héréditaires correspondant aux différents symboles de la signature :

— Cas de la constante (cas de base) : pour tout terme constant n on a $\mathsf{nbCst}(\mathsf{n}) = 1$ et $\mathsf{nbOp}(\mathsf{n}) = 0$. Ainsi la propriété P est bien vérifiée pour le terme n.
— Cas de l'addition (cas récursif) : soient deux expressions e_1 et e_2 pour lesquelles

2. L'existence et l'unicité d'un tel ensemble minimum pour l'inclusion pour un certain ensemble de propriétés de clôture ne sont pas triviales en général. Elles peuvent même être mises en défaut si les propriétés de clôture n'ont pas la bonne forme. Mais dans le cas de la définition des termes tout fonctionne.

la propriété P est vraie. On a alors

$$\begin{array}{lll} & \mathsf{nbCst}(\mathsf{Add}(e_1, e_2)) & \\ = & \mathsf{nbCst}(e_1) + \mathsf{nbCst}(e_2) & \text{par définition de nbCst} \\ = & (\mathsf{nbOp}(e_1) + 1) + (\mathsf{nbOp}(e_2) + 1) & \text{par hypothèses de récurrence} \\ = & (1 + \mathsf{nbOp}(e_1) + \mathsf{nbOp}(e_2)) + 1 & \text{(réarrangement)} \\ = & \mathsf{nbOp}(\mathsf{Add}(e_1, e_2)) + 1 & \text{par définition de nbOp} \end{array}$$

Autrement dit, la propriété P est encore vraie pour le terme $\mathsf{Add}(e_1, e_2)$.

— Cas de la multiplication (cas récursif) : similaire au cas de l'addition.

On a donc démontré par récurrence sur la structure des termes que pour tout terme $e \in \mathsf{T}(\Sigma_a)$ on a bien $\mathsf{nbCst}(e) = \mathsf{nbOp}(e) + 1$.

Signatures et termes avec sortes

On peut enrichir la notion d'arité d'un symbole pour ne pas seulement donner le nombre des éléments qu'il assemble, mais également la nature de chacun, appelée ***sorte***. On peut ainsi définir une signature pour des termes représentant des listes chaînées d'entiers avec deux symboles :

— un symbole Nil d'arité 0 pour la liste vide,

— un symbole Cel d'arité 2 s'appliquant à un nombre entier et à une liste pour une cellule.

En notant liste la sorte des listes chaînées, on pourra résumer l'arité de Nil avec la notation liste (le symbole représente lui-même une liste) et l'arité de Cel avec la notation $\mathbb{N} \times \text{liste} \longrightarrow \text{liste}$ (le symbole combine un entier et une liste pour former une nouvelle liste).

La liste chaînée contenant dans l'ordre les nombres 1, 2 et 4 peut alors être représentée par le terme Cel(1, Cel(2, Cel(4, Nil))).

Le principe de raisonnement par récurrence sur les termes avec sortes est identique au principe précédent, à ceci près que la récurrence ne porte que sur les éléments de sorte adaptée. Par exemple, pour démontrer par récurrence qu'une propriété P est vraie pour toutes les listes d'entiers, on montre :

— que P(Nil) est vraie,

— que pour tout $n \in \mathbb{N}$ et tout $l \in$ liste, si $P(l)$ est vraie alors $P(\mathsf{Cel}(n, l))$ est encore vraie.

2.3 Variables et environnements

Une variable est un nom désignant une valeur stockée en mémoire [3].

```
int x = 3;
int y = 1 + 2 * x;
return 2 * x * y;
```

```
let x = 3 in
let y = 1 + 2 * x in
2 * x * y
```

Évaluation des expressions avec variables

Pour représenter les variables, il suffit d'ajouter à la signature des expressions un symbole d'arité 0 pour chaque nom de variable. La fonction eval d'évaluation des expressions change cependant de nature : elle a maintenant besoin d'une information sur la mémoire, ou plus précisément sur les valeurs associées aux différentes variables. On peut abstraire cette information sous la forme d'une fonction ρ appelée ***environnement***, qui à des noms de variables associe leur valeur.

La fonction eval devient alors une fonction prenant un environnement en deuxième paramètre, à laquelle elle fait appel pour obtenir la valeur des variables.

$$\begin{array}{rcl} \mathsf{eval}(\mathsf{n}, \rho) & = & n \\ \mathsf{eval}(\mathsf{x}, \rho) & = & \rho(\mathsf{x}) \\ \mathsf{eval}(\mathsf{Add}(e_1, e_2), \rho) & = & \mathsf{eval}(e_1, \rho) + \mathsf{eval}(e_2, \rho) \\ \mathsf{eval}(\mathsf{Mul}(e_1, e_2), \rho) & = & \mathsf{eval}(e_1, \rho) \times \mathsf{eval}(e_2, \rho) \end{array}$$

L'environnement ρ évolue à mesure que l'évaluation d'un programme progresse, mais la nature de cette évolution peut être très différente d'un paradigme de programmation à l'autre.

3. Notez la différence avec la notion de variable en mathématiques, qui désigne au contraire une chose indéterminée.

Version fonctionnelle : définition de variable locale

Dans un langage comme caml, les variables sont *immuables* : elles reçoivent lors de leur définition une valeur, qui n'est jamais modifiée.

L'expression `let y = 1+2*x in 2*x*y` définit une variable `y` en lui donnant la valeur calculée par l'expression `1+2*x`. Cette valeur peut ensuite être utilisée lors de l'évaluation de l'expression `2*x*y`. Autrement dit, l'expression `1+2*x` est évaluée dans un certain environnement ρ, donnant notamment la valeur de `x`, puis l'expression `2*x*y` est évaluée dans un nouvel environnement ρ', qui étend ρ en `y` ajoutant l'association de sa valeur à `y`.

Notez que cette variable est locale à l'expression `2*x*y` située à droite du **`in`**, elle n'existe pas dans les autres parties du programme. Autrement dit, l'environnement étendu ρ' n'est utilisé que pour l'évaluation de cette sous-expression.

Notons $\mathsf{Let}(\mathsf{x}, e_1, e_2)$ le terme représentant l'expression caml `let x = `e_1` in `e_2. Pour un environnement ρ, une variable x et une valeur v, notons $\rho[x \mapsto v]$ l'environnement qui à x associe v et à toute variables $y \neq x$ associe $\rho(y)$. On obtient alors la nouvelle équation suivante pour la fonction eval.

$$\mathsf{eval}(\mathsf{Let}(\mathsf{x}, e_1, e_2), \rho) \quad = \quad \mathsf{eval}(e_2, \rho[x \mapsto \mathsf{eval}(e_1, \rho)])$$

Bien que les variables soient immuables, rien n'interdit en caml de redéfinir une nouvelle variable locale, du même nom qu'une variable existante.

```
let x = 1 in
let x = 2 in
x
```

Cette nouvelle version *masque* la précédente, et le `x` final de l'expression précédente sera donc associé à la valeur `2` donnée par la deuxième définition. Ce masquage ne vaut cependant que dans la partie de l'expression où la nouvelle variable existe. Ainsi dans le programme

```
let x = 1 in
let y = (let x = 2 in x) in
x + 2*y
```

le `x` de la dernière ligne vaut `1`, et `y` vaut `2`, soit la valeur de la deuxième définition de `x`. Notez que ce programme est à tous points de vue équivalent à la version suivante dans laquelle le deuxième `x` a été nommé `z` pour éviter les confusions.

```
let x = 1 in
let y = (let z = 2 in z) in
x + 2*y
```

De même, l'expression `let x = 1 in let x = 2 in x` de l'exemple précédent est équivalente à l'expression `let z = 1 in let x = 2 in x`.

Transparence référentielle

Dans un langage d'expressions arithmétiques où les variables locales sont immuables, l'expression $\mathsf{Let}(x, e_1, e_2)$ est équivalente à l'expression e_2 dans laquelle chaque occurrence de x aurait été remplacée par l'expression e_1. Par ***équivalente*** on signifie que les deux expressions produisent le même résultat, quelque soit le contexte ρ dans lequel on les évalue toutes deux.

Définition de la ***substitution*** $e[x := s]$ de chaque occurrence de x dans e par s.

$$\begin{aligned}
\mathsf{n}[x := s] &= \mathsf{n} \\
\mathsf{y}[x := s] &= \begin{cases} s & \text{si } x = y \\ \mathsf{y} & \text{sinon} \end{cases} \\
\mathsf{Add}(e_1, e_2)[x := s] &= \mathsf{Add}(e_1[x := s], e_2[x := s]) \\
\mathsf{Mul}(e_1, e_2)[x := s] &= \mathsf{Mul}(e_1[x := s], e_2[x := s]) \\
\mathsf{Let}(y, e_1, e_2)[x := s] &= \begin{cases} \mathsf{Let}(y, e_1[x := s], e_2) & \text{si } x = y \\ \mathsf{Let}(y, e_1[x := s], e_2[x := s]) & \text{si } x \neq y \text{ et } y \notin \mathsf{fv}(s) \end{cases}
\end{aligned}$$

où l'ensemble $\mathrm{fv}(e)$ des ***variables libres*** d'une expression e est défini par les équations

$$\begin{array}{rcl}
\mathrm{fv}(\mathsf{n}) & = & \emptyset \\
\mathrm{fv}(\mathsf{x}) & = & \{x\} \\
\mathrm{fv}(\mathsf{Add}(e_1, e_2)) & = & \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \\
\mathrm{fv}(\mathsf{Mul}(e_1, e_2)) & = & \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \\
\mathrm{fv}(\mathsf{Let}(x, e_1, e_2)) & = & \mathrm{fv}(e_1) \cup (\mathrm{fv}(e_2) \setminus \{x\})
\end{array}$$

Pour justifier l'équivalence entre la construction Let et la substitution, on démontre la propriété suivante.

$$\forall x, e_1, e_2, \rho,\ x \notin \mathrm{fv}(e_1) \implies \mathrm{eval}(\mathsf{Let}(x, e_1, e_2), \rho) = \mathrm{eval}(e_2[x := e_1], \rho)$$

Preuve par récurrence sur la struture de e_2.

— Cas n.
On a d'une part

$$\begin{array}{l}
\mathrm{eval}(\mathsf{Let}(x, e_1, \mathsf{n}), \rho) \\
= \quad \mathrm{eval}(\mathsf{n}, \rho[x \mapsto \mathrm{eval}(e_1, \rho)]) \\
= \quad \mathsf{n}
\end{array}$$

et d'autre part

$$\begin{array}{l}
\mathrm{eval}(\mathsf{n}[x := e_1], \rho) \\
= \quad \mathrm{eval}(\mathsf{n}, \rho) \\
= \quad \mathsf{n}
\end{array}$$

On a donc bien $\mathrm{eval}(\mathsf{Let}(x, e_1, \mathsf{n}), \rho) = \mathrm{eval}(\mathsf{n}[x := e_1], \rho)$.

— Cas y.
  — Si $x = y$ on a d'une part

$$\begin{array}{l}
\mathrm{eval}(\mathsf{Let}(x, e_1, x), \rho) \\
= \quad \mathrm{eval}(x, \rho[x \mapsto \mathrm{eval}(e_1, \rho)]) \\
= \quad \mathrm{eval}(e_1, \rho)
\end{array}$$

et d'autre part

$$\begin{array}{l}
\mathrm{eval}(x[x := e_1], \rho) \\
= \quad \mathrm{eval}(e_1, \rho)
\end{array}$$

On a donc bien $\mathrm{eval}(\mathsf{Let}(x, e_1, x), \rho) = \mathrm{eval}(x[x := e_1], \rho)$.

  — Sinon on a d'une part

$$\begin{array}{l}
\mathrm{eval}(\mathsf{Let}(x, e_1, y), \rho) \\
= \quad \mathrm{eval}(y, \rho[x \mapsto \mathrm{eval}(e_1, \rho)]) \\
= \quad \rho(y)
\end{array}$$

et d'autre part

$$\begin{array}{l}
\mathrm{eval}(y[x := e_1], \rho) \\
= \quad \mathrm{eval}(\mathsf{y}, \rho) \\
= \quad \rho(y)
\end{array}$$

On a donc bien $\mathrm{eval}(\mathsf{Let}(x, e_1, y), \rho) = \mathrm{eval}(y[x := e_1], \rho)$.

— Cas Add.
Soient e_{21} et e_{22} deux expressions telles que

$$\begin{array}{lcl}
\mathrm{eval}(\mathsf{Let}(x, e_1, e_{21}), \rho) & = & \mathrm{eval}(e_{21}[x := e_1], \rho) \\
\mathrm{eval}(\mathsf{Let}(x, e_1, e_{22}), \rho) & = & \mathrm{eval}(e_{22}[x := e_1], \rho)
\end{array}$$

On a d'une part

$$\begin{array}{l}
\mathrm{eval}(\mathsf{Let}(x, e_1, \mathsf{Add}(e_{21}, e_{22})), \rho) \\
= \quad \mathrm{eval}(\mathsf{Add}(e_{21}, e_{22}), \rho[x \mapsto \mathrm{eval}(e_1, \rho)]) \\
= \quad \mathrm{eval}(e_{21}, \rho[x \mapsto \mathrm{eval}(e_1, \rho)]) + \mathrm{eval}(e_{22}, \rho[x \mapsto \mathrm{eval}(e_1, \rho)])
\end{array}$$

et d'autre part

$$\begin{aligned}&\mathsf{eval}(\mathsf{Add}(e_{21}, e_{22})[x := e_1], \rho)\\ =\ &\mathsf{eval}(\mathsf{Add}(e_{21}[x := e_1], e_{22}[x := e_1]), \rho)\\ =\ &\mathsf{eval}(e_{21}[x := e_1], \rho) + \mathsf{eval}(e_{22}[x := e_1], \rho)\end{aligned}$$

avec (par hypothèse de récurrence)

$$\begin{aligned}&\mathsf{eval}(e_{21}[x := e_1], \rho)\\ =\ &\mathsf{eval}(\mathsf{Let}(x, e_1, e_{21}), \rho)\\ =\ &\mathsf{eval}(e_{21}, \rho[x \mapsto \mathsf{eval}(e_1, \rho)])\end{aligned}$$

et de même $\mathsf{eval}(e_{22}[x := e_1], \rho) = \mathsf{eval}(e_{22}, \rho[x \mapsto \mathsf{eval}(e_1, \rho)])$.
On a donc bien $\mathsf{eval}(\mathsf{Let}(x, e_1, \mathsf{Add}(e_{21}, e_{22})), \rho) = \mathsf{eval}(\mathsf{Add}(e_{21}, e_{22})[x := e_1], \rho)$.

— Deux derniers cas similaires.

Exercice 2.1 (Décomposition d'opérations)**.** Déduire de la propriété précédente qu'une expression $\mathsf{Add}(e_1, e_2)$ peut, en prenant deux noms de variables non encore utilisés x_1 et x_2, être décomposée en l'une ou l'autre des séquences $\mathsf{Let}(x_1, e_1, \mathsf{Let}(x_2, e_2, \mathsf{Add}(x_1, x_2)))$ et $\mathsf{Let}(x_2, e_2, \mathsf{Let}(x_1, e_1, \mathsf{Add}(x_1, x_2)))$ qui lui sont toutes deux équivalentes.

Version impérative : instruction d'affectation

Dans un langage comme C, python ou java, les variables sont au contraire *mutables* : les instructions successives d'un programme peuvent modifier à volonté la valeur des variables de ce programme.

L'instruction `y = 1+2*x` affecte à la variable `y` une nouvelle valeur, calculée par l'expression `1+2*x`. Une fois cette instruction exécutée dans un certain environnement ρ, toute la suite du programme sera exécutée dans l'environnement modifié ρ', qui est identique à ρ si ce n'est qu'il associe à y la nouvelle valeur.

Notons $\mathsf{Set}(\mathsf{x}, e)$ le terme représentant l'instruction d'affectation `x = e;`. Nous définissons là une nouvelle signature pour un nouvel ensemble de termes représentant les instructions, en parallèle des termes déjà utilisés pour les expressions. L'exécution des instructions peut être décrite par une nouvelle fonction exec, qui s'applique à une instruction ou une séquence d'instructions et à un environnement. Pour modéliser le fait que l'effet d'une instruction est de modifier l'environnement avant l'exécution des instructions suivantes, on va définir comme résultat de exec l'environnement ρ' modifié. Ainsi

$$\begin{aligned}\mathsf{exec}(\mathsf{Set}(\mathsf{x}, e), \rho) &= \rho[x \mapsto \mathsf{eval}(e, \rho)]\\ \mathsf{exec}(i_1\,;\, i_2, \rho) &= \mathsf{exec}(i_2, \mathsf{exec}(i_1, \rho))\end{aligned}$$

Dans un tel langage, toute nouvelle affectation écrase la valeur précédente de la variable. Ainsi dans le programme

```
x = 1;
x = 2;
```

les deux occurrences du nom `x` désignent bien la même variable. La deuxième affectation remplace définitivement l'association de `x` à 1 par une association de `x` à 2.

2.4 Interprétation d'un langage impératif

On considère le mini-langage IMP, qui illustre les bases de la programmation impérative. Voici un exemple de programme IMP.

```
n = 6;
r = 1;
while (0 < n) {
  r = r * n;
  n = n + (-1);
}
```

On distingue dans ce langage une notion d'expression, formée de constantes entières, de variables, d'additions, de multiplications et de comparaisons, et une notion d'instruction comprenant l'affectation d'une valeur à une variable, la boucle `while` et le branchement conditionnel `if/else`.

Syntaxe abstraite

La syntaxe abstraite de ce langage est donnée par deux ensembles de termes : un décrivant les instructions et un décrivant les expressions. La signature Σ_e des expressions contient :

— un symbole `n` d'arité 0 pour chaque entier n,
— un symbole d'arité 0 pour chaque nom de variable,
— des symboles `Add`, `Mul` et `Lt` d'arité 2 désignant respectivement l'addition `+`, la multiplication `*` et la comparaison `<`.

On peut définir en caml le type correspondant

```
type expr =
  | Cst of int
  | Var of string
  | Add of expr * expr
  | Mul of expr * expr
  | Lt of expr * expr
```

La signature (sortée) des instructions contient :

— un symbole `Set` s'appliquant à un nom de variable et à une expression pour l'affectation,
— un symbole `If` s'appliquant à une expression et à deux listes d'instructions pour le branchement conditionnel,
— un symbole `While` s'appliquant à une expression et à une liste d'instructions pour la boucle.

On peut définir en caml le type correspondant

```
type instr =
  | Set of string * expr
  | If of expr * instr list * instr list
  | While of expr * instr list
```

Finalement, l'exemple de programme IMP donné plus haut est représenté par le terme caml

```
[ Set("n", Cst 6);
  Set("r", Cst 1);
  While(Lt(Cst 0, Var "n"),
        [ Set("r", Mul(Var "r", Var "n"));
          Set("n", Add(Var "n", Cst(-1)))
        ])
]
```

Valeurs et environnements

On se donne pour représenter les valeurs produites par les expressions IMP un type `value` désignant simplement des nombres entiers. Un environnement doit associer des noms de variables (c'est-à-dire des chaînes de caractères) à des valeurs (de type `value`). On a donc besoin d'une structure de ***tableau associatif***, que l'on peut réaliser efficacement soit à l'aide d'arbres de recherche équilibrés (module `Map` de caml) lorsque l'on programme en style purement fonctionnel, soit à l'aide d'une table de hachage (module `Hashtbl` de caml) lorsque l'on s'autorise des structures de données mutables.

On va utiliser ici la solution à base d'arbres de recherche équilibrés, que l'on peut introduire par les déclarations suivantes.

```
module Env = Map.Make(String)
type env = value Env.t
```

Après cette déclaration, le type `env` désigne donc des tables associatives avec des clés de type `string` et des valeurs de type `value`. Le module `Env` fournit une constante `Env.empty` pour une table vide et de nombreuses fonctions, dont `Env.find` pour la récupération de la valeur associée à une clé dans une table, ou `Env.add` pour l'ajout ou le remplacement d'une association à une table.

Notez qu'en pratique, lorsque les approches à base d'arbres de recherche et à base de table de hachage sont toutes deux possibles, la deuxième est plus efficace. Les accès sont en effet en temps constant plutôt que logarithmique. Ici nous utilisons la version

purement fonctionnelle à base d'arbres de recherche essentiellement pour montrer que le fait que nous écrivions un interprète pour un langage impératif n'empêche pas d'écrire l'interprète lui-même dans un style purement fonctionnel.

Évaluation des expressions

On peut reprendre les règles déjà données pour l'évaluation des expressions. La seule nouveauté est une décision à prendre quant à la valeur produite par une comparaison. On peut par exemple introduire des valeurs particulières vrai et faux à côté des nombres déjà utilisés, ou encore réutiliser des entiers particuliers, par exemple 1 pour vrai et 0 pour faux.

$$\text{eval}(\text{Lt}(e_1, e_2), \rho) \quad = \quad \begin{cases} 1 & \text{si eval}(e_1, \rho) < \text{eval}(e_2, \rho) \\ 0 & \text{sinon} \end{cases}$$

```
let rec eval_expr (e: expr) (env: env): value = match e with
  | Cst n -> n
  | Var x -> Env.find x env
  | Add(e1, e2) -> (eval_expr e1 env) + (eval_expr e2 env)
  | Mul(e1, e2) -> (eval_expr e1 env) * (eval_expr e2 env)
  | Lt(e1, e2) ->
     if (eval_expr e1 env) < (eval_expr e2 env) then 1 else 0
```

Sémantique des instructions

On reprend de même les équations déjà données pour la fonction exec, en l'étendant pour tenir compte des constructions if et while. Dans chaque cas on obtient une équation avec deux voies possibles, en fonction de la valeur obtenue en évaluant la garde.

$$\text{exec}(\text{If}(e, b_1, b_2), \rho) \quad = \quad \begin{cases} \text{exec}(b_1, \rho) & \text{si eval}(e, \rho) \neq 0 \\ \text{exec}(b_2, \rho) & \text{sinon} \end{cases}$$

$$\text{exec}(\text{While}(e, b), \rho) \quad = \quad \begin{cases} \rho & \text{si eval}(e, \rho) = 0 \\ \text{exec}(\text{While}(e, b), \text{exec}(b, \rho)) & \text{sinon} \end{cases}$$

L'équation pour la séquence de deux instructions peut également être généralisée pour décrire en une seule fois un bloc d'une longueur n arbitraire.

$$\text{exec}(i_1 ; i_2 ; \ldots i_n, \rho) \quad = \quad \text{exec}(i_n, \text{exec}(i_{n-1}, \ldots \text{exec}(i_1, \rho) \ldots))$$

```
let rec eval_instr (i: instr) (env: env): env = match i with
  | Set(x, e) ->
     let v = eval_expr e env in
     Env.add x v env
  | If(e, b1, b2) ->
     if eval_expr e env = 0 then
       eval_block b1 env
     else
       eval_block b2 env
  | While(e, b) ->
     if eval_expr e env = 0 then
       env
     else
       let env' = eval_block b env in
       eval_instr i env'

and eval_block (b: instr list) (env: env): env = match b with
  | [] -> env
  | i :: b' -> let env' = eval_instr i env in
               eval_block b' env'
```

Décomposition de l'exécution de notre programme exemple, en partant de l'environement vide.

exec(Set(n, 6); Set(r, 1); While(...), [])
= exec(While(...), exec(Set(r, 1), exec(Set(n, 6), [])))
= exec(While(...), exec(Set(r, 1), $[n \mapsto 6]$))
  car: eval(6, [])
  = 6
= exec(While(...), $[n \mapsto 6, r \mapsto 1]$)
  car: eval(1, $[n \mapsto 6]$)
  = 1
= exec(While(...), exec(Set(r, Mul(r, n)); Set(n, Add(n, -1)), $[n \mapsto 6, r \mapsto 1]$))
  car: eval(Lt(0, n), $[n \mapsto 6, r \mapsto 1]$)
  = 1
  car: eval(0, $[n \mapsto 6, r \mapsto 1]$) = 0 < 6 = eval(n, $[n \mapsto 6, r \mapsto 1]$)
= exec(While(...), $[n \mapsto 5, r \mapsto 6]$))
  exec(Set(r, Mul(r, n)); Set(n, Add(n, -1)), $[n \mapsto 6, r \mapsto 1]$)
  = exec(Set(n, Add(n, -1)), exec(Set(r, Mul(r, n)), $[n \mapsto 6, r \mapsto 1]$))
  = exec(Set(n, Add(n, -1)), $[n \mapsto 6, r \mapsto 6]$)
  car: eval(Mul(r, n), $[n \mapsto 6, r \mapsto 1]$)
  = eval(r, $[n \mapsto 6, r \mapsto 1]$) × eval(n, $[n \mapsto 6, r \mapsto 1]$)
  = 1 × 6
  = 6
  = $[n \mapsto 5, r \mapsto 6]$
  car: eval(Add(n, -1), $[n \mapsto 6, r \mapsto 6]$)
  = eval(n, $[n \mapsto 6, r \mapsto 6]$) + eval(-1, $[n \mapsto 6, r \mapsto 6]$)
  = 6 + (−1)
  = 5
= exec(While(...), $[n \mapsto 4, r \mapsto 30]$) car ...
= exec(While(...), $[n \mapsto 3, r \mapsto 120]$) car ...
= exec(While(...), $[n \mapsto 2, r \mapsto 360]$) car ...
= exec(While(...), $[n \mapsto 1, r \mapsto 720]$) car ...
= exec(While(...), $[n \mapsto 0, r \mapsto 720]$) car ...
= $[n \mapsto 0, r \mapsto 720]$
  car: eval(Lt(0, n), $[n \mapsto 0, r \mapsto 720]$)
  = 0
  car: eval(0, $[n \mapsto 0, r \mapsto 720]$) = 0 $\not<$ 0 = eval(n, $[n \mapsto 0, r \mapsto 720]$)

Si l'on tentait de faire un tel raisonnement sur le programme Set(x, 0); While(1, Set(x, Add(x, 1))) en revanche on obtiendrait

- exec(Set(x, 0), []) = $[x \mapsto 0]$
  car eval(0, []) = 0
- exec(While(1, Set(x, Add(x, 1))), $[x \mapsto 0]$) =???
  car eval(1, $[x \mapsto 0]$) = 1 et
  - exec(Set(x, Add(x, 1)), $[x \mapsto 0]$) = $[x \mapsto 1]$
    car eval(Add(x, 1), $[x \mapsto 0]$) = 1
  - exec(While(1, Set(x, Add(x, 1))), $[x \mapsto 1]$) =???
    car eval(1, $[x \mapsto 1]$) = 1 et
    - exec(Set(x, Add(x, 1)), $[x \mapsto 1]$) = $[x \mapsto 2]$
      car eval(Add(x, 1), $[x \mapsto 1]$) = 2
    - exec(While(1, Set(x, Add(x, 1))), $[x \mapsto 2]$) =???
      car eval(1, $[x \mapsto 2]$) = 1 et
      - ...
      - ...

et ainsi de suite sans jamais pouvoir atteindre une égalité qui n'ait plus besoin de justification. La boucle infinie de ce programme déclenche du côté sémantique une fuite infinie dans les justifications. De ce fait, il est impossible d'obtenir un environnement ρ pour lequel on aurait Set(x, 0); While(1, Set(x, Add(x, 1))) = ρ. Cet exemple met en évidence que la fonction exec n'est pas une fonction totale : il existe des paires programme/environnement pour lesquelles elle n'est pas définie. C'est l'une

des raisons pour lesquelles la sémantique est souvent définie comme une relation plutôt que comme une fonction (nous y reviendrons).

2.5 Stratégies d'évaluation

Nous avons donné ci-dessus une manière simple d'évaluer une expression de la forme e_1 `+` e_2 : d'abord évaluer les deux sous-expressions e_1 et e_2, puis additionner les résultats. Cette méthode ne s'étend cependant pas directement à tous les opérateurs binaires.

Opérateurs paresseux

Comment calcule-t-on la valeur de l'expression suivante ?

```
n > 0 && x  mod n == 0
```

Quelques scénarios.

- Si `n` vaut 3 et `x` vaut 6, les deux opérandes du `&&` s'évaluent à vrai, indiquant que 6 est bien un multiple de 3.
- Si `n` vaut 3 et `x` vaut 7, l'opérande de gauche s'évalue à vrai et celui de droite à faux, indiquant que 7 n'est pas un multiple de 3.
- Si en revanche `n` vaut 0 et `x` vaut 1, l'évaluation de l'opérande de droite conduirait à une erreur « division par zéro » et empêcherait le calcul d'aboutir.

En général, le troisième scénario ne se réalise pas, les langages réalisant la stratégie suivante, dite *paresseuse* :

1. évaluer l'opérande de gauche du `&&`,
2. puis en fonction du résultat,
   - si vrai, alors évaluer l'opérande de droite,
   - si faux, alors indiquer le résultat global faux sans évaluer l'opérande de droite.

Ainsi, les différents ordres possibles pour organiser l'interprétation d'un opérateur et l'évaluation de ses opérandes ne sont pas toujours équivalents. On appelle *strict* un opérateur dont tous les opérandes sont évalués systématiquement (comme `+`, `*` ou `<`), et *paresseux* un opérateur dont les opérandes ne sont évalués qu'en fonction des besoins (comme `&&` ou `||`).

Stratégies d'évaluation des fonctions

Les notions d'évaluation stricte ou paresseuse s'étendent des opérateurs aux appels de fonction. Considérons la fonction

```
int f(int x, int y, int z) {
  return x*x + y*y;
}
```

et un appel `f(1+2, 2+2, 1/0)` à cette fonction. On appelle `x`, `y` et `z` les ***paramètres formels*** de la fonction `f`, et on appelle `1+2`, `2+2` et `1/0` les ***paramètres effectifs*** de l'appel. On distingue trois stratégies.

- On peut évaluer tous les paramètres effectifs `1+2`, `2+2` et `1/0` avant de passer la main à la fonction `f` elle-même. En l'occurrence le calcul sera alors interrompu par une erreur « division par zéro » sur l'évaluation du troisième paramètre effectif. Cette stratégie stricte est nommée *appel par valeur*. Elle est en vigueur dans de très nombreux langages, dont caml, C, java, python.
- On peut remplacer chaque occurrence des paramètres formels `x`, `y` et `z` dans le corps de la fonction `f` par les paramètres effectifs non évalués. Les paramètres effectifs ne sont alors évalués que lorsqu'ils sont nécessaires, et le troisième ne sera pas évalué et donc ne déclenchera pas d'erreur. En revanche, les deux premiers paramètres seront évalués deux fois chacun. Cette stratégie paresseuse simple est nommée *appel par nom*. Cette stratégie est rare dans des langages généralistes, mais correspond par exemple au traitement des macros.
- On peut améliorer l'appel par nom à l'aide d'un mécanisme permettant de ne pas recalculer la valeur d'un paramètre effectif qui aurait déjà été évalué. Dans ce cas les deux premiers paramètres sont évalués une fois chacun, et le troisième n'est pas évalué. Cette stratégie paresseuse plus élaborée est nommée *appel par nécessité*. Elle est utilisée dans certains langages purement fonctionnels, comme Haskell.

Ordre d'évaluation et effets

Revenons à une simple addition :

$$e_1 + e_2$$

L'opérateur d'addition étant strict, on sait qu'il faut évaluer les deux sous-expressions e_1 et e_2 pour obtenir la valeur de l'expression complète. Cependant, y a-t-il une différence entre les scénarios suivants ?

1. Évaluer d'abord e_1, puis e_2.
2. Évaluer d'abord e_2, puis e_1.
3. Évaluer en alternance des fragments de e_1 et de e_2.

Quizz : savez-vous comment procèdent de ce point de vue caml, C, java, python ?

La règle d'évaluation

$$\text{eval}(\text{Add}(e_1, e_2), \rho) \quad = \quad \text{eval}(e_1, \rho) + \text{eval}(e_2, \rho)$$

donne une réponse : les sous-expressions e_1 et e_2 sont évaluées indépendamment l'une de l'autre, chacune dans le même environnement ρ, et l'ordre ne change rien aux valeurs obtenues.

Cette règle et ce raisonnement ne valent en revanche que lorsque l'évaluation des expressions e_1 et e_2 ne produit pas d'effets collatéraux (ou *effets de bord*) tels qu'une modification de la mémoire ou un affichage. Dans le cas contraire, l'ordre d'évaluation de e_1 et e_2 se traduit par un ordre de réalisation des effets de bords.

```
int x = 1;

int incr() {
  x += 1;
  return x;
}
int double() {
  x *= 2;
  return x;
}

incr() + double();
```

Si l'on évalue d'abord l'opérande de gauche, l'appel `incr()` modifie la mémoire en affectant 2 à `x`, et renvoie 2. Puis l'appel `double()` affecte 4 à `x` et renvoie 4. Le résultat est 6. Si au contraire on évalue d'abord l'opérande de droite, l'appel `double()` affecte 2 à `x` et renvoie 2, puis l'appel `incr()` affecte 3 à `x` et renvoie 3. Le résultat est cette fois 5.

Finalement, dans les langages purements fonctionnels le caractère immuable des variables (et plus généralement l'absence d'effets de bord) fait que les différentes partie d'une expression peuvent être évaluées dans un ordre arbitraire, sans que cela ne modifie d'aucune façon le résultat obtenu : les deux opérandes d'un opérateur binaire peuvent être évalués dans un ordre arbitraire, de même que tous les paramètres effectifs d'un appel de fonction strict. Cette remarque justifie également que les stratégies non strictes (appel par nom ou par nécessité) produisent encore les mêmes résultats [4].

À l'inverse la possibilité pour une expression de produire, via un appel de fonction, un effet de bord sur la mémoire invalide la définition de eval vue jusque là : si l'évaluation d'une expression est susceptible de modifier l'environnement, alors la fonction eval doit, comme la fonction exec des instructions, renvoyer l'environnement modifié. Nous aurions ainsi une fonction eval renvoyant une paire valeur/environnement, et la règle pour l'addition pourrait devenir :

$$\text{eval}(\text{Add}(e_1, e_2), \rho) \quad = \quad (v_1 + v_2,\ \rho_2) \qquad \begin{array}{ll} \text{si} & \text{eval}(e_1, \rho) = (v_1,\ \rho_1) \\ \text{et} & \text{eval}(e_2, \rho_1) = (v_2,\ \rho_2) \end{array}$$

Notez que, par la succession des environnement ρ, ρ_1 et ρ_2, notre règle fixe une évaluation de gauche à droite des opérandes d'une addition.

4. On fait abstraction dans cette remarque de la situation où le calcul est interrompu car l'évaluation de l'un des fragments produit une erreur. Notez qu'une telle erreur est parfois elle-même considérée comme un *effet*.

2.6 Variables, adresses et partage

En présence de variables mutables, la définition précise de l'effet d'une instruction d'affectation présente quelques subtilités.

Questions à propos des variables

À la fin de l'exécution de chacun des programmes suivants, la variable `y` doit-elle valoir 1 ou 2 ?

```
int x = 1;
int y = x;
x = 2;
```

```
void f(int x) {
  x = x + 1;
}
int y = 1;
f(y);
```

```
int x = 1;
int f() {
  return x;
}
x = 2;
int y = f();
```

Quizz : qu'en est-il dans les langages que vous connaissez ? Certains langages permettent-ils les deux comportements ?

Les différents scénarios possibles viennent d'une double signification des variables. Une variable est *un nom* qui, selon l'endroit où elle est utilisée, peut désigner des choses différentes :

- utilisé dans une expression arithmétique, un nom de variable désigne en général *la valeur* de cette variable ;
- utilisé à gauche du symbole d'affectation, un nom de variable désigne en général *l'emplacement mémoire* de cette variable (autrement dit *un pointeur*), dont on veut modifier le contenu.

Cette polysémie est partagée avec les autres formes d'expression que l'on peut trouver à gauche d'un symbole d'affectation, que l'on nomme collectivement les *valeurs gauches* : par exemple l'expression `t[i]` désignant une case d'un tableau ou l'expression `s.x` désignant un champ ou un attribut d'une structure ou d'un objet.

Variables partagées (*aliasing*)

Considérons notre premier exemple.

```
x = 1;
y = x;
x = 2;
```

Qu'attendons-nous ici comme valeur finale pour `y` ? La réponse à cette question dépend du sens donné à l'affectation `y = x;`

- Il peut s'agir de l'affectation à `y` de la valeur courante de la variable `x` (en l'occurrence 1), après quoi les deux variables `x` et `y` restent indépendantes. Dans ce cas, `y` vaut toujours 1 à la fin.
- Il peut s'agir de définir `y` comme étant *la même variable* que `x`. On obtient alors deux noms désignant la même variable (on parle d'*aliasing*), et toute modification de cette variable commune vaut donc pour les deux noms. Dans cette situation, la valeur finale de `y` est 2.

Quizz : en C, en python, en java, laquelle de ces deux significations est retenue ? Dans chacun de ces langages, est-il possible d'obtenir l'autre effet ? Si oui, comment modifier l'instruction pour cela ?

Il est possible en C d'obtenir le pointeur vers l'emplacement mémoire d'une variable `x` avec l'expression `&x`, mais peu de langages actuels permettent un tel accès direct et illimité à ce pointeur. C++, java, python ou caml par exemple encapsulent les manipulations de pointeurs dans des constructions de plus haut niveau (objets, références, etc).

Pour définir une fonction exec en présence de pointeurs et d'aliasing, la notion d'environnement/de mémoire qui jusque là était simplement désignée par ρ doit être découpée en deux étages :

1. un environnement qui à chaque nom de variable associe l'emplacement mémoire (virtuel) associé,
2. une mémoire, qui à chaque emplacement mémoire virtuel associe la valeur qui y est stockée.

Ainsi, le phénomène d'*aliasing* se manifeste par l'existence dans l'environnement de deux variables associées au même emplacement mémoire, et qui partagent donc de fait une même valeur.

Mode de passage des paramètres

Considérons notre deuxième exemple.

```
void f(int x) {
  x = x + 1;
}
int y = 1;
f(y);
```

Qu'attendons-nous ici comme valeur finale pour `y` ? La réponse à cette question dépend de ce qui est transmis exactement à la fonction `f` lors de l'appel `f(y)`.

- Il peut s'agir de la valeur courante de `y`, à savoir 1. L'effet de l'instruction `x = x+1;` est alors d'affecter `1 + 1 = 2` à une variable locale `x`, sans effet sur `y` qui vaudra toujours 1.
- Il peut s'agir de l'emplacement mémoire occupé par `y`, auquel cas on peut imaginer que `y` remplace les deux occurrences de `x` dans l'affectation `x = x+1;`. Dans ce cas la valeur de `y` sera modifiée et passera à 2.

Ces deux scénarios correspondent à deux *modes de passage* des paramètres d'une fonction. Le premier correspond au *passage par valeur*, utilisé par défaut dans la plupart des langages. Dans cette situation les paramètres formels d'une fonction sont des variables locales à cette fonction, qui reçoivent les valeurs des paramètres effectifs. En tant que variables locales, les paramètres formels n'ont donc aucune interaction avec les variables extérieures. Le deuxième scénario correspond en revanche au *passage par référence*, possible dans certains langages (par exemple C++, pascal). Dans ce deuxième mode, les paramètres formels désignent les mêmes valeurs gauches que les paramètres effectifs.

Quizz : comment simuler un passage par référence en C, en caml, en python, en java ?

Accès à des variables non locales

Considérons notre troisième exemple.

```
int x = 1;
int f() {
  return x;
}
x = 2;
int y = f();
```

Qu'attendons-nous comme valeur renvoyée par l'appel `f()` final ? Autrement dit, quelle valeur de `x` doit-elle être prise en compte dans l'instruction `return x;` de la fonction `f` ?

- Si l'on considère la valeur qu'avait la variable `x` au moment de la définition de la fonction `f`, alors on retiendra la valeur 1.
- Si l'on considère la valeur qu'a la variable `x` au moment où l'instruction `return x;` est exécutée, alors on retiendra la valeur 2.

Les langages impératifs suivent en général le deuxième scénario.

Remarquez au passage que la premier scénario n'aurait de sens que dans les langages où la définition d'une fonction est une instruction comme les autres, qui est effectivement « exécutée ». Cela pourrait être le cas en python, ou dans une certaine mesure en caml (dans ce dernier, il s'agit d'une expression et non d'une instruction), mais pas en java.

Le deuxième scénario est en revanche incompatible avec certaines situations liées à la programmation fonctionnelle.

```
let plus n =
  let f x = x + n in
  f
in
let succ = plus 1 in
succ 2
```

Ici, on a une fonction `plus` définissant *et renvoyant* une fonction locale `f` [5]. La définition de `f` utilise une variable `n` qui est extérieure à `f`. En l'occurrence, cette variable `n` est également une variable locale à la fonction `plus` (c'est le paramètre formel d'une fonction obéissant au mode de passage par valeur). Autrement dit, la variable `n` n'a de valeur que dans le cadre d'un appel à la fonction `plus`, et disparaît sitôt l'appel terminé. Ainsi `succ` est une fonction faisant référence à une variable `n` qui n'existe plus : on ne peut imaginer en prendre « la valeur courante » et seul le premier scénario reste envisageable. La variable `n` rencontrée dans l'évaluation d'un appel à `subst` aura invariablement la valeur qu'elle avait lorsqu'a été définie la fonction `subst` (ou plus précisément lorsqu'a été définie la fonction `f` à laquelle `subst` est égale), à savoir 1.

Ainsi, lorsque l'on dit qu'en programmation fonctionnelle une fonction peut être une valeur comme les autres on simplifie un petit peu la situation : la *valeur* renvoyée par notre fonction `plus` précédente n'est pas simplement la fonction `f`, mais « la fonction `f` accompagnée d'un environnement donnant la valeur de la variable `n` à laquelle fait référence `f` » (et cette variable étant immuable, la notion de moment importe peu). De manière plus générale, une valeur-fonction est une fonction accompagnée d'un environnement donnant, au minimum, les valeurs de toutes les variables extérieures utilisées par cette fonction (pour faire simple, on peut se contenter de transmettre l'intégralité de l'environnement dans lequel la fonction a été définie). On appelle cet ensemble fonction/environnement une *clôture*.

$$
\begin{aligned}
\text{eval}(\text{Fun}(\text{x}, e), \rho) &= \text{Clos}(\text{x}, e, \rho) \\
\text{eval}(\text{App}(e_1, e_2), \rho) &= \begin{cases} \text{eval}(e_f, \rho_f[x_f \mapsto \text{eval}(e_2, \rho)]) & \text{si eval}(e_1, \rho) = \text{Clos}(x_f, e_f, \rho_f) \\ \text{indéfini sinon} \end{cases}
\end{aligned}
$$

2.7 Interprétation d'un langage fonctionnel

On considère le mini-langage FUN, qui illustre les bases de la programmation fonctionnelle. Voici un exemple de programme FUN.

```
let rec fact = fun n ->
  if n = 0 then
    1
  else
    n * fact (n-1)
in
fact 6
```

Dans un tel langage, on ne manipule que des expressions (il n'y a plus de notions d'instruction ou de séquence). Les expressions ont en revanche des formes bien plus variées que dans le langage IMP, puisqu'elles permettent en outre les expressions conditionnelles et la définition et l'application de fonctions (éventuellement récursives).

Syntaxe abstraite

On garde les parties déjà mentionnées pour les expressions logico-arithmétiques et les variables locales :

- un symbole `n` d'arité 0 pour chaque entier n,
- des symboles `Add`, `Sub`, `Mul` d'arité 2 désignant respectivement l'addition `+`, la soustraction `-` et la multiplication `*`,
- un symbole d'arité 0 pour chaque nom de variable,
- un symbole sorté `Let` d'arité 3 s'appliquant à un nom de variable et deux expressions pour la définition d'une variable locale.

On peut définir en caml un type correspondant à ce noyau :

```
type expr =
  (* arithmétique *)
  | Cst of int
  | Add of expr * expr
  | Sub of expr * expr
```

5. Nous nommons ici cette fonction pour les besoin de l'explication, mais le code caml `let plus n = fun x -> x + n in` ou même `let plus n x = x + n` aurait produit exactement le même effet.

```
  | Mul of expr * expr
  (* variables *)
  | Var of string
  | Let of string * expr * expr
```

On y ajoute pour avoir suffisamment d'expressivité :

- un symbole If d'arité 3 pour les expressions conditionnelles et un symbole Eq d'arité 2 pour le test d'égalité,
- un symbole sorté Fun d'arité 2 pour la création d'une fonction anonyme de la forme `fun x -> e`,
- un symbole App d'arité 2 pour l'application d'une fonction,
- un symbole sorté LetRecFun d'arité 4 pour représenter la définition d'une fonction récursive de la forme `let rec f x = e1 in e2`.

Notre type caml `expr` est étendu de la sorte pour représenter ces nouveaux termes.

```
  (* branchements *)
  | Eq of expr * expr
  | If of expr * expr * expr
  (* fonctions *)
  | Fun of string * expr
  | App of expr * expr
  | LetFun of string * string * expr * expr
```

Le type `value` représentant les résultats possibles de l'évaluation d'une expression est étendu. Il contient comme précédemment des nombres entiers, mais également des clôtures. Le type `env` des environnements est défini comme pour l'évaluateur de IMP à l'aide du module `Map`.

```
module Env = Map.Make(String)

type value =
  | VCst of int
  | VClos of string * expr * value Env.t
```

On peut alors définir une fonction d'évaluation `eval: expr -> value Env.t -> value` prenant en paramètre une expression et un environnement et renvoyant la valeur de cette expression.

```
let rec eval (e: expr) (env: value Env.t): value = match e with
  | Cst n -> VCst n
```

Comme les valeurs ont maintenant plusieurs formes possibles, l'évaluation d'une opération arithmétique demande maintenant de s'assurer que les valeurs des deux opérandes sont bien des entiers et non des clôtures, et de récupérer ces entiers. Cela nécessite une opération de filtrage que l'on factorise dans une fonction auxiliaire.

```
let rec eval_binop op e1 e2 env =
  match eval e1 env, eval e2 env with
  | VCst n1, VCst n2 -> VCst (op n1 n2)
  | _ -> assert false
```

On peut alors faire appel à cette fonction auxiliaire pour chaque opérateur binaire.

```
let rec eval (e: expr) (env: value Env.t): value = match e with
  ...
  | Add(e1, e2) -> eval_op (+) e1 e2 env
  | Sub(e1, e2) -> eval_op (-) e1 e2 env
  | Mul(e1, e2) -> eval_op ( * ) e1 e2 env
```

L'accès à une variable consulte l'environnement. La déclaration d'une variable locale avec `let x = e1 in e2` définit un environnement étendu associant `x` à la valeur de `e1`, et évalue l'expression `e2` dans ce nouvel environnement.

```
  | Var x -> Env.find x env
  | Let(x, e1, e2) ->
    let v1 = eval e1 env in
    eval e2 (Env.add x v1 env)
```

Lors d'un branchement conditionnel on vérifie d'abord que la valeur produite par le test est bien un entier, puis on évalue uniquement la branche concernée.

```
| Eq(e1, e2) ->
   eval_op (fun n1 n2 -> if n1=n2 then 1 else 0) e1 e2 env
| If(c, e1, e2) ->
   begin match eval c env with
   | VCst n -> if n <> 0 then
                 eval e1 env
               else
                 eval e2 env
   | _ -> assert false
   end
```

Une fonction anonyme produit directement une valeur : on la couple simplement avec l'environnement courant pour former une clôture. Dans l'évaluation d'une application, on vérifie que la valeur du membre gauche est bien une clôture, dont on extrait la fonction et l'environnement. On poursuit alors avec l'évaluation du corps de la fonction, dans l'environnement fourni par la clôture (ce dernier est simplement étendu d'une association entre le paramètre formel de la fonction et la valeur du paramètre effectif de l'application).

```
| Fun(x, e) -> VClos(x, e, env)
| App(f, a) ->
   begin match eval f env with
   | VClos(x, b, env') ->
      let va = eval a env in
      let env'' = Env.add x va env' in
      eval b env''
   | _ -> assert false
   end
```

Le cas de la définition d'une fonction récursive est particulier : la fonction `f` est représentée comme d'habitude par une clôture `c`, mais pour permettre les appels récursifs l'environnement de cette clôture doit contenir l'association entre `f` et la clôture `c` elle-même. Autrement dit la clôture doit être récursive. Pour cela on peut commencer par créer la clôture sans préciser son environnement, puis définir l'environnement étendu, et enfin *modifier* la clôture pour lui associer le bon environnement. Comme la structure de données que nous avons utilisée ici pour les clôtures n'est pas mutable, cela demande un peu de magie.

```
| LetRecFun(f, x, e1, e2) ->
   let c = VClos(x, e1, Env.empty) in
   let env' = Env.add f c env in
   (Obj.magic c).(2) <- env';
   eval e2 env'
```

3 Théorie des langages réguliers et analyse lexicale

L'analyse lexicale consiste à découper le texte d'un programme en une séquence de mots.

3.1 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

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

$$m = a_1 a_2 \ldots a_n$$

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 de chiffre maximal dans l'écriture d'un entier !).

Expressions et langages associés

Les ***expressions régulières*** sont des termes qui décrivent des ensembles de mots à l'aide des éléments que nous venons d'énumérer : caractères isolés, séquences, alternatives et répétitions.

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

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. Ce langage est spécifié comme suit.

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

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

Exemples, sur l'alphabet $\{\ a, b\ \}$:

— $(a|b)(a|b)(a|b)$: 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 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)

Reconnaissance

On dit qu'un mot m est ***reconnu*** 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 $m \in L(e_1 e_2) \iff \exists m_1, m_2,\ m = m_1 m_2 \wedge m_1 \in L(e_1) \wedge m_2 \in L(e_2)$ 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 reconnaissance 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 reconnaissent le mot vide ?
- $am \in L(e)$: 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 $L(e)$.

Reconnaissance du mot vide : on définit une fonction null telle que $\mathsf{null}(e)$ indique si $\varepsilon \in L$.

$$
\begin{aligned}
\mathsf{null}(\emptyset) &= \mathsf{F} \\
\mathsf{null}(\varepsilon) &= \mathsf{V} \\
\mathsf{null}(a) &= \mathsf{F} \\
\mathsf{null}(e_1 e_2) &= \mathsf{null}(e_1) \wedge \mathsf{null}(e_2) \\
\mathsf{null}(e_1|e_2) &= \mathsf{null}(e_1) \vee \mathsf{null}(e_2) \\
\mathsf{null}(e^*) &= \mathsf{V}
\end{aligned}
$$

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

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

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

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

let rec null = function
  | Vide -> false
  | Epsilon -> true
  | Caractere a -> false
  | Sequence(e1, e2) -> null e1 && null e2
  | Alternative(e1, e2) -> null e1 || null 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 null 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 reconnait e = function
  | [] -> null e
  | a::m -> reconnait (residu e a) m
```

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

Ainsi, les expressions régulières sont des termes décrivant des ensembles de mots. La suite du chapitre s'intéressera principalement aux points suivants :

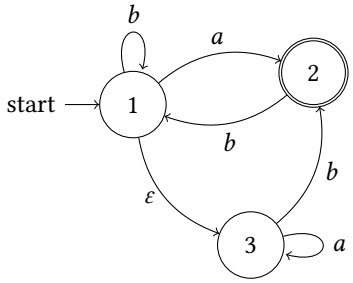
- quels ensembles de mots peuvent ou non être décrits par des expressions rationnelles ?
- quels outils algorithmiques permettent une reconnaissance la plus efficace possible ?
- comment transformer cela en un outil d'analyse lexicale ?

3.2 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, ont 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).



Formallement, un ***automate fini*** 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***.

On dit qu'un mot m sur un alphabet A est ***reconnu*** 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 :

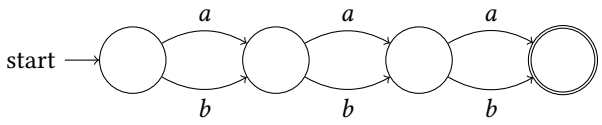
- part d'un des états initiaux : $q_0 \in I$,
- est étiqueté par les caractères de m, pris dans l'ordre : $m = a_1 a_2 \dots a_n$ et
- arrivé à l'un des états acceptants : $q_n \in F$.

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).

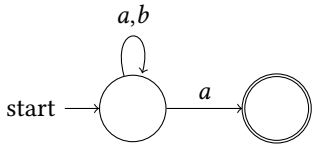
Le ***langage reconnu*** par un automate est l'ensemble de ses mots reconnus, c'est-à-dire l'ensemble des mots correspondant à l'ensemble des chemins d'un état initial à un état acceptant.

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

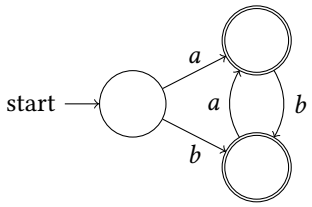
— Mots de 3 lettres sur l'alphabet { a, b }.



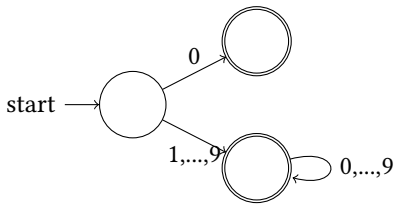
— Mots sur l'alphabet { a, b } terminant par un a.



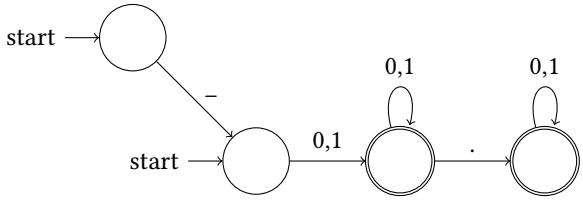
— Mots alternant a et b.



— 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, sans retour en arrière.

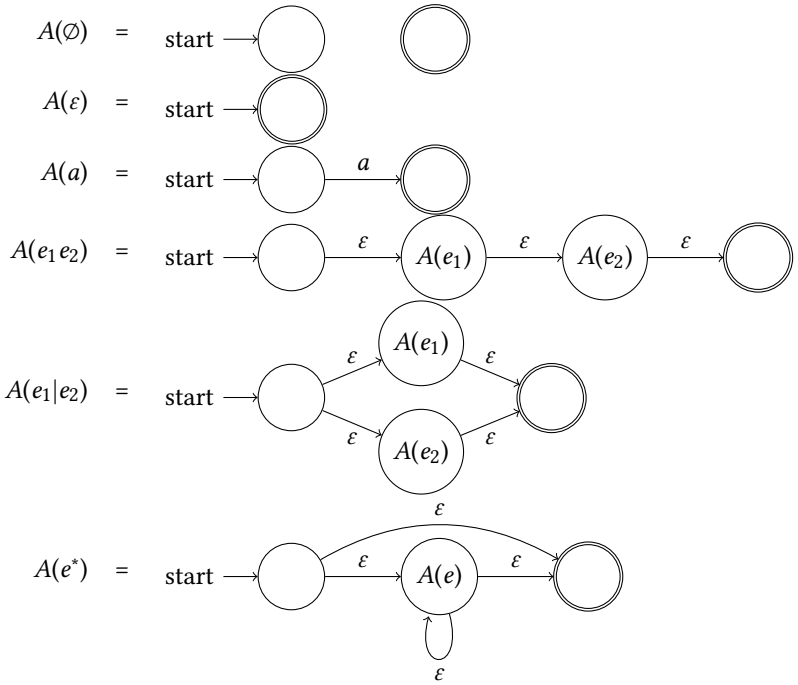
Les clés de lecture suivantes permettent de comprend le lien entre un automate et un tel algorithme :

— les états de l'automates 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).



Dans un tel automate, un même mot peut étiqueter de multiple chemins partant du ou des états initiaux. Le test de reconnaissance par recherche de chemins, bien que possible, est donc potentiellement coûteur. 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 :

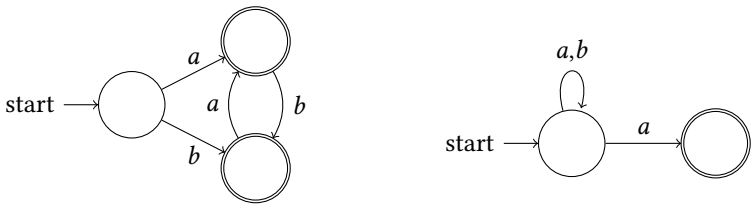
— un seul état initial,
— pas de transitions ε,
— partant de chaque état, au plus une transition par caractère.

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 est reconnu si l'état d'arrivée est acceptant.

Ainsi, s'il bien un chemin et que cet unique chemin mène à un état acceptant, alors le mot 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 n'est pas reconnu.

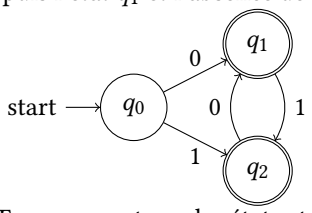
En reprenant ici deux exemples d'automates déjà présentés, celui de gauche est déterministe, mais pas celui de droite (partant de son état initial on trouve deux transitions avec l'étiquette a).



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 1 depuis l'état q_1 et l'absence de transition étiquetée 0 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];
}
```

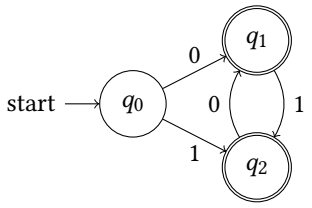
On peut alors écrire simplement une fonction de reconnaissance comme une boucle qui passe d'un état à l'autre en suivant les transitions. On suppose ici que les mots sont représentés par des listes d'entiers.

```
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

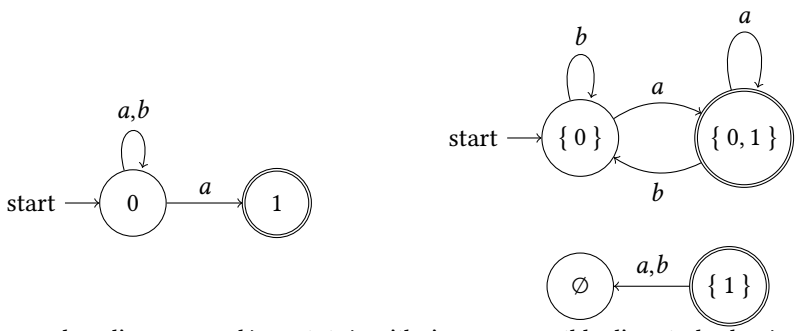
Tout automate peut être transformé en un automate déterministe ***équivalent***, c'est-à-dire reconnaissant le même langage. Ce processus de transformation est appelé ***déterminisation***.

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 $X \xrightarrow{a} X'$ dès lors qu'il existe dans l'automate d'origine un chemin $q \xrightarrow{\varepsilon} \dots \xrightarrow{\varepsilon}\xrightarrow{a}\xrightarrow{\varepsilon} \dots \xrightarrow{\varepsilon} q'$ avec $q \in X$ et $q' \in X'$. 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.

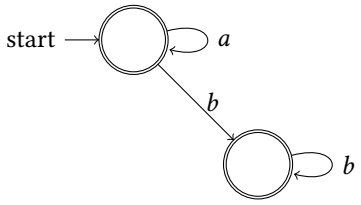
À retenir de ce processus de déterminisation : il assure que tout langage pouvant être reconnu par un automate fini quelconque peut l'être en particulier par un automate fini déterministe. En revanche, le nombre d'état peut augmenter grandement : l'automate déterminisé peut contenir $2^{|Q|}$ états !

3.3 Langages non reconnaissables

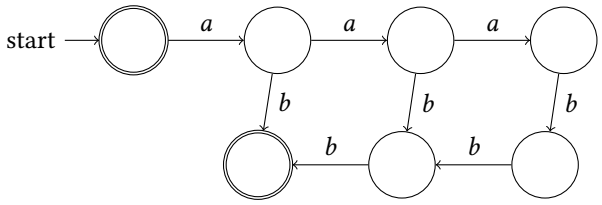
Les automates représentent des algorithmes travaillant avec une mémoire bornée. Nous allons voir ce que cela implique concernant les langages qui peuvent ou non être reconnus par un automate fini (déterministe ou non).

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

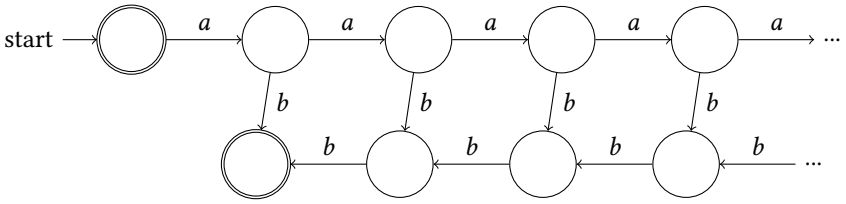
— $L_1 = \{ a^n b^m \mid n \in \mathbb{N} \wedge m \in \mathbb{N} \}$
Le langage L_1 est aussi décrit par l'expression régulière $a^* b^*$ et est reconnu par l'automate



— $L_2 = \{ a^n b^n \mid n \leqslant 3 \}$
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.



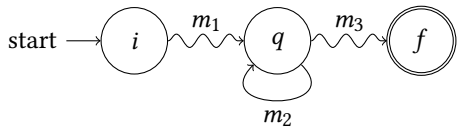
De manière générale, s'il faut des états différents pour distinguer chaque nombre possible de a lus (pour ensuite compter le bon nombre de b) alors on a besoin d'un nombre infini d'états, ce qui n'est pas permis dans avec un automate *fini*.

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 paragraph 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.

3.4 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 être 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.

Mise en jambes : algorithme de Roy-Warshall

Considérons un graphe orienté G dont les sommets sont numérotés de 1 à N, représenté par sa matrice d'adjacence. On cherche à déterminer, pour toute paire (i, j) de sommets, s'il existe un chemin de i vers j dans le graphe G.

L'algorithme de Roy-Warshall répond à ce problème en construisant une séquence de matrices booléennes $(A^k)_{0 \leqslant k \leqslant N}$ telle que $A^k_{i,j}$ vaut Vrai si et seulement s'il existe

dans G un chemin de i vers j n'utilisant que des sommets intermédiaires de numéro inférieur ou égal à k. Note : on utilise ici la lettre A comme initiale de « accessibilité ».

On construit dans un premier temps la matrice A^0. On veut, pour tous i, j, que $A^0_{i,j}$ vale Vrai s'il est possible d'aller de i à j en passant uniquement par des sommets de numéro $\leqslant 0$. Comme la numérotation des sommets commence à 1, cela signifie donc aller de i à j sans passer par aucun sommet intermédiaire. C'est possible uniquement si $i = j$ ou s'il existe une arête directe de i vers j. On pose donc $A^0_{i,j}$ = Vrai pour $i = j$ ou s'il existe une arête $i \longrightarrow j$ dans G, et $A^0_{i,j}$ = Faux sinon.

On va ensuite construire successivement les matrices A^k pour $k > 0$, en déduisant la matrice A^{k+1} de la matrice précédente A^k. Clé du raisonnement : j est accessible depuis i en utilisant uniquement des sommets intermédiaires $\leqslant k+1$ si et seulement si l'une des deux conditions suivantes est vérifiée :

- j est déjà accessible depuis i en utilisant uniquement des sommets intermédiaires $\leqslant k$, ou
- on peut d'abord aller de i à $k+1$ avec des sommets intermédiaires $\leqslant k$, puis de $k+1$ à j avec encore des sommets intermédiaires $\leqslant k$.

On utilise donc la formule suivante :

$$A^{k+1}_{i,j} = A^k_{i,j} \vee (A^k_{i,k+1} \wedge A^k_{k+1,j})$$

la dernière matrice d'accessibilité A^N indique ainsi, pour chaque paire de sommets (i, j), s'il existe un chemin de i vers j utilisant des sommets intermédiaires $\leqslant N$, c'est-à-dire sans restriction sur les sommets intermédiaires.

Algorithme de McNaughton et Yamada

Nous allons maintenant transposer le principe de l'algorithme de Roy-Warshall du monde des graphes vers celui des automates, pour produire une matrice d'expressions régulières décrivant pour chaque paire d'états (q_i, q_j) l'ensemble des mots pouvant étiqueter un chemin de q_i à q_j.

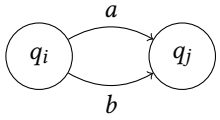
On part d'un automate (Q, T, I, F) dont les états sont numérotés de 1 à N. On va construire une séquence de matrice $(E^k)_{0 \leqslant k \leqslant N}$ telle que $E^k_{i,j}$ est une expression régulière décrivant l'ensemble des mots étiquetant un chemin de l'automate allant de q_i à q_j en ne passant que par des états intermédiaires de numéro $\leqslant k$.

On pourra ensuite obtenir une expression régulière décrivant l'ensemble des mots reconnus par l'automate en prenant l'alternative entre les expressions régulières $E^N_{i,f}$ pour tout état initial $q_i \in I$ et tout état acceptant $q_f \in F$:

$$E^N_{i_1,f_1} \mid \ldots \mid E^N_{i_k,f_l}$$

Initialisation : construction de E^0. On veut une expression régulière décrivant tous les passages directs de l'état q_i à l'état q_j.

- Si $i \neq j$, on prend l'alternative entre les étiquettes de toutes les transitions allant de l'état q_i à l'état q_j.



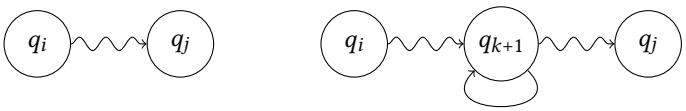
$a \mid b$

Note : on inclut également ε dans l'alternative s'il y a une transition ε.

- Si $i = j$, on prend l'alternative entre les étiquettes de toutes les transitions allant de l'état q_i à lui-même, *et* ε.

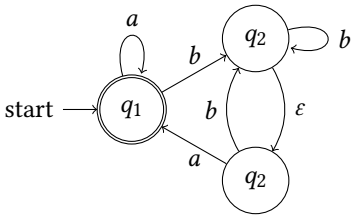
$\varepsilon \mid a$

Itération : construction de E^{k+1}. Pour définir $E^{k+1}_{i,j}$, on combine les expressions rationnelles décrivant les chemins de q_i à q_j qui ne passent pas par q_{k+1} et celles décrivant des chemins allant de q_i à q_{k+1} puis de q_{k+1} à q_j après avoir éventuellement fait plusieurs boucles autour de q_{k+1}.



$$E_{i,j}^{k+1} = E_{i,j}^{k} \mid E_{i,k+1}^{k}(E_{k+1,k+1}^{k})^* E_{k+1,j}^{k}$$

Exemple de tels calculs sur l'automate



$$\begin{aligned} E_{3,2}^{1} &= E_{3,2}^{0} \mid E_{3,1}^{0}(E_{1,1}^{0})^* E_{1,2}^{0} \\ &= b \mid a(\varepsilon|a)^* b \\ &= b \mid aa^* b \end{aligned} \qquad \begin{aligned} E_{3,3}^{2} &= E_{3,3}^{1} \mid E_{3,2}^{1}(E_{2,2}^{1})^* E_{2,3}^{1} \\ &= \varepsilon \mid (b|aa^* b)(\varepsilon|b)^* \varepsilon \\ &= \varepsilon \mid (b|aa^* b)b^* \end{aligned}$$

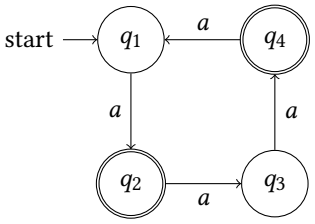
3.5 Minimisation

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 $\sim_N = \sim_{N+1} = \sim_{N+2} = \sim_{N+3} = \cdots$

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

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

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.

3.6 Analyse lexicale

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 d'un programme extrayant du 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 prend 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 maintenant 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, " ")
# prochain_mot ();;
- : mot * string = (PLUS, "+")
```

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 gloutonne 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'etat visite n'est pas l'etat puits, on poursuit
         l'analyse. En revanche, si l'etat est acceptant on met a 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 a l'etat puits ou a la fin de la chaine n'est plus
         necessairement un echec : c'est a ce moment que l'on consulte
         le dernier etat acceptant rencontre et que l'on peut renvoyer
         un lexeme (et mettre a jour la position de depart 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 exemple dès la séquence suivante sur les outils de la famille lex.

3.7 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.

Structure : 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 :

```
  | <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 commence et termine également par deux zones entre accolades dans lesquelles 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 un 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 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 autre 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) 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 suivante.

```
  | [' ' '\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.

```
  | "/**"  { lines := !lines + 1; 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`.

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 première règle. 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 caractère, 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 au prochain 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.

3.8 Application 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";
        (* a completer *) ];
    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 différent 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 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
}
```