Langages, interprétation, compilation

Thibaut Balabonski @ Université Paris-Saclay
http://www.lri.fr/~blsk/CompilationLDD3/
V1.5, été 2022

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
```

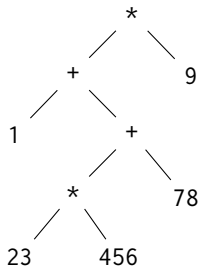
Une fois saisie sur le clavier d'une calculatrice ou d'un ordinateur, cette expression prend la forme d'une chaîne de caractères, formée de la suite de caractères suivante.

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

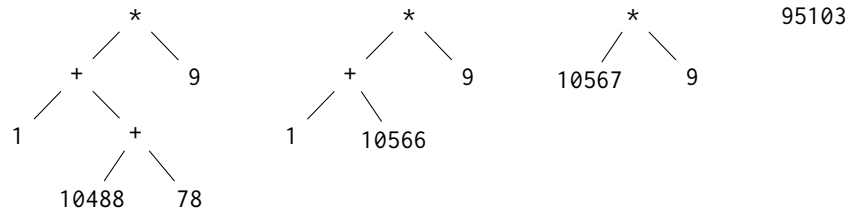
Cette séquence est plate, et ses éléments atomisés. Nous allons devoir l'analyser pour reconstruire sa structure et la manipuler réellement comme une expression arithmétique et non comme une simple suite de symboles. La première étape, appelée ***analyse lexicale***, consiste à regrouper les symboles 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 calculer 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.

```
      | ' ' -> 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.

```
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

```
  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 $1 + 2 \times 3$ 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 différents types de langages de programmation et y verrons de nouvelles techniques de preuve.

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).

```
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

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.

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 trad = function
  | Cst n       -> [ICst n]
  | Add(e1, e2) -> (trad e1) @ [IPush] @ (trad e2) @ [IAdd]
  | Mul(e1, e2) -> (trad e1) @ [IPush] @ (trad 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 `trad`.

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

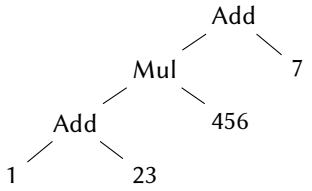
Notre calculatrice du chapitre précédent permettait d'écrire des calculs variées. Par exemple :

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

Ces trois écritures ont beau être différentes, elle représentent toutes 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. Il s'agit d'un texte, une chaîne de caractères. À l'inverse, la ***syntaxe abstraite*** donne une représentation structurée du programme, sous la forme d'un arbre. 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. 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.

Différences entre les deux niveaux de syntaxe 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 correctement 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 pointeurs [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`.

Structure récursive de la syntaxe abstraite Observons que la syntaxe abstraite des expressions arithmétique présente une structure récursive. En effet, une expression arithmétique e est :

- soit la combinaison de deux expressions plus petites e_1 et e_2 à l'aide d'un opérateur `+` ou `*`,
- soit une simple constante (entière, positive).

Autrement dit, on a deux manières de construire une expression arithmétique :

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

1. une constante entière positive suffit à former une expression (de la forme la plus simple qui soit),
2. deux expressions e_1 et e_2 déjà formées peuvent être combinées à l'aide d'un opérateur + ou * pour former une nouvelle expression (plus grande).

Ce principe peut être généralisé à la syntaxe abstraite de tout langage de programmation, et a bien d'autres choses.

2.2 Structures inductives

Les arbres de syntaxe abstraite des programmes ont une structure ***inductive***, que l'on peut décrire à l'aide d'une forme de récurrence : un programme est construit en combinant plusieurs fragments de programmes, eux-mêmes obtenus par combinaisons d'éléments plus petits, etc. Cette structure permet de facilement définir des *fonctions récursives* s'appliquant à des arbres de syntaxe abstraite, ou encore de raisonner sur les programmes à l'aide du *principe d'induction structurelle*.

Objets inductifs On définit un ensemble d'objets inductifs en décrivant :

1. des objets de base,
2. des manières de combiner des objets déjà construits pour en construire un plus grand.

On s'intéresse alors à l'ensemble des objets qui peuvent être construits en utilisant les deux points précédents.

On formalise cette description par un ensemble Σ de symboles appelés ***constructeurs***, dont chacun correspond soit à un objet de base, soit à une manière de construire un nouvel objet. Chaque constructeur est associé à un nombre entier positif ou nul appelé ***arité***. On définit les objets de l'ensemble $\mathsf{T}(\Sigma)$ à l'aide de la ***signature*** Σ de la manière suivante :

1. si c est un symbole de Σ d'arité 0, alors c est dans $\mathsf{T}(\Sigma)$,
2. si c est un symbole d'arité $n > 0$ et si t_1, ..., t_n sont dans $\mathsf{T}(\Sigma)$, alors $\mathsf{c}(t_1, ..., t_n)$ est dans $\mathsf{T}(\Sigma)$.

Un objet de $\mathsf{T}(\Sigma)$ construit par application répétée des deux critères précédents est appelé un ***terme***. Notez que le premier critère est un *cas de base* : il est vrai sans condition pour tout symbole d'arité 0 (on appelle ces symboles des ***constantes***), tandis que le deuxième critère est un *cas récursif* : pour que $\mathsf{c}(t_1, ..., t_n)$ appartienne à $\mathsf{T}(\Sigma)$ on demande que chacun des objets t_1 à t_n appartienne déjà à cet ensemble. Les deux critères sont appelés des ***propriétés de clôture***.

On demande en outre (mais on laisse ce point informel pour l'instant) que les deux critères précédents décrivent intégralement l'ensemble $\mathsf{T}(\Sigma)$, 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 $\mathsf{Add}(1, \mathsf{Mul}(2, 3))$. Vérifions au passage que $\mathsf{Add}(1, \mathsf{Mul}(2, 3))$ appartient bien à l'ensemble $\mathsf{T}(\Sigma_a)$: en temps que symboles d'arité nulle, 2 et 3 sont des termes, en outre Mul est un symbole d'arité 2 et donc $\mathsf{Mul}(2, 3)$ est bien un terme. De plus, 1 est un terme et Add un symbole d'arité 2, donc $\mathsf{Add}(1, \mathsf{Mul}(2, 3))$ est bien un terme.

Pour alléger l'écriture, on peut se donner des ***notations*** plus proches de l'écriture naturelle pour les constructions $\mathsf{Add}(e_1, e_2)$ et $\mathsf{Mul}(e_1, e_2)$. Ici, on utilisera les symboles $\oplus$ et $\otimes$, que l'on garde bien distincts des opérateurs + et × classiques :

— le constructeur Add (ou sa notation $\oplus$) est un élément syntaxique, qui s'applique à deux expressions pour en construire une troisième,
— l'opérateur + est un élément mathématique, qui s'applique à deux nombres pour en calculer un troisième.

Attention cependant, on utilisera toujours cette notation avec suffisamment de parenthèses pour éviter toute ambiguïté sur la structure des expressions. On s'interdira donc d'écrire $e_1 \oplus e_2 \oplus e_3 \oplus e_4$ et on choisira à la place l'une des cinq structures possibles, comme par exemple $(e_1 \oplus e_2) \oplus (e_3 \oplus e_4)$ ou $e_1 \oplus ((e_2 \oplus e_3) \oplus e_4)$. On évitera même de se reposer sur les conventions mathématiques de priorité : on utilisera le parenthésage explicite $1 \oplus (2 \otimes 3)$ pour représenter l'expression mathématique usuelle $1 + 2 \times 3$.

Définition de fonctions sur un ensemble d'objets inductifs L'ensemble $\mathsf{T}(\Sigma)$ étant intégralement défini par des propriétés de clôture, ces même propriétés nous donnent un schéma pour définir des fonctions s'appliquant aux 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 c d'arité n non nulle, donner une équation exprimant $f(\mathsf{c}(t_1, ..., t_n))$ 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).

Les éléments précédents permettent ainsi de calculer la valeur de f pour chaque objet pouvant être construit à partir de la signature Σ.

Voici par exemple trois ensembles d'équations sur nos termes représentant des expressions arithmétiques. Ces équations définissent des fonctions nbCst, nbOp et eval, telles que $\mathsf{nbCst}(e)$ donne le nombre de constantes dans l'expression arithmétique e, $\mathsf{nbOp}(e)$ le nombre d'opérateurs apparaissant dans e, et $\mathsf{eval}(e)$ donne la valeur obtenue en effectuant le calcul décrit par e. Remarquez que l'écriture de telles fonctions demande de bien distinguer les expressions arithmétiques elles-mêmes (la *syntaxe*) de la valeur associé (la *sémantique*). En particulier, le constructeur Add est un élément syntaxique représentant une addition (un constructeur), à ne pas confondre avec l'opérateur + qui est l'interprétation mathématique de l'addition (une fonction).

$$\begin{cases} \mathsf{nbCst}(\mathsf{n}) &= 1 \\ \mathsf{nbCst}(e_1 \oplus e_2) &= \mathsf{nbCst}(e_1) + \mathsf{nbCst}(e_2) \\ \mathsf{nbCst}(e_1 \otimes e_2) &= \mathsf{nbCst}(e_1) + \mathsf{nbCst}(e_2) \end{cases}$$

$$\begin{cases} \mathsf{nbOp}(\mathsf{n}) &= 0 \\ \mathsf{nbOp}(e_1 \oplus e_2) &= 1 + \mathsf{nbOp}(e_1) + \mathsf{nbOp}(e_2) \\ \mathsf{nbOp}(e_1 \otimes e_2) &= 1 + \mathsf{nbOp}(e_1) + \mathsf{nbOp}(e_2) \end{cases}$$

$$\begin{cases} \mathsf{eval}(\mathsf{n}) &= n \\ \mathsf{eval}(e_1 \oplus e_2) &= \mathsf{eval}(e_1) + \mathsf{eval}(e_2) \\ \mathsf{eval}(e_1 \otimes e_2) &= \mathsf{eval}(e_1) \times \mathsf{eval}(e_2) \end{cases}$$

Principe d'induction structurelle On peut montrer que tous les éléments d'un ensemble $\mathsf{T}(\Sigma)$ de termes vérifient une certaine propriété P en vérifiant que chaque manière possible de construire un terme est bien compatible avec P. Autrement dit :

1. pour tout $\mathsf{c} \in \Sigma$ d'arité 0, on vérifie que la propriété $P(\mathsf{c})$ est vraie, et
2. pour tous $\mathsf{c} \in \Sigma$ 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, on vérifie que la propriété $P(\mathsf{c}(t_1, ..., t_n))$ est encore vraie.

Une fois ces vérifications faites, on obtient la garantie que les règles de clôture définissant $\mathsf{T}(\Sigma)$ ne permettent pas de construire un objet qui ne vérifierait pas P. Autrement dit, pour tout élément $t \in \mathsf{T}(\Sigma)$ la propriété $P(t)$ est nécessairement vraie.

Cette technique de preuve est une ***preuve par récurrence***, qu'on appelle plus précisément *preuve par récurrence sur la structure des termes*, ou encore ***preuve par induction structurelle***. 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 $P(\mathsf{c}(t_1, ..., t_n))$ 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 récursifs 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 la

propriété P est vraie. On a alors

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

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

— 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 $\mathbb{N} \times \text{liste} \to \text{liste}$ (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 $P(\mathsf{Nil})$ 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.

Pour alléger la manipulation des objets inductifs, on se donne encore des notations pour les objets de base et les différentes manières de combiner plusieurs objets à l'aide d'un constructeur : on pourra ainsi noter `[]` la liste vide, et $e::l$ la liste formée en plaçant l'élément e en tête de la liste l, et retrouver les notations familières de caml. Avec ces notations, on peut réexprimer le principe d'induction sur les listes de la manière suivante. Soit P une propriété telle que

— $P([])$ est vraie, et

— pour tout e et tout l vérifiant $P(l)$, la propriété $P(e::l)$ est encore vraie.

Alors $P(l)$ est vraie pour toute liste l.

2.3 Variables et environnements

Une variable est un nom désignant une valeur stockée en mémoire [2].

```
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 chaque nom de variables associe sa valeur.

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

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

$$\begin{aligned} \mathsf{eval}(\mathsf{n}, \rho) &= n \\ \mathsf{eval}(\mathsf{x}, \rho) &= \rho(\mathsf{x}) \\ \mathsf{eval}(e_1 \oplus e_2, \rho) &= \mathsf{eval}(e_1, \rho) + \mathsf{eval}(e_2, \rho) \\ \mathsf{eval}(e_1 \otimes e_2, \rho) &= \mathsf{eval}(e_1, \rho) \times \mathsf{eval}(e_2, \rho) \end{aligned}$$

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. Pour l'instant, on se concentre sur la version fonctionnelle, dont la description théorique est plus simple.

Version fonctionnelle : définition de variables locales immuables 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 d'une valeur à la variable `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.

On se donne un constructeur ternaire Let s'appliquant à une variable et deux expressions, et on conserve la notation `let x =` e_1 `in` e_2 déjà connue pour représenter le terme $\mathsf{Let}(\mathsf{x}, e_1, e_2)$. 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.

$$\mathsf{eval}(\texttt{let x =}\ e_1\ \texttt{in}\ e_2, \rho) = \mathsf{eval}(e_2, \rho[x \mapsto \mathsf{eval}(e_1, \rho)])$$

Masquage 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`.

Preuve d'une propriété sémantique Démontrons que la valeur d'une expression e ne dépend que des variables effectivement présentes dans e. Autrement dit, si deux environnements ρ et ρ' donnent les mêmes valeurs aux variables de e on a à coup sûr $\mathsf{eval}(e, \rho) = \mathsf{eval}(e, \rho')$.

Cet énoncé demande une petite clarification, car il y a deux manières différentes dont une variable `x` peut « apparaître » dans une expression e :

— la variable x peut être introduite par un `let`, à l'intérieur de e (on parle de variable ***liée***, ou locale),
— ou e peut faire référence à une variable x supposée définie par ailleurs (on parle de variable ***libre***).

La notion qui nous intéresse dans l'énoncé précédent est celle de variable libre. L'ensemble $\mathsf{fv}(e)$ des ***variables libres*** d'une expression e est défini par les équations

$$\begin{aligned}
\mathsf{fv}(\mathsf{n}) &= \emptyset \\
\mathsf{fv}(\mathsf{x}) &= \{x\} \\
\mathsf{fv}(e_1 \oplus e_2) &= \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\
\mathsf{fv}(e_1 \otimes e_2) &= \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\
\mathsf{fv}(\texttt{let x = } e_1 \texttt{ in } e_2) &= \mathsf{fv}(e_1) \cup (\mathsf{fv}(e_2) \setminus \{x\})
\end{aligned}$$

On peut maintenant formaliser la propriété à démontrer :

Soient e une expression et ρ, ρ' deux environnements.
Si pour tout $x \in fv(e)$ on a $\rho(x) = \rho'(x)$,
alors $eval(e, \rho) = eval(e, \rho')$.

On démontre cette propriété par induction structurelle sur l'expression e. La propriété $P(e)$ qui nous intéresse est donc

« pour tous ρ, ρ', si $\forall x \in \mathsf{fv}(e), \rho(x) = \rho'(x)$, alors on a $\mathsf{eval}(e, \rho) = \mathsf{eval}(e, \rho')$ »

Le raisonnement par induction structurelle présente un cas pour chaque construction de la syntaxe abstraite.

— Cas n. Quelque soient ρ et ρ' on a $\mathsf{eval}(\mathsf{n}, \rho) = \mathsf{n} = \mathsf{eval}(\mathsf{n}, \rho')$.
— Cas x. Soient ρ, ρ' tels que $(\forall y \in \mathsf{fv}(\mathsf{x}), \rho(y) = \rho'(y))$, c'est-à-dire tels que $\rho(\mathsf{x}) = \rho'(\mathsf{x})$. On a $\mathsf{eval}(\mathsf{x}, \rho) = \rho(\mathsf{x}) = \rho'(\mathsf{x}) = \mathsf{eval}(\mathsf{x}, \rho')$.
— Cas Add. Soient e_1 et e_2 deux expressions telles que les propriétés $P(e_1)$ et $P(e_2)$ sont valides. Soient ρ, ρ' tels que $\forall x \in \mathsf{fv}(e_1 \oplus e_2), \rho(x) = \rho'(x)$. Comme $\mathsf{fv}(e_1 \oplus e_2) = \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2)$, on a en particulier $\forall x \in \mathsf{fv}(e_1), \rho(x) = \rho'(x)$ et $\forall x \in \mathsf{fv}(e_2), \rho(x) = \rho'(x)$. Donc par $P(e_1)$ on a $\mathsf{eval}(e_1, \rho) = \mathsf{eval}(e_1, \rho')$ et par $P(e_2)$ on a $\mathsf{eval}(e_2, \rho) = \mathsf{eval}(e_2, \rho')$. Finalement,

$$\begin{array}{lll}
& \mathsf{eval}(e_1 \oplus e_2, \rho) & \\
= & \mathsf{eval}(e_1, \rho) + \mathsf{eval}(e_2, \rho) & \text{par déf. de eval} \\
= & \mathsf{eval}(e_1, \rho') + \mathsf{eval}(e_2, \rho') & \text{par hyp. d'induction} \\
= & \mathsf{eval}(e_1 \oplus e_2, \rho') & \text{par déf. de eval}
\end{array}$$

— Cas Mul similaire.
— Cas Let. Soient e_1 et e_2 deux expressions telles que les propriétés $P(e_1)$ et $P(e_2)$ sont valides. Soient ρ, ρ' tels que $\forall y \in \mathsf{fv}(\texttt{let x = } e_1 \texttt{ in } e_2), \rho(y) = \rho'(y)$. Par définition de eval on a

$$\mathsf{eval}(\texttt{let x = } e_1 \texttt{ in } e_2, \rho) = \mathsf{eval}(e_2, \rho[x \mapsto \mathsf{eval}(e_1, \rho)])$$

et

$$\mathsf{eval}(\texttt{let x = } e_1 \texttt{ in } e_2, \rho') = \mathsf{eval}(e_2, \rho'[x \mapsto \mathsf{eval}(e_1, \rho')])$$

On peut conclure par hypothèse d'induction $P(e_2)$, à condition de justifier que les deux environnements $\rho[x \mapsto \mathsf{eval}(e_1, \rho)]$ et $\rho'[x \mapsto \mathsf{eval}(e_1, \rho')]$ coïncindent sur $\mathsf{fv}(e_2)$. Soit $y \in \mathsf{fv}(e_2)$. On considère deux cas.
  — Si $y \neq x$, alors $y \in (\mathsf{fv}(e_2) \setminus \{x\})$ et donc $y \in \mathsf{fv}(\texttt{let x = } e_1 \texttt{ in } e_2)$. Alors par hypothèse $\rho(y) = \rho'(y)$ et on a donc

$$\begin{array}{lll}
& (\rho[x \mapsto \mathsf{eval}(e_1, \rho)])(y) & \\
= & \rho(y) & y \neq x \\
= & \rho'(y) & \text{par hyp.} \\
= & (\rho'[x \mapsto \mathsf{eval}(e_1, \rho')])(y) & y \neq x
\end{array}$$

  — Si $y = x$, alors

$$\begin{array}{lll}
& (\rho[x \mapsto \mathsf{eval}(e_1, \rho)])(y) & \\
= & \mathsf{eval}(e_1, \rho) & y = x \\
= & \mathsf{eval}(e_1, \rho') & \text{par hyp. d'induction } P(e_1) \\
= & (\rho'[x \mapsto \mathsf{eval}(e_1, \rho')])(y) & y = x
\end{array}$$

Les deux environnements donnent donc toujours la même valeur pour $y \in \mathrm{fv}(e_2)$, donc par hypothèse d'induction $P(e_2)$ on conclut que $\mathrm{eval}(\texttt{let x = } e_1 \texttt{ in } e_2, \rho) = \mathrm{eval}(\texttt{let x = } e_1 \texttt{ in } e_2, \rho')$.

Exercice 2.1. Dans un langage d'expressions arithmétiques où les variables locales sont immuables, l'expression `let x =` e_1 `in` 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.

$$
\begin{array}{rcl}
\mathsf{n}[x := s] & = & \mathsf{n} \\
\mathsf{y}[x := s] & = & \begin{cases} s & \text{si } x = y \\ \mathsf{y} & \text{sinon} \end{cases} \\
(e_1 \oplus e_2)[x := s] & = & (e_1[x := s]) \oplus (e_2[x := s]) \\
(e_1 \otimes e_2)[x := s] & = & (e_1[x := s]) \otimes (e_2[x := s]) \\
(\texttt{let y = } e_1 \texttt{ in } e_2)[x := s] & = & \begin{cases} \texttt{let y = } (e_1[x := s]) \texttt{ in } e_2 & \text{si } x = y \\ \texttt{let y = } (e_1[x := s]) \texttt{ in } (e_2[x := s]) & \text{si } x \neq y \text{ et } y \notin \mathrm{fv}(s) \end{cases}
\end{array}
$$

Pour justifier l'équivalence entre la construction Let et la substitution, démontrer la propriété

$$\forall x, e_1, e_2, \rho,\ x \notin \mathrm{fv}(e_1) \ \Rightarrow\ \mathrm{eval}(\texttt{let } x = e_1 \texttt{ in } e_2, \rho) \ =\ \mathrm{eval}(e_2[x := e_1], \rho)$$

par induction sur la struture de e_2.

2.4 Manipulation de syntaxes abstraites et d'environnements dans un programme

La syntaxe abstraite d'un programme est une structure de données qui peut être manipulée par un autre programme, par exemple un interprète ou un compilateur.

Représentation en caml d'une syntaxe abstraite On peut définir un type caml représentant nos expressions arithmétiques avec variables locales en introduisant un constructeur caml pour chaque symbole de notre syntaxe abstraite.

```
type expr =
  (* arithmétique *)
  | Cst of int
  | Add of expr * expr
  | Mul of expr * expr
  (* variables *)
  | Var of string
  | Let of string * expr * expr
```

Alors l'expression `let x = 41 in x + 1` est représentée par la valeur caml

```
Let("x", Cst 41, Add(Var "x", Cst 1))
```

Valeurs et environnements On se donne pour représenter les valeurs produites par les expressions arithmétiques un type `value` désignant simplement des nombres entiers.

```
type value = int
```

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***, qui peut être réalisée à l'aide de différentes structures de données. Citons notamment :

- les arbres de recherche équilibrés (module `Map` de caml), qui permettent de représenter un environnement comme une structure de données immuable, en style purement fonctionnel,
- les tables de hachage (module `Hashtbl` de caml), qui forment une structure de données mutable.

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 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 tables 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, car elle est plus facile à mettre en œuvre.

Écriture d'un interprète Maintenant que l'on a mis en place les éléments pour représenter une expression (à l'aide de sa syntaxe abstraite) et un environnement (à l'aide d'un tableau associatif), on peut écrire une fonction caml

```
eval: expr -> env -> value
```

telle que `eval` e ρ renvoie la valeur de l'expression e évaluée dans l'environnement ρ. Il suffit pour cela d'écrire un cas pour chacune des équations déjà vues pour eval.

```
let rec eval e env = match e with
  | Cst n -> n
  | Add(e1, e2) -> eval e1 env + eval e2 env
  | Mul(e1, e2) -> eval e1 env * eval 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
     let env' = Env.add x v1 env in
     eval e2 env'
```

On en déduit une fonction `eval_top` procédant à l'évaluation d'une expression dans l'enrivonnement vide.

```
let eval_top (e: expr): value =
  eval e Env.empty
```

Exercice 2.2. Écrire un interprète pour les expressions précédentes, en utilisant une table de hachage plutôt qu'une structure `Map`. Attention, il y a un piège.

2.5 Interprétation d'un langage fonctionnel

On va maintenant étendre les principes vus ci-dessus pour les appliquer au 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 manipule des expressions plus riches qu'aux sections précédentes, contenant notamment des expressions conditionnelles, ainsi que 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
  | 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
```

Clôtures et valeurs En programmation fonctionnelle, les fonctions sont des valeurs comme les autres, qui peuvent être passées en paramètres à d'autres fonctions ou renvoyées comme résultats. Il y a cependant une petite subtilité.

```
let plus n =
  let f x = x + n in
  f
```

Ici, on a une fonction `plus` définissant *et renvoyant* une fonction locale `f` [3]. La définition de `f` utilise une variable `n` qui est extérieure à `f` (selon le vocabulaire déjà vu : une variable libre). En l'occurrence, cette variable `n` est un paramètre de la fonction `plus` [4]. Deux appels `plus 2` et `plus 3` produisent deux fonctions différentes. Toutes deux correspondent à `fun x -> x + n`, mais avec $n = 2$ dans le premier cas et $n = 3$ dans le deuxième cas.

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` ». 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 paire fonction/environnement une ***clôture***.

3. 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.

4. On reviendra sur quelques aspects techniques de ceci dans le paragraphe sur les modes de passagee des paramètres.

$$\text{eval}(\texttt{fun x -> } e, \rho) = \langle \texttt{fun x -> } e, \rho \rangle$$
$$\text{eval}(e_1\ e_2, \rho) = \begin{cases} \text{eval}(e_f, \rho_f[x_f \mapsto \text{eval}(e_2, \rho)]) & \text{si eval}(e_1, \rho) = \langle \texttt{fun } x_f \texttt{ -> } e_f, \rho_f \rangle \\ \text{indéfini sinon} \end{cases}$$

Le type `value` représentant les résultats possibles de l'évaluation d'une expression est donc étendu. Il contient comme précédemment des nombres entiers, mais également des clôtures.

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

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

La valeur entière 3 est représentée par `VCst 3`, et la valeur correspondant à la fonction `fun x -> e` définie dans l'environnement ρ est représentée par `VClos(x, e, ρ)`.

Interprétation On peut maintenant définir une fonction d'évaluation `eval: expr -> env -> value` prenant en paramètres une expression et un environnement et renvoyant la valeur de cette expression.

```
let rec eval e env = match e with
  | Cst n -> VCst n
  | Var x -> Env.find x env
  | Let(x, e1, e2) ->
     let v1 = eval e1 env in
     eval e2 (Env.add x v1 env)
```

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 env = 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
```

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
        eval b (Env.add x va 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 [5].

```
  | 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'
```

2.6 Variables mutables

Nous avons dit précédemment : une variable est un nom désignant une valeur stockée en mémoire.

```
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
```

Nous avons vu jusqu'ici le cas des variables *immuables*, qui reçoivent une valeur définitive à leur création. 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.

Donnons-nous un nouveau constructeur binaire Set pour désigner une instruction d'affectation `x = e;`. On notera également `x := e` le terme Set(x, *e*) représentant l'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

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

Notez que l'application répétée de la règle pour la séquence i_1 ; i_2 de deux instructions permet de traiter une séquence de longueur arbitraire :

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

Calculons l'état de l'environnement après l'exécution de la séquence d'affectations suivante, en prenant un environnement initialement vide.

5. La fonction `Obj.magic: 'a -> 'b` permet de court-circuiter le système de types de caml. Elle ne doit pas être considérée comme une fonction de bibliothèque ordinaire et n'est que *très* rarement pertinente. Je n'en revendique personnellement que deux utilisations en quinze ans, la présente comprise.

```
x = 6
y = x + 1
x = x * y
```

$$
\begin{aligned}
&\mathsf{exec}(\mathtt{x := 6; y := } x \oplus 1\mathtt{; x := } x \otimes y,\ \emptyset)\\
=\ &\mathsf{exec}(\mathtt{x := } x \otimes y,\ \mathsf{exec}(\mathtt{y := } x \oplus 1,\ \mathsf{exec}(\mathtt{x := 6},\ \emptyset)))\\
=\ &\mathsf{exec}(\mathtt{x := } x \otimes y,\ \mathsf{exec}(\mathtt{y := } x \oplus 1,\ [x \mapsto 6]))\\
&\qquad \mid \mathsf{car}\ \mathsf{eval}(6, \emptyset) = 6\\
=\ &\mathsf{exec}(\mathtt{x := } x \otimes y,\ [x \mapsto 6, y \mapsto 7])\\
&\qquad \mid \mathsf{car}\ \mathsf{eval}(x \oplus 1, [x \mapsto 6]) = 7\\
=\ &[x \mapsto 42, y \mapsto 7]\\
&\qquad \mid \mathsf{car}\ \mathsf{eval}(x \otimes y, [x \mapsto 6, y \mapsto 7]) = 42
\end{aligned}
$$

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.

Représentation en caml et interprétation On peut se donner des définitions caml minimales pour des expressions arithmétiques avec variables, et des séquences d'instructions d'affectation. On conserve les notions de valeurs et d'environnement déjà utilisées, on garde l'essentiel de la définition des expressions simples, et on n'ajoute que la notion d'instruction.

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

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

type instr =
  | Set of string * expr
```

La fonction `eval: expr -> env -> value` peut être définie comme on l'a déjà fait pour les variables immuables, et on ajoute ici deux fonctions `exec: instr -> env -> env` et `exec_seq: instr list -> env -> env` respectivement pour l'exécution d'une instruction et pour l'exécution d'une séquence. L'une et l'autre renvoient comme résultat un nouvel environnement tenant compte des affectations effectuées.

```
let exec i env = match i with
  | Set(x, e) -> let v = eval e env in
                 Env.add x v env

let rec exec_seq s env = match s with
  | []      -> env
  | i :: s' -> let env' = exec i env in
               exec_seq s' env'

let exec_top (s: instr list): env =
  exec_seq s Env.empty
```

Remarquez que l'on vient d'écrire une fonction d'interprétation pour des instructions manipulant des variables mutables, à l'aide uniquement d'une structure de données immuable. C'est tout à fait possible ! Et c'est au plus proche des équations mathématiques écrites à la page précédente.

Variante avec structure de données mutable Bien que ce ne soit techniquement pas indispensable et que, à terme, cela complexifie le raisonnement sur les programmes, on peut réaliser une variante des fonctions `exec` précédentes à l'aide d'une structure de données mutable comme une table de hachage.

```
type value = int
type env = (string, value) Hashtbl.t
```

La fonction d'évaluation est quasiment identique, mais utilise cette fois la fonction `find` de la bibliothèque `Hashtbl` (petit détail historique : cette fonction prend ses arguments dans l'ordre inverse par rapport à celle de `Map`).

```
let rec eval (e: expr) (env: env): value = match e with
  | Cst n -> n
  | Var x -> Hashtbl.find env x
  | Add(e1, e2) -> eval e1 env + eval e2 env
  | Mul(e1, e2) -> eval e1 env * eval e2 env
```

Les nouvelles fonctions `exec` et `exec_seq` prennent toujours un environnement `env` en paramètre, mais n'ont plus besoin de renvoyer l'environnement modifié en résultat. À la place, on effectue des modifications en place de `env`, ici à l'aide de la fonction `Hashtbl.replace: env -> string -> value -> unit`[6] qui crée une nouvelle association entre une clé et une valeur, ou remplace l'association précédente si la clé existait déjà. On a donc les nouveaux types `exec: instr -> env -> unit` et `exec_seq: instr list -> env -> unit`.

```
let exec i env = match i with
  | Set(x, e) -> let v = eval e env in
                 Hashtbl.replace env x v

let rec exec_seq s env = match s with
  | []      -> ()
  | i :: s' -> exec i env; exec_seq s' env
```

On charge ensuite la fonction principale `exec_top` de créer une table vide dans laquelle commencer l'exécution. Ici on demande à cette dernière de renvoyer la table à la fin, pour pouvoir consulter son état.

```
let exec_top (s: instr list): env =
  let env = Hashtbl.create 64 in
  exec_seq s env;
  env
```

Cette nouvelle fonction d'interprétation respecte toujours, d'une certaine manière, les équations de sémantique, mais du fait de la mutation de la table de hachage cette correspondance n'est plus si immédiate. Avec les équations de sémantique et l'interprète purement fonctionnel, on pouvait écrire des égalités de la forme

$$\text{exec}(s, \rho) = \rho'$$

liant deux environnements ρ et ρ' distincts. Avec les tables de hachage en revanche, on n'a plus qu'un unique environnement, passant par plusieurs états. On retrouve un lien avec les équations précédentes en distinguant un unique environnement `env` (une structure de données) et ses états successifs ρ_1, ρ_2, ρ_3, ... (différentes fonctions des variables vers les valeurs). Dans l'interprète impératif, les équations s'appliquent exclusivement aux états ρ_i de la structure mutable `env`, et pas à la structure elle-même. L'équation $\text{exec}(s, \rho) = \rho'$ se traduit alors par le fait que l'exécution de la séquence `s` fait passer l'environnement `env` de l'état ρ à l'état ρ'.

2.7 Variables, adresses et partage

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

6. En caml, `unit` est le type des expressions qui ne renvoient pas de résultat. Il s'applique notamment à des expressions qui agissent uniquement par effet de bord, comme `Hashtbl.replace`, qui sont similaires à la notion d'instruction d'un langage comme C ou java. Techniquement, le type `unit` contient une unique valeur, appelée *unité*, et notée `()`.

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. Reprenons l'exemple d'une fonction `plus`, qui attend un argument `n` et renvoie une fonction dépendant de `n`.

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

On a vu que cette fonction `plus` renvoie une fonction `f` comportant une variable libre `n`. 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. On a donc bien besoin ici de la notion de clôture.

2.8 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

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

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 parties 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 [7].

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 :

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

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.

2.9 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;
```

7. 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*.

```
  n = n + (-1);
}
print(r);
```

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, Lt, And... d'arité 2 désignant respectivement l'addition +, la multiplication *, la comparaison <, la conjonction &&...

On peut définir en caml le type correspondant (ici en factorisant les opérations binaires).

```
type bop = Add | Mul | Lt | And
type expr =
  | Cst  of int
  | Var  of string
  | Bop  of bop * 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,
- un symbole Print s'appliquant à une expression pour l'affichage d'un nombre sur la sortie standard.

On peut définir en caml le type correspondant

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

La séquence d'instructions IMP donnée plus haut est ainsi représentée en ocaml par la liste

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

Interprétation On va reprendre le principe de l'évaluateur utilisant une table de hachage pour l'environnement mutable, avec une notation un peu plus compacte. Pour cela, remarquons que toutes les fonctions `eval`, `exec` et `exec_seq` partagent une unique table `env`. On peut donc définir cette table une fois pour toute à l'extérieur, puis définir nos trois fonctions sans qu'elles aient besoin de prendre l'environnement en paramètre. La fonction `eval`, par exemple, n'a plus besoin de prendre comme argument que l'expression à évaluer, et on peut ensuite conserver les règles déjà données pour les expressions.

```
let exec_prog (p: seq): unit =
  let env = Hashtbl.create 64 in

  let rec eval: expr -> value = function
    | Cst n -> n
    | Var x -> Hashtbl.find env x
```

```
    | Bop(Add, e1, e2) -> eval e2 + eval e1
    | Bop(Mul, e1, e2) -> eval e2 * eval e1
```

Notez une différence par rapport à la première version : l'ordre des opérandes des opérations binaires est inversé, pour assurer que `e1` est bien évalué avant `e2` (car caml évalue ces éléments de droite à gauche). Pour ne pas dépendre de ceci, on aurait également pu évaluer les deux expressions dans l'ordre de notre choix à l'aide de **`let`** avant de combiner les résultats : **`let`** `v1 = eval e1` **`in let`** `v2 = eval e2` **`in`** `v1+v2`.

On a comme nouveauté une décision à prendre quant à la valeur produite par le nouvel opérateur de 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.

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

On donne également à la conjonction sa sémantique paresseuse, en ne faisant intervenir l'opérande de droite que lorsque cela est nécessaire.

```
    | Bop(Lt,  e1, e2) -> if eval e2 > eval e1 then 1 else 0
    | Bop(And, e1, e2) -> if eval e1 = 0 then 0 else eval e2
```

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.

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

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

Pour exécuter une séquence d'instructions, il suffit d'itérer la fonction `exec`.

```
  and exec: instr -> unit = function
    | Set(x, e) -> let v = eval e in Hashtbl.replace env x v
    | If(e, s1, s2) ->
       if eval e <> 0 then exec_seq s1 else exec_seq s2
    | While(e, s) as i ->
       if eval e <> 0 then (exec_seq s; exec i)
    | Print(e) -> let v = eval e in Printf.printf "%d\n" v
  and exec_seq (s: seq): unit =
    List.iter exec s
  in
  exec_seq p
```

Voici quelques étapes de l'exécution de notre programme exemple, en partant de l'environement vide. On a d'abord une phase initialisant les variables `n` et `r`, puis une succession d'étapes dans la boucle **`while`**.

$$\begin{aligned}
&\text{exec}(\texttt{n := 6;r := 1;while...}, \emptyset) \\
=\ &\text{exec}(\texttt{while...}, \text{exec}(\texttt{r := 1}, \text{exec}(\texttt{n := 6}, \emptyset))) \\
=\ &\text{exec}(\texttt{while...}, [n \mapsto 6, r \mapsto 1]) \\
=\ &\text{exec}(\texttt{while...}, \text{exec}(\texttt{r := } r \otimes n\texttt{;n := } n \oplus (-1), [n \mapsto 6, r \mapsto 1])) \\
&\quad \text{car} \left| \begin{array}{l} \text{eval}(0 \lessgtr n, [n \mapsto 6, r \mapsto 1]) \ = \ 1 \\ \quad \text{car} \left| \begin{array}{l} \text{eval}(0, [n \mapsto 6, r \mapsto 1]) \ = \ 0 \\ \qquad < \ 6 \ = \ \text{eval}(n, [n \mapsto 6, r \mapsto 1]) \end{array} \right. \end{array} \right. \\
=\ &\text{exec}(\texttt{while...}, [n \mapsto 5, r \mapsto 6]) \\
=\ &\text{exec}(\texttt{while...}, [n \mapsto 4, r \mapsto 30]) \\
=\ &\text{exec}(\texttt{while...}, [n \mapsto 3, r \mapsto 120]) \\
=\ &\text{exec}(\texttt{while...}, [n \mapsto 2, r \mapsto 360]) \\
=\ &\text{exec}(\texttt{while...}, [n \mapsto 1, r \mapsto 720]) \\
=\ &\text{exec}(\texttt{while...}, [n \mapsto 0, r \mapsto 720]) \\
=\ &[n \mapsto 0, r \mapsto 720] \\
&\quad \text{car} \left| \begin{array}{l} \text{eval}(0 \lessgtr n, [n \mapsto 0, r \mapsto 720]) \ = \ 0 \\ \quad \text{car} \left| \begin{array}{l} \text{eval}(0, [n \mapsto 0, r \mapsto 720]) \ = \ 0 \\ \qquad \not< \ 0 \ = \ \text{eval}(n, [n \mapsto 0, r \mapsto 720]) \end{array} \right. \end{array} \right.
\end{aligned}$$

Si l'on tentait de faire un tel raisonnement sur le programme `x := 0;while (1) {x :=` $x \oplus 1$`}`, le calcul aurait une forme différente.

$$
\begin{aligned}
&\texttt{exec(x := 0;while (1) \{x :=}\ x \oplus 1\texttt{\}}, \emptyset) \\
=\ &\texttt{exec(while (1) \{x :=}\ x \oplus 1\texttt{\}}, [x \mapsto 0]) \\
&\ \ \texttt{car eval}(1, [x \mapsto 0]) = 1 \\
=\ &\texttt{exec(while (1) \{x :=}\ x \oplus 1\texttt{\}}, [x \mapsto 1]) \\
&\ \ \texttt{car eval}(1, [x \mapsto 1]) = 1 \\
=\ &\texttt{exec(while (1) \{x :=}\ x \oplus 1\texttt{\}}, [x \mapsto 2]) \\
&\ \ \texttt{car eval}(1, [x \mapsto 2]) = 1 \\
=\ &\ldots
\end{aligned}
$$

et ainsi de suite sans jamais pouvoir atteindre un résultat. 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 exec(`x := 0;while (1) {x :=` $x \oplus 1$`}`, $\emptyset$) = ρ. Cet exemple met en évidence le fait 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). Notez en outre que l'on peut observer le même phénomène dans le langage FUN en utilisant des fonctions récursives.

Ajout de fonctions On peut étendre notre langage IMP en ajoutant la possibilité de définir et d'utiliser des fonctions. On apporte trois modifications au langage pour cela.

— Un programme n'est plus composé seulement d'une séquence d'instructions, mais comporte également un ensemble de définitions de fonctions.

```
type prog = {
    functions: fun_def list;
    main: seq;
  }
```

— On introduit un nouveau type `fun_def` pour décrire les fonctions. Chaque fonction a des paramètres et peut introduire en outre des variables locales. Le corps de la fonction est donné par une séquence d'instructions. Pour simplifier, on impose ici que chaque fonction se termine par une unique instruction `return` *e*, et on isole l'expression *e* dans un champ dédié [8].

```
type fun_def = {
    name:   string;
    params: string list;
    locals: string list;
    body:   seq;
    return: expr;
  }
```

— On ajoute une nouvelle forme d'expression : l'application d'une fonction (identifiée par son nom) à une liste d'arguments.

```
type expr =
  ...
  | Call of string * expr list
```

Il ne reste qu'à étendre notre fonction `exec_prog` pour qu'elle traite ces nouveaux cas. Pour faire le lien entre les identifiants et les définitions des fonctions, on crée en plus de l'environnement `env` une nouvelle table `fenv`, initialisée à l'aide de la liste de définitions données par le champ `functions` du programme.

```
let exec_prog p =
  let env = Hashtbl.create 64 in
  let fenv = Hashtbl.create 64 in
  List.iter (fun fdef -> Hashtbl.add fenv fdef.name fdef)
    p.functions;
```

8. Ceci sera généralisé en TP.

Le code des fonctions `exec` et `exec_seq` est inchangé, et celui de la fonction `eval` est simplement étendu d'un nouveau cas associé au constructeur `Call`. Dans ce cas, on évalue chaque argument puis on fait appel à une fonction d'interprétation dédiée `exec_call` (notre interprète impose donc le mode de passage des paramètres « par valeur »). L'exécution du programme complet suit la séquence d'instructions `p.main`.

```
let rec eval = function
  ...
  | Call(f, args) -> exec_call f (List.map eval args)
and exec i = ...
and exec_seq s = ...
and exec_call (f: string) (args: value list): value = ...
in
exec_seq p.main
```

La fonction `exec_call` a pour principal rôle d'exécuter le code de la fonction appelée, qu'elle obtient en allant chercher la définition de la fonction dans la table `fenv`.

```
and exec_call (f: string) (args: value list): value =
  let fdef = Hashtbl.find fenv f in
  ...
```

Cette exécution est faite en deux étapes : d'abord exécuter la séquence d'instructions `fdef.body`, puis évaluer (et renvoyer) l'expression finale `fdef.return`.

```
  ...
  exec_seq fdef.body;
  eval fdef.return
  ...
```

À ceci, il faut cependant ajouter une gestion de l'environnement `env`, pour associer à chaque paramètre formel la valeur qui a été donnée en argument, et pour permettre la bonne gestion des variables locales. On procède en trois étapes :

1. avant d'exécuter le code de la fonction, ajouter de nouvelles associations dans `env` pour les paramètres formels,
2. puis (toujours avant d'exécuter le code de la fonction), ajouter de nouvelles associations dans `env` pour les variables locales, ici initialisées par défaut à la valeur `0`,
3. après l'exécution du code de la fonction, retirer de `env` les variables locales et les paramètres.

Le code complet de la fonction `exec_call` est donc le suivant.

```
and exec_call (f: string) (args: value list): value =
  let fdef = Hashtbl.find fenv f in
  List.iter2 (Hashtbl.add env) fdef.params args;
  List.iter (fun x -> Hashtbl.add env x 0) fdef.locals;
  exec_seq fdef.body;
  let v = eval fdef.return in
  List.iter (fun x -> Hashtbl.remove env x) fdef.locals;
  List.iter (Hashtbl.remove env) fdef.params;
  v
in
```

Note : cet interprète fixe implicitement une règle pour la gestion des collisions de noms. Dans le corps d'une fonction `f`, les paramètres formels masquent les éventuelles variables globales qui porteraient les mêmes noms, et les variables locales masquent à la fois les variables globales et les paramètres. Autrement dit, pour savoir à quoi fait référence un nom de variable `x` dans le corps d'une fonction `f`, on applique les trois critères suivants, dans l'ordre :

1. si `f` contient une variable locale `x`, alors c'est à elle que `x` fait référence,
2. sinon, si un paramètre formel de `f` a le nom `x`, alors c'est à lui que `x` fait référence,
3. sinon, `x` fait référence à une variable globale.

En dehors de toute définition de fonction, c'est-à-dire dans la séquence d'instructions `p.main`, on n'a en revanche accès qu'aux variables globales. Notre code impose cette discipline en vertu d'une propriété des tables de hachage de caml :

— une association ajoutée avec `Hashtbl.add` *masque* une éventuelle précédente association de même nom, sans la supprimer,

— la fonction `Hashtbl.remove` ne retire qu'*une* association du nom demandé, plus précisément la dernière.

Ainsi, après avoir ajouté deux associations pour un même nom `"x"` et appelé une seule fois `Hashtbl.remove`, on retrouve la première association.

```
let env = Hashtbl.create 64 in
Hashtbl.add env "x" 1;    (* "x" -> 1                          *)
Hashtbl.add env "x" 2;    (* "x" -> 2 active, x -> 1 masquée *)
Hashtbl.remove env "x";   (* "x" -> 1                          *)
```

C'est exactement ce qui se passe dans notre interprète à la sortie de la fonction `exec_call` : les associations pour les paramètres formels et les variables locales sont retirées de la table, ce qui, de fait, restaure les éventuelles associations précédentes de même nom (qui concernaient alors des variables globales).

Pour simplement *modifier* une association déjà existante, sans en empiler une nouvelle, on utilise la fonction `Hashtbl.replace` plutôt que `add` (c'est bien ce que fait notre fonction `exec` en présence d'une instruction d'affectation `Set`).

Pour finir, notez que cet ajout des fonctions est, d'un point de vue sémantique, loin d'être anodin. Jusqu'ici, l'évaluation d'une expression ne causait aucun effet de bord, et en particulier ne modifiait pas l'environnement. Maintenant au contraire, une expression peut contenir un appel de fonction, qui modifie certaines variables globales. Quasiment toutes les équations de eval et exec doivent alors être modifiées pour en tenir compte. Les équations pour le branchement conditionnel deviendraient par exemple :

$$\text{exec}(\texttt{if (}e\texttt{) \{}b_1\texttt{\} else \{}b_2\texttt{\}}, \rho) \quad = \quad \begin{cases} \text{exec}(b_1, \rho') & \text{si eval}(e, \rho) = (v, \rho') \text{ avec } v \neq 0 \\ \text{exec}(b_2, \rho') & \text{sinon} \end{cases}$$

En l'occurrence, le code caml de l'interprète présenté dans cette section a été écrit directement avec ceci en tête. Il a donc bien le comportement attendu, même si ce n'est pas immédiatement visible dans son code : tout repose sur les modifications de la table de hachage, qui sont faites à la volée et dans le bon ordre. Tout ceci aurait au contraire été explicite dans une version utilisant une structure immuable de type `Map`.

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***.

$$m = a_1 a_2 \dots a_n$$

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

Dans ce chapitre, on appelle ***langage*** un ensemble de mots. Un langage L sur l'alphabet A est donc un sous-ensemble de A^*.

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

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

Les symboles comme `+` ou `;` correspondent à des caractères isolés, et les mots-clés à des séquences prédéfinies de symboles. Les nombres introduisent quelques idées additionnelles : comme l'alternative entre plusieurs formats, ou la répétition arbitraire de certains types de symboles (on ne fixe pas a priori de nombre 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 $m \in L(e_1e_2) \iff \exists m_1, m_2,\ m = m_1m_2 \wedge m_1 \in L(e_1) \wedge m_2 \in L(e_2)$ qui donnent une spécification mathématique tout à fait convenable, mais une complexité algorithmique déplorable : faut-il essayer toutes les manières de décomposer m en deux parties, et tester pour chacune $m_1 \in L(e_1)$ puis $m_2 \in L(e_2)$?

Pour obtenir un algorithme de 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 $\varepsilon \in L$.

$$\begin{aligned}
\mathsf{null}(\emptyset) &= \mathsf{F} \\
\mathsf{null}(\varepsilon) &= \mathsf{V} \\
\mathsf{null}(a) &= \mathsf{F} \\
\mathsf{null}(e_1e_2) &= \mathsf{null}(e_1) \wedge \mathsf{null}(e_2) \\
\mathsf{null}(e_1|e_2) &= \mathsf{null}(e_1) \vee \mathsf{null}(e_2) \\
\mathsf{null}(e^*) &= \mathsf{V}
\end{aligned}$$

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 $L(e)$.

$$\begin{aligned}
\mathsf{résidu}(\emptyset, a) &= \emptyset \\
\mathsf{résidu}(\varepsilon, a) &= \emptyset \\
\mathsf{résidu}(b, a) &= \begin{cases} \varepsilon & \text{si } b = a \\ \emptyset & \text{si } b \neq a \end{cases} \\
\mathsf{résidu}(e_1e_2, a) &= \begin{cases} \mathsf{résidu}(e_1, a)e_2 & \text{si } \varepsilon \notin L(e_1) \\ \mathsf{résidu}(e_1, a)e_2 \mid \mathsf{résidu}(e_2, a) & \text{si } \varepsilon \in L(e_1) \end{cases} \\
\mathsf{résidu}(e_1|e_2, a) &= \mathsf{résidu}(e_1, a) \mid \mathsf{résidu}(e_2, a) \\
\mathsf{résidu}(e^*) &= \mathsf{résidu}(e, a)e^*
\end{aligned}$$

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.

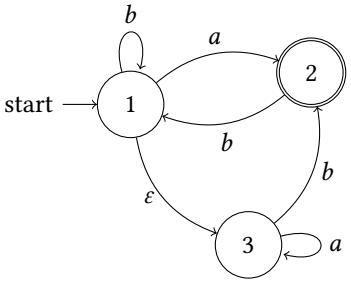
Ainsi, les expressions régulières sont des termes décrivant des ensembles de mots. La suite du chapitre s'intéressera principalement aux points suivants :

- 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***,

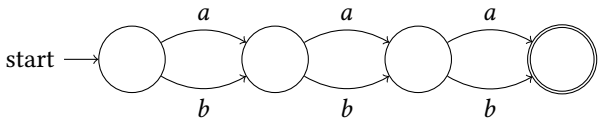
— T est un ensemble de triplets de $Q \times (A \cup \{\varepsilon\}) \times Q$ appelés ***transitions***,
— I est un ensemble d'états ***initiaux***, et
— F est un ensemble d'états ***acceptants***.

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 : $q_n \in F$.

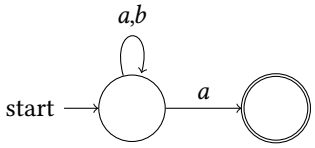
Note : dans la lecture des étiquettes du chemin, on ignore les ε (mais on fait bien la transition entre l'état de départ et l'état d'arrivée d'une éventuelle transition ε).

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.

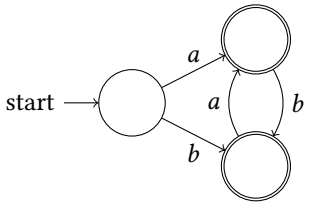
Voici quelques exemples de langages déjà évoqués, et des automates les reconnaissant.
— Mots de 3 lettres sur l'alphabet $\{a, b\}$.



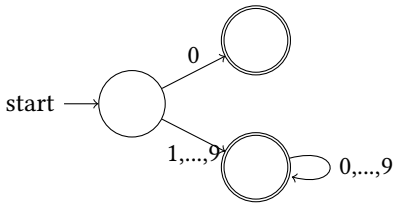
— Mots sur l'alphabet $\{a, b\}$ terminant par un a.



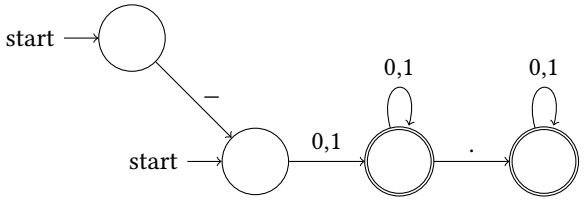
— Mots alternant a et b.



— 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,

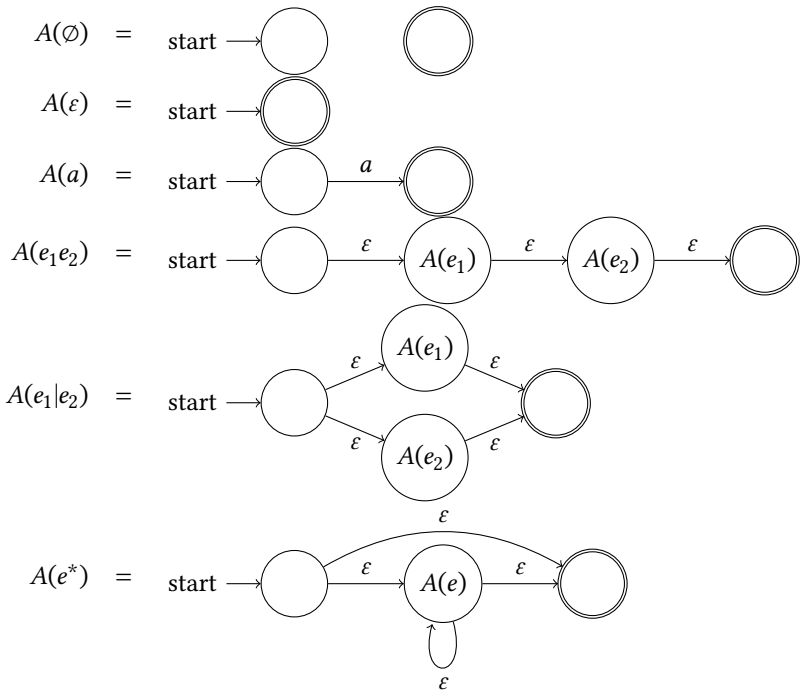
— il lit son entrée de gauche à droite, sans retour en arrière.

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 :

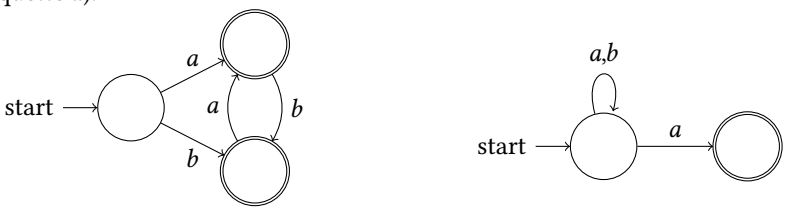
— un seul état initial,
— pas de transitions ε,
— partant de chaque état, au plus une transition par caractère.

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.

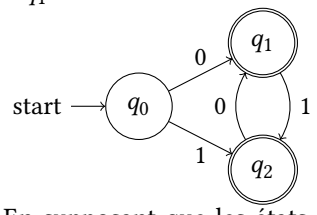
Ainsi, s'il bien un chemin et que cet unique chemin mène à un état acceptant, alors le mot est reconnu. Sinon, c'est-à-dire s'il n'existe pas de chemin ou si l'unique chemin mène à un état qui n'est pas acceptant, alors le mot n'est pas reconnu.

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.

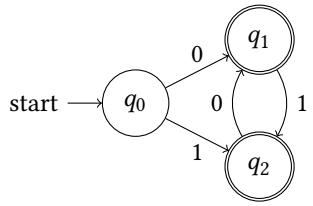
```
type automate = {
  initial: int;
  transitions: int option array array;
  accepting: int list;
}

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
```

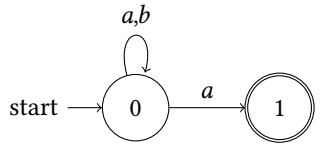
Déterminisation Tout automate peut être transformé en un automate déterministe ***équivalent***, c'est-à-dire reconnaissant le même langage. Ce processus de transformation est appelé ***déterminisation***.

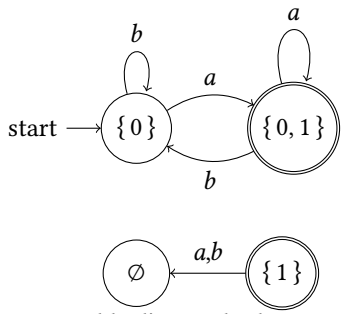
Lorsque l'on considère un automate non déterministe (Q, T, I, F), chaque mot d'entrée est susceptible de correspondre à plusieurs chemins. Ces chemins correspondentaux différents choix possible d'un état initial dans I, à l'emprunt d'éventuelles transitions ε, et aux choix faits entre plusieurs transitions 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 !

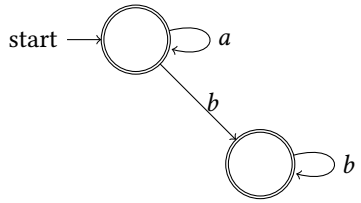
3.3 Langages non reconnaissables

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

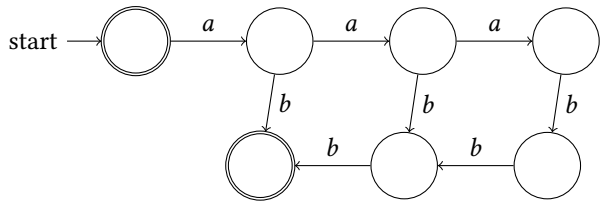
par un automate fini (déterministe ou non).

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

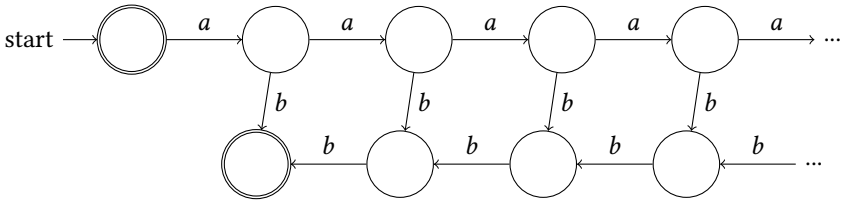
— $L_1 = \{ a^n b^m \mid n \in \mathbb{N} \wedge m \in \mathbb{N} \}$
Le langage L_1 est aussi décrit par l'expression régulière a^*b^* et est reconnu par l'automate



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

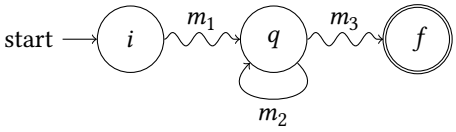


De manière générale, s'il faut des états différents pour distinguer chaque nombre possible de a lus (pour ensuite compter le bon nombre de b) alors on a besoin d'un nombre infini d'états, ce qui n'est pas permis dans avec un automate *fini*.

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 $i \to j$ 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.

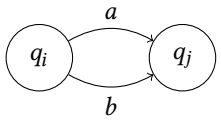
On part d'un automate (Q, T, I, F) dont les états sont numérotés de 1 à N. On va construire une séquence de matrice $(E^k)_{0 \leqslant k \leqslant N}$ telle que $E^k_{i,j}$ est une expression régulière décrivant l'ensemble des mots étiquetant un chemin de l'automate allant de q_i à q_j en ne passant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



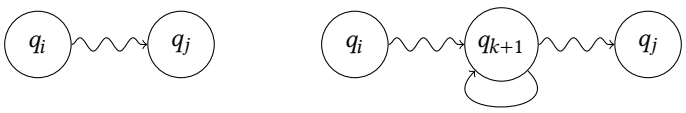
$a \mid b$

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.

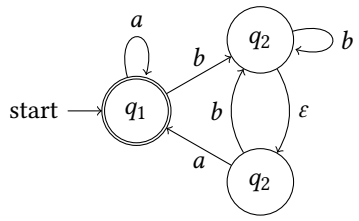
$\varepsilon \mid a$

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}.



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

Exemple de tels calculs sur l'automate



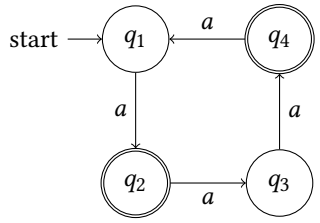
$$\begin{aligned} E^1_{3,2} &= E^0_{3,2} \mid E^0_{3,1}(E^0_{1,1})^* E^0_{1,2} \\ &= b \mid a(\varepsilon|a)^* b \\ &= b \mid aa^* b \end{aligned} \qquad \begin{aligned} E^2_{3,3} &= E^1_{3,3} \mid E^1_{3,2}(E^1_{2,2})^* E^1_{2,3} \\ &= \varepsilon \mid (b|aa^*b)(\varepsilon|b)^*\varepsilon \\ &= \varepsilon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

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.

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

On définit alors l'équivalence par

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

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

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

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

Pour tout état $q \in Q$, notons $[q]$ la classe d'équivalence de q. L'automate quotient $(Q_\sim, T_\sim, i_\sim, F_\sim)$ est un automate fini déterministe complet défini par les éléments suivants :

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

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

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

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

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

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

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

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

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

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

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

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

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

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

On peut même caractériser à l'aide des langages résiduels un automate déterministe complet de taille minimale, appelé ***automate des résiduels***. Prenons un langage reconnaissable L, doté d'un ensemble fini $\{\, R_1, \ldots, R_k \,\}$ de résiduels. Définition de l'automate des résiduels :
— les états sont les résiduels R_1, ..., R_k,
— la fonction de transition est définie par $R.a = a^{-1}R$,
— l'état initial est $\varepsilon^{-1}L$, c'est-à-dire L,
— les états acceptants sont les R_i tels que $\varepsilon \in R_i$.

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

résiduel $L(q)$, et que cette correspondance est compatible avec les fonctions de transition de l'automate quotient et de l'automate des résiduels.

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

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

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

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

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

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.

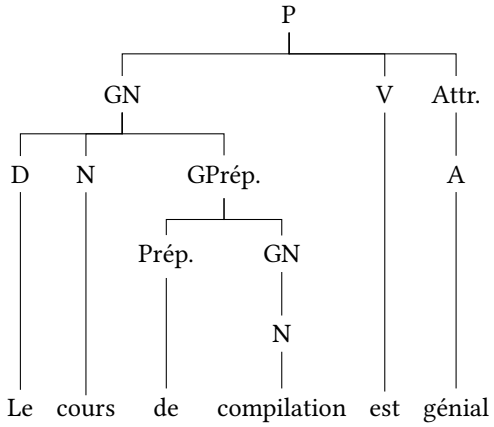
```
  | "(*"  { print "(*"; comment lexbuf; comment lexbuf }
```

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
}
```

4 Grammaires et analyse syntaxique

L'analyse syntaxique consiste à aller des mots aux phrases : il s'agit de regrouper les lexèmes produits par l'analyse lexicale pour reconstruire la structure du programme.



4.1 Le problème à l'envers : affichage

Avant de décrire l'analyse syntaxique elle-même, c'est-à-dire le passage d'une séquence de lexèmes à un arbre de syntaxe, nous allons regarder le problème inverse : partant d'un arbre de syntaxe, nous allons chercher à afficher joliment son contenu.

Prenons un noyaux simpliste d'expressions arithmétiques, avec constantes entières, additions et multiplications.

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

On peut ainsi considérer la valeur caml `e`

```
let e = Add(Cst 1, Mul(Add(Cst 2, Cst 3), Cst 4))
```

représentant l'expression `1+(2+3)*4`.

Résumons les règles d'écriture des expressions arithmétiques basées sur ces opérateurs :

- une constante entière est une expression arithmétique,
- la concaténation e_1 `+` e_2 d'une première expression arithmétique e_1, du symbole `+` et d'une deuxième expression arithmétique e_2 forme une expression arithmétique,
- la concaténation e_1 `*` e_2 d'une première expression arithmétique e_1, du symbole `*` et d'une deuxième expression arithmétique e_2 forme une expression arithmétique,
- la concaténation `(` e `)` d'une parenthèse ouvrante `(`, d'une expression arithmétique e et d'une parenthèse fermante `)` forme une expression arithmétique.

En notant E l'ensemble des expressions arithmétiques, on peut résumer ces règles avec la notation abrégée suivante :

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

Partant d'une valeur caml telle que `e` on veut donc afficher l'expression correspondante, ou disons renvoyer une chaîne de caractères représentant cette expression, en respectant les règles d'écriture que nous venons d'énoncer.

Une version naïve d'une telle fonction pourrait s'écrire ainsi.

```
let rec pp = function
  | Cst n -> string_of_int n
  | Add(e1, e2) -> pp e1 ^ " + " ^ pp e2
  | Mul(e1, e2) -> pp e1 ^ " * " ^ pp e2
```

Chaque opération binaire est directement traduite en la combinaison de ses deux opérandes par l'opérateur correspondant. Or, une telle fonction ne produit pas le résultat escompté.

```
# pp e
- : string = "1 + 2 + 3 * 4"
```

Il manque des parenthèses pour que cette chaîne décrive bien la structure de l'expression d'origine.

Modifier notre fonction pour ajouter des parenthèses partout n'est guère satisfaisant non plus, puisque cela écrit des parenthèses superflues, que l'on ne souhaite pas voir dans un affichage « joli ».

```
let rec pp = function
  | Cst n -> string_of_int n
  | Add(e1, e2) -> "(" + pp e1 ^ " + " ^ pp e2 + ")"
  | Mul(e1, e2) -> "(" + pp e1 ^ " * " ^ pp e2 + ")"
```

```
# pp e
- : string = "(1 + ((2 + 3) * 4))"
```

En décidant de ne jamais mettre de parenthèses autour des multiplications, ces parenthèses superflues peuvent être limitées, mais pas totalement éliminées.

```
let rec pp = function
  | Cst n -> string_of_int n
  | Add(e1, e2) -> "(" + pp e1 ^ " + " ^ pp e2 + ")"
  | Mul(e1, e2) -> pp e1 ^ " * " ^ pp e2
```

```
# pp e
- : string = "(1 + (2 + 3) * 4)"
```

Il faut donc encore regarder d'un peu plus près la structure de nos expressions arithmétiques.

On peut améliorer notre affichage en distinguant plusieurs catégories de positions à l'intérieur d'une expression arithmétique, qui doivent être traitées différemment par l'afficheur. En l'occurrence, l'écriture des opérandes d'une multiplication va parfois nécessiter des parenthèses qui auraient été inutiles pour la même expression placée à un autre endroit. Pour rendre compte de cela on peut séparer notre grammaire en deux catégories : E pour les expressions ordinaires et M pour les opérandes d'une multiplication. La différence entre les deux est qu'une opération d'addition doit être entourée de parenthèses lorsqu'elle est l'opérande d'une opération de multiplication.

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

On peut maintenant créer une nouvelle fonction d'affichage, plus fine, donnée en réalité par deux fonctions mutuellement récursives, chacune correspondant à l'une des deux positions E ou M que nous venons d'identifier.

```
let rec pp = function
  | Cst n -> string_of_int n
  | Add(e1, e2) -> pp e1 ^ " + " ^ pp e2
  | Mul(e1, e2) -> pp_m e1 ^ " * " ^ pp_m e2
and pp_m = function
  | Cst n -> string_of_int n
  | Add(e1, e2) -> "(" + pp e1 ^ " + " ^ pp e2 + ")"
  | Mul(e1, e2) -> pp_m e1 ^ " * " ^ pp_m e2
```

On obtient cette fois un affichage de notre expression exemple qui fait apparaître exactement les parenthèses nécessaires, et pas une de plus.

```
# pp e
- : string = "1 + (2 + 3) * 4"
```

On peut cependant améliorer encore une fois notre analyse et la fonction d'affichage qui s'en déduit : pour l'instant, les mêmes motifs apparaissent plusieurs fois dans notre description, et donc également plusieurs fois dans le code. On peut factoriser la description des expressions arithmétiques de manière à ce que les éléments n, E `+` E et M `*` M n'apparaissent plus qu'une fois chacun. Pour cela, on coupe maintenant en trois catégories : les expressions ordinaires E, les expressions multiplicatives M et les expressions atomiques A.

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

La fonction d'affichage peut encore être adaptée à cette nouvelle interprétation de la structure des expressions arithmétiques, pour donner un code équivalent au précédent mais sans plus aucune redondance.

```
let rec pp = function
  | Add(e1, e2) -> pp e1 ^ " + " ^ pp e2
  | e -> pp_m e
and pp_m = function
  | Mul(e1, e2) -> pp_m e1 ^ " * " ^ pp_m e2
  | e -> pp_a e
and pp_a = function
  | Cst n -> string_of_int n
  | e -> "(" ^ pp e ^ ")"
```

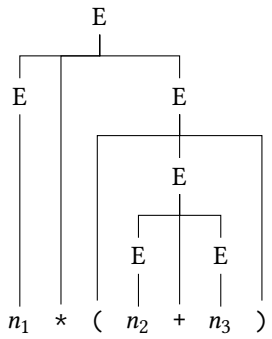
4.2 Grammaires et dérivations

La ***grammaire*** d'un langage décrit les structures de phrases légales dans ce langage. On la définit à l'aide de deux éléments en plus du lexique :

- les différentes catégories de (fragments de) phrases du langage,
- les manières dont différents éléments peuvent être regroupés pour former un fragment de l'une ou l'autre des catégories (données par des règles de combinaisons).

Les phrases bien formées sont alors celles dont les mots peuvent être regroupés d'une manière compatible avec les règles de combinaison.

Structure grammaticale d'une phrase On peut exposer la structure grammaticale d'une phrase en dessinant un arbre dont les feuilles sont les mots de la phrase (dans l'ordre), et dont chaque nœud interne dénote le regroupement de ses fils. Avec la première grammaire décrite pour les expressions arithmétiques nous pouvons construire ainsi la structure de la phrase n_1`*(`n_2`+`n_3`)`.



Grammaires et dérivations Formellement, on définit une ***grammaire algébrique*** par un quadruplet (T, N, S, R) où :

- T est un ensemble de symboles appelés ***terminaux***, désignant des mots (il s'agit des *lexèmes* produits par l'analyse lexicale),
- N est un ensemble de symboles appelés ***non terminaux***, désignant des groupes de mots (il s'agit des *catégories* évoquées plus haut),

— $S \in N$ est le ***symbole de départ***, qui décrit une phrase complète,
— $R \subseteq N \times (N \cup T)^*$ est un ensemble de ***règles de production***, associant des symboles non terminaux à des séquences de symboles (terminaux ou non).

Dans une grammaire naïve des expressions arithmétiques, on peut prendre :
— les symboles terminaux `+`, `*`, `(`, `)` et, pour chaque constante entière, `n`,
— le symbole non terminal E,
— le symbole de départ E,
— les règles de production $(E, \texttt{n})$, $(E, E\texttt{+}E)$, $(E, E\texttt{*}E)$ et $(E, \texttt{(}E\texttt{)})$, encore écrites

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

Étant donnée une grammaire (T, N, S, R), un ***arbre de dérivation*** est un arbre vérifiant les conditions suivantes :
— chaque nœud interne est étiqueté par un symbole de N,
— chaque feuille est étiquetée par un symbole de T ou de N, ou par le symbole ε,
— pour chaque nœud interne étiqueté par un symbole $X \in N$, les étiquettes de ses fils prises dans l'ordre forment une séquence β telle que $(X, \beta) \in R$.

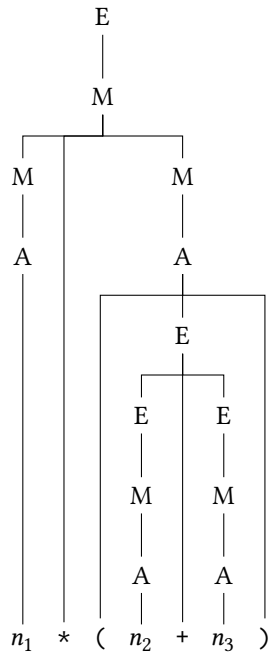
On dit qu'une phrase m est ***dérivable*** à partir d'un symbole non terminal X s'il existe un arbre de dérivation dont la racine est étiquetée par X et dont les feuilles, prises dans l'ordre, forment cette phrase. Les phrase ***dérivables*** de la grammaire sont celles qui sont dérivables à partir du symbole de départ S.

L'arbre vu ci-dessus était un arbre de dérivation pour l'expression arithmétique $n_1\texttt{*(}n_2\texttt{+}n_3\texttt{)}$ suivant la grammaire naïve, à partir de son symbole de départ E. La description plus fine des expressions arithmétiques que nous avons vue en fin de section précédente donne elle une grammaire distinguant trois catégories de fragments notées E, M et A et précisant les règles suivantes :
— le regroupement d'un fragment E, du mot `+` et d'un deuxième fragment E forme un fragment E,
— un fragment M seul forme également un fragment E,
— le regroupement d'un fragment M, du mot `*` et d'un deuxième fragment M forme un fragment M,
— un fragment A seul forme également un fragment M,
— un mot n seul forme un fragment A,
— le regroupement du mot `(`, d'un fragment E et du mot `)` forme un fragment A.

Autrement dit, nous avons une nouvelle grammaire définie par :
— les mêmes symboles terminaux `+`, `*`, `(`, `)` et, pour chaque constante entière, `n`,
— les symboles non terminaux E, M et A,
— le symbole de départ E,
— les règles de production $(E, E\texttt{+}E)$, (E, M), $(M, M\texttt{*}M)$, (M, A), $(A, \texttt{n})$ et $(A, \texttt{(}E\texttt{)})$, encore écrites

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

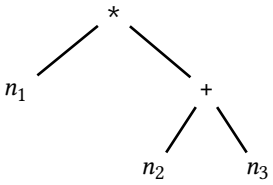
Nous avons alors pour $n_1\texttt{*(}n_2\texttt{+}n_3\texttt{)}$ un nouvel arbre de dérivation à partir du symbole de départ E.



Note : les arbres de dérivation matérialisent la structure d'une phrase, exprimée en fonction des règles de la grammaire. Il s'agit d'un concept différent de celui d'arbre de syntaxe abstraite. En l'occurrence, l'arbre de syntaxe abstraite associé à l'expression `n1 * (n2 + n3)` s'écrirait en caml

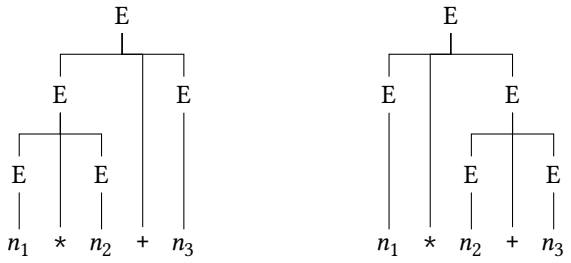
```
Mul(Cst n1, Add(Cst n2, Cst n3))
```

et serait dessiné ainsi, indépendamment de la grammaire utilisée pour caractériser la structure des expressions bien formées.



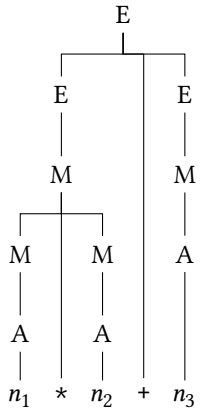
L'objectif d'un analyseur syntaxique est de construire cet arbre de syntaxe abstraite. L'arbre de dérivation, lui, décrit les différentes étapes de l'analyse réalisée, mais n'est pas construit explicitement par l'analyseur.

Ambiguïtés Une grammaire est ***ambiguë*** s'il existe une phrase qui peut être dérivée par deux arbres différents. Si l'on considère par exemple la phrase n_1*n_2+n_3 et qu'on l'analyse en suivant les règles de la grammaire naïve, deux arbres de dérivation sont possibles.



Le premier identifie comme un groupe la sous-expression n_1*n_2, et correspond bien à l'interprétation conventionnelle des expressions arithmétiques. Le deuxième en revanche forme un groupe avec le fragment n_2+n_3 et garde la multiplication à part. Cette deuxième interprétation est incorrecte par rapport aux conventions usuelles.

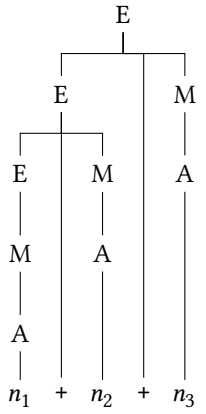
La deuxième grammaire ne présente pas d'ambiguïté sur cette expression, qui ne peut être dérivée que par l'arbre suivant.



On peut vérifier en revanche que cette grammaire plus élaborée reste ambiguë, par exemple avec la phrase n_1+n_2+n_3. On peut obtenir une grammaire non ambiguë avec (par exemple) la variante minime suivante.

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

On a alors notamment un unique arbre de dérivation pour la phrase n_1+n_2+n_3.



Présentation alternative des dérivations Alternativement, la dérivation d'une phrase selon une grammaire (T, N, S, R) donnée peut être caractérisée en utilisant, plutôt qu'un arbre, une suite de transformations de phrases partant du symbole de départ S et remplaçant progressivement des occurrences d'un symbole non terminal X par une séquence β telle que (X, β) est une règle dans R. Autrement dit, on *expanse* X. Dans ce cadre, on parle de ***grammaire générative*** et les règles sont appelées ***règles de production***.

Formellement, étant donnée une grammaire (T, N, S, R) et u, v deux mots sur $T \cup N$, on dit que u se ***dérive*** en v, et on note $u \to v$, si on a deux décompositions

$$
\begin{array}{rcl}
u & = & u_1 X u_2 \\
v & = & u_1 \beta u_2
\end{array}
$$

avec $X \in N$ et $(X, \beta) \in R$.

Avec notre exemple de grammaire arithmétique simple, E*(E) se dérive par exemple en E*(E+E), en prenant

$$
\begin{array}{rcl}
u_1 & = & E\texttt{*(} \\
u_2 & = & \texttt{)} \\
X & = & E \\
\beta & = & E\texttt{+}E
\end{array}
$$

De manière générale, on appelle ***dérivation*** une suite

$$m_1 \to m_2 \to \ldots \to m_n$$

de telles transformations, qu'on note encore

$$m_1 \to^* m_n$$

Par exemple :

$$\begin{aligned} E &\to E * E \\ &\to \mathtt{n}_1 * E \\ &\to \mathtt{n}_1 * (E) \\ &\to \mathtt{n}_1 * (E + E) \\ &\to \mathtt{n}_1 * (\mathtt{n}_2 + E) \\ &\to \mathtt{n}_1 * (\mathtt{n}_2 + \mathtt{n}_3) \end{aligned}$$

On dit qu'une séquence m est ***dérivable*** lorsqu'il existe une dérivation $S \to^* m$. On parle de ***dérivation gauche*** lorsque, comme ici, on expanse à chaque étape le symbole non terminal le plus à gauche. Remarquez que, lorsque l'on arrive à une séquence ne contenant plus que des symboles terminaux, il n'y a plus de dérivation possible.

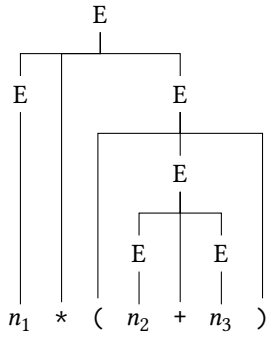
Équivalence entre les deux notions de dérivabilité Les deux notions de dérivabilité sont équivalentes. Que l'on considère des arbres de dérivation ou des suites de transformations, les mêmes phrases sont dérivables à partir des mêmes symboles.

La nuance entre les deux formalismes se trouve dans les différentes manières de dériver une même phrase : une séquence de transformations est associée à un ordre dans lequel expanser les différents symboles non terminaux, alors qu'un arbre de dérivation ne retient qu'une vision structurée de la manière dont chaque symbole non terminal est expansé, sans spécifier d'ordre dans lequel les règles sont appliquées. Autrement dit, un arbre de dérivation décrit potentiellement plusieurs séquences de dérivation différentes, qui peuvent varier quant à l'ordre dans lequel sont appliquées les différentes règles. En revanche, toutes les séquences de dérivation associées à un arbre donné conservent la même structure, et les mêmes associations entre symboles non terminaux et groupes de symboles de la séquence produite.

Ci-dessous, les deux séquences de transformation correspondent au même arbre de dérivation.

$$\begin{aligned} E &\to E * E \\ &\to \mathtt{n}_1 * E \\ &\to \mathtt{n}_1 * (E) \\ &\to \mathtt{n}_1 * (E + E) \\ &\to \mathtt{n}_1 * (\mathtt{n}_2 + E) \\ &\to \mathtt{n}_1 * (\mathtt{n}_2 + \mathtt{n}_3) \end{aligned}$$

$$\begin{aligned} E &\to E * E \\ &\to E * (E) \\ &\to E * (E * E) \\ &\to E * (E * \mathtt{n}_3) \\ &\to E * (\mathtt{n}_2 * \mathtt{n}_3) \\ &\to \mathtt{n}_1 * (\mathtt{n}_2 * \mathtt{n}_3) \end{aligned}$$



Montrons que pour tout arbre de dérivation d'une phrase m à partir d'un symbole a, on peut construire une séquence de dérivation de m à partir de a. On procède par récurrence sur la structure de l'arbre.

- Cas d'un arbre réduit à un unique nœud étiqueté par le symbole a, où la phrase m est a : on a la séquence de dérivation vide (ce cas couvre notamment la situation où a est un symbole terminal).
- Cas d'un arbre formé d'une racine X, dont les sous-arbres dérivent des mots m_1 à m_k à partir respectivement des symboles a_1 à a_k, où $(X, a_1 \ldots a_k)$ est une règle de la grammaire. Par hypothèse d'induction il existe des séquences de dérivation de a_i à m_i

pour tout $i \in [1; k]$. On peut alors par exemple construire la séquence de dérivation gauche suivante

$$X \to a_1 a_2 \dots a_k \to^* m_1 a_2 \dots a_k \to^* m_1 m_2 \dots a_k \to^* \dots \to^* m_1 m_2 \dots m_k$$

pour aller de X à $m_1 \dots m_k$.

Montrons à l'inverse que pour toute séquence de dérivation $a \to^* m$ d'une phrase m à partir d'un symbole a, il existe un arbre de dérivation de m à partir de a. On procède par récurrence sur la longueur de la séquence de dérivation.

- Cas d'une séquence de longueur zéro, où $a = m$, on conclut avec l'arbre comportant un unique nœud étiqueté par a.
- Cas d'une séquence $a \to m_1 \to \dots \to m_k \to m_{k+1}$ de longueur $k + 1$. Par hypothèse de récurrence il existe A un arbre de dérivation de m_k à partir de a. La dérivation $m_k \to m_{k+1}$ est faite avec des décompositions $m_k = u_1 X u_2$ et $m_{k+1} = u_1 \beta u_2$ pour une règle (X, β). Autrement dit, en notant l le nombre de symbole dans u_1, la $l + 1$-ème feuille de A porte le symbole X. On crée un arbre de dérivation pour m_{k+1} en donnant à cette feuille des fils, qui sont de nouvelles feuilles étiquetées par les symboles de la séquence β.

4.3 Analyse récursive descendante

L'analyse syntaxique prend en entrée une séquence de lexèmes, et doit vérifier que cette séquence est cohérente avec les règles de grammaire du langage. Pour chaque entrée à laquelle on applique un analyseur syntaxique, on attend l'une des issues suivantes :

- en cas de succès, un arbre de syntaxe abstraite donnant la structure de la séquence lue en entrée,
- en cas d'échec, la localisation et l'explication de ce qui est grammaticalement incohérent dans l'entrée.

Dans cette section, nous ne construirons pas encore l'arbre de syntaxe abstraite et nous contenterons d'indiquer si l'analyse a réussi (la phrase a une structure correcte) ou a échoué (la phrase a une structure incorrecte).

Comme on l'avait fait pour l'analyse lexicale, on va éviter les algorithmes coûteux consistant à regarder toutes les décompositions possibles d'un phrase et préférer une lecture de gauche à droite, sans retour en arrière.

La technique la plus simple à programmer revient à construire un arbre de dérivation en partant de la racine, c'est-à-dire du symbole de départ, et en expansant des symboles terminaux en essayant de former la phrase cible. On parle d'analyse *descendante*, ou *top-down*, puisque l'on découvre l'arbre de dérivation de haut en bas.

On réalise simplement un tel analyseur en définissant, pour chaque symbole non terminal X, une fonction f_X prenant en entrée une séquence de lexèmes, essayant de reconnaître dans un préfixe de cette séquence un fragment de phrase correspondant à la structure X, et renvoyant les lexèmes qui n'ont pas été utilisés (ou échouant si elle ne parvient pas à reconnaître un fragment correspondant à X).

Construction manuelle d'un analyseur Prenons notre grammaire non ambiguë des expressions arithmétiques.

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

Définissons ensuite un type de données pour les lexèmes,

```
type token =
  | Int of int
  | Plus
  | Mult
  | ParO
  | ParF
  | EOF
```

et voyons à quoi pourrait ressembler une fonction d'analyse pour le symbole non terminal A. Cette fonction prend en paramètre une liste de lexèmes à analyser, et renvoie la liste des lexèmes qui n'ont pas servi.

```
let parse_a: token list -> token list = function
```

La grammaire donne plusieurs règles d'expansion pour A. Pour éviter une explosion de l'analyse, on ne s'autorisera pas une approche par essais et erreurs (essayer d'abord avec une règle, puis avec l'autre si cela n'a pas fonctionné, et ainsi de suite). Il faut donc choisir, en fonction du contexte, la règle qui aura les meilleures chances de mener à une analyse réussie. Les informations à notre disposition pour choisir sont :

— notre connaissance de la grammaire,
— le prochain lexème de l'entrée.

En l'occurrence ici, le premier lexème de l'entrée permet sans ambiguïté de sélectionner une unique règle.

— Si le prochain lexème est une constante n, alors l'analyse va réussir avec la seule application de la règle $A \prec n$ et il suffit de renvoyer la suite de la liste.

```
    | Int _ :: l -> l
```

— Si le prochain lexème est une parenthèse ouvrante `(`, alors l'analyse *peut* réussir avec la règle $A \prec (E)$, si la suite de la liste correspond bien à la séquence restante $E)$. Il faut donc d'abord chercher à reconnaître E dans la suite de la liste, ce que l'on fait avec un appel à une fonction `parse_e` dédiée à la reconnaissance de cette structure (toutes les fonctions de reconnaissance vont donc être mutuellement récursives), puis vérifier que la séquence continue avec une parenthèse fermante.

```
    | ParO :: l ->
      let l' = parse_e l in
      let l'' = match l' with
        | ParF :: l'' -> l''
        | _ -> failwith "syntax error"
      in l''
```

— Dans tous les autres cas, aucune règle ne peut s'appliquer : la liste de lexèmes ne contient pas de préfixe correspondant à la structure A, et l'analyse doit échouer.

```
    | _ -> failwith "syntax error"
```

Ne reste donc pour compléter l'analyseur qu'à définir des fonctions analogues pour les autres symboles non terminaux. Le cas du symbole E n'est cependant pas si simple : nous avons deux expansions possibles $E \prec E$+M et $E \prec M$, entre lesquelles on ne peut choisir à coup sûr en ne connaissant que le prochain symbole de l'entrée (ni même en s'autorisant à regarder un certain nombre $k \geqslant 1$ de lexèmes, puisque le symbole + distinguant la première expansion peut apparaître arbitrairement loin).

Pour compléter notre analyseur, il va falloir modifier la grammaire de sorte à simplifier le choix entre les différentes règles à expanser. On peut par exemple modifier les règles du symbole E en les suivantes

$$\begin{array}{rcl} E & ::= & M \texttt{ + } E \\ & | & M \end{array}$$

de sorte à faire ressortir un préfixe commun aux deux règles : pour reconnaître un fragment E on commence dans tous les cas par chercher à reconnaître un fragment M, puis on peut ou non avoir une suite introduite par +.

```
and parse_e l =
  let l' = parse_m l in
  let l'' = match l' with
    | Plus :: l'' -> parse_e l''
    | _ -> l'
  in l''
```

On peut appliquer une modification similaire aux règles du symbole non terminal M pour écrire la fonction `parse_m` et compléter l'analyseur. La fonction principale de reconnaissance d'une expression complète consiste à alors à appliquer `parse_e` à la liste de lexèmes complète

et vérifier que l'on est allé jusqu'au bout de la séquence, c'est-à-dire qu'il ne reste plus que le symbole `EOF` de fin d'entrée.

```
let recognize l =
  parse_e l = [EOF]
```

Notez que les fonctions qui viennent d'être décrites peuvent être factorisées pour obtenir le code compact et complet suivant.

```
let expect t = function
  | t' :: l when t = t' -> l
  | _ -> failwith "syntax error"

let rec parse_e l = parse_m l |> parse_e'
and parse_e' = function
  | Plus :: l -> parse_e l
  | l -> l
and parse_m l = parse_a l |> parse_m'
and parse_m' = function
  | Mult :: l -> parse_m' l
  | l -> l
and parse_a = function
  | ParO :: l -> parse_e l |> expect ParF
  | Int _ :: l -> l
  | _ -> failwith "syntax error"

let recognize l = parse_e l = [EOF]
```

Cette factorisation du code est d'ailleurs liée à la factorisation possible suivante de la grammaire.

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

Enfin, on peut imaginer une variante de ce programme d'analyse qui construit au passage l'arbre de syntaxe abstraite de l'expression reconnue. Dans la version proposée ici on utilise une référence mutable sur la liste des lexèmes restant à prendre en compte. Les fonctions d'analyse modifient cette référence en retirant les lexèmes qu'elles utilisent et renvoient l'expression reconnue.

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

let parse l =
  let tokens = ref l in
  let next_token () = List.hd !tokens in
  let forward () = tokens := List.tl !tokens in
  let expect t =
    if next_token () = t then forward ()
    else failwith "syntax error"
  in

  let rec parse_e () =
    let e1 = parse_m () in
    parse_e' e1
  and parse_e' e1 = match next_token () with
    | Plus -> forward(); let e2 = parse_m () in parse_e' (Add(e1, e2))
```

```
    | _ -> e1
and parse_m () =
  let e1 = parse_a () in
  parse_m' e1
and parse_m' e1 = match next_token () with
  | Mult -> forward(); let e2 = parse_a () in parse_m' (Mul(e1, e2))
  | _ -> e1
and parse_a () = match next_token () with
  | Int n -> forward(); Cst n
  | ParO -> forward(); let e = parse_e () in expect ParF; e
  | _ -> failwith "syntax error"
in

let e = parse_e () in
if next_token () = EOF then e
else failwith "syntax error"
```

Note : si vous regardez de près les fonctions `parse_*` *de ce dernier programme, vous pourrez observer qu'elles apportent encore une légère variation par rapport à la dernière version de la grammaire des expressions arithmétiques. Quelle est cette nouvelle grammaire ? Quel programme aurait-on obtenu en suivant directement la version précédente de la grammaire ? En quoi diffèrent-ils ?*

Technique de construction d'un analyseur descendant La clé de la construction d'un analyseur récursif descendant est l'identification des règles d'expansion à choisir en fonction du prochain lexème de l'entrée. Pour cela on analyse la grammaire afin de trouver, pour chaque membre droit β de règle $X \prec \beta$, les symboles terminaux susceptibles de se trouver en tête d'une séquence dérivée de β. Pour obtenir un analyseur déterministe, on espère ne pas avoir plusieurs règles pour un même symbole X susceptibles de dériver des séquences commençant par un même symbole terminal.

Considérons une grammaire (T, N, S, R), et une séquence $\alpha \in (T \cup N)^*$ de symboles terminaux ou non terminaux. On note $\mathsf{Premiers}(\alpha)$ l'ensemble des symboles terminaux qui peuvent se trouver en tête d'une séquence dérivée à partir de α.

$$\mathsf{Premiers}(\alpha) = \{ a \in T \mid \exists \beta, \alpha \to^* a\beta \}$$

Si α a déjà la forme $a\beta$, avec $a \in T$ un symbole terminal, alors toute séquence dérivée aura encore cette forme, et a est l'unique symbole pouvant se trouver en tête. La situation est plus subtile lorsque α a la forme $X\beta$, avec $X \in N$ un symbole non terminal :

— d'une part, tout premier symbole de X est un premier possible de $X\beta$,
— d'autre part, *dans le cas où il est possible de dériver ε à partir de X*, les premiers de β sont également susceptibles de se trouver en tête.

Pour une analyse complète de la notion de premiers, nous caractérisons donc également l'ensemble des annulables, c'est-à-dire des symboles non terminaux à partir desquels il est possible de dériver la séquence vide.

$$\mathsf{Annulables} = \{ X \in N \mid X \to^* \varepsilon \}$$

Caractérisation de l'ensemble Annulables. Le symbole X est annulable :

— s'il existe une règle $X \prec \varepsilon$, ou
— s'il existe une règle $X \prec X_1 \dots X_k$ où tous les X_i sont des symboles non terminaux qui sont également annulables.

Caractérisation de l'ensemble $\mathsf{Premiers}(\alpha)$, pour α une séquence de $(T \cup N)^*$. On a les équations suivantes :

$$\begin{aligned}
\mathsf{Premiers}(\varepsilon) &= \emptyset \\
\mathsf{Premiers}(a\beta) &= \{a\} \\
\mathsf{Premiers}(X) &= \textstyle\bigcup_{X \prec \beta} \mathsf{Premiers}(\beta) \\
\mathsf{Premiers}(X\beta) &= \begin{cases} \mathsf{Premiers}(X) & \text{si } X \notin \mathsf{Annulables} \\ \mathsf{Premiers}(X) \cup \mathsf{Premiers}(\beta) & \text{si } X \in \mathsf{Annulables} \end{cases}
\end{aligned}$$

Dans un cas comme dans l'autre, on a des équations récursives. Pour trouver les ensembles des annulables et des premiers, on cherche donc des points fixes à ces équations.

Théorème de point fixe de Tarski. Si on considère un ensemble ordonné A fini et doté d'un plus petit élément $\bot$, et une fonction $F : A \to A$ croissante, alors :

— la fonction F admet au moins un point fixe, c'est-à-dire qu'il existe un élément $x \in A$ tel que $F(x) = x$, et
— en itérant F à partir de $\bot$, c'est-à-dire en calculant $F(\bot)$, puis $F(F(\bot))$, puis $F(F(F(\bot)))$, etc, on finit nécessairement par trouver un point fixe de F (et même plus précisément : le plus petit des points fixes).

Calcul des annulables. On cherche un ensemble de symboles non terminaux. On considère donc l'ensemble des parties de N, ordonné par l'ordre d'inclusion. Le plus petit élément est $\emptyset$. La fonction croissante qui nous intéresse est $F : \mathscr{P}(N) \to \mathscr{P}(N)$ définie par

$$F(E) = \{ X \mid X \to \varepsilon \} \cup \{ X \mid X \to X_1 \dots X_k,\ X_i \in E \}$$

Avec notre exemple des expressions arithmétiques nous avons : $F(\emptyset) = \{ E', M' \}$ et $F(\{ E', M' \}) = \{ E', M' \}$. Le point fixe est donc l'ensemble $\{ E', M' \}$, et les deux symboles E' et M' sont donc les deux seuls symboles non terminaux annulables. Ce calcul est souvent présenté dans un tableau de booléens donnant les annulables trouvés à chaque itération (on s'arrête lorsque deux lignes consécutives sont identiques).

	E	E'	M	M'	A
0.	F	F	F	F	F
1.	F	V	F	V	F
2.	F	V	F	V	F

Calcul des premiers. On cherche, pour chaque symbole non terminal, un ensemble de symboles terminaux. On considère donc l'ensemble $\mathscr{P}(T) \times \dots \times \mathscr{P}(T)$ (avec une composante par symbole non terminal), ordonné par le produit de l'ordre inclusion. Son plus petit élément est $(\emptyset, \dots, \emptyset)$, et la fonction croissante utilisée est celle qui recalcule les informations pour chaque symbole non terminal en fonction des informations déjà connues.

On présente généralement ce calcul dans un tableau, où chaque colonne correspond à un symbole non terminal, et chaque ligne à une itération. Le calcul s'arrête lorsque deux lignes deviennent identiques.

	E	E'	M	M'	A
0.	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1.	$\emptyset$	$\{+\}$	$\emptyset$	$\{*\}$	$\{n, (\}$
2.	$\emptyset$	$\{+\}$	$\{n, (\}$	$\{*\}$	$\{n, (\}$
3.	$\{n, (\}$	$\{+\}$	$\{n, (\}$	$\{*\}$	$\{n, (\}$
4.	$\{n, (\}$	$\{+\}$	$\{n, (\}$	$\{*\}$	$\{n, (\}$

L'ensemble Premiers permet de caractériser une règle $X \prec \beta$ à appliquer lorsque le prochain symbole de l'entrée est le premier symbole de la séquence dérivée à partir de β.

Mais que faire alors des règles de la forme $X \prec \varepsilon$? Dans une telle règle il n'existe pas de symbole pouvant se trouver en tête. On l'applique lorsque le prochain lexème de l'entrée n'est pas le premier symbole de X, mais plutôt le premier symbole *suivant* X.

À nouveau en analysant la grammaire, on peut caractériser l'ensemble des ***suivants*** d'un symbole non terminal $X \in N$, c'est-à-dire les symboles terminaux qui peuvent apparaître immédiatement après X dans une phrase dérivée à partir du symbole de départ.

$$\mathsf{Suivants}(X) = \{ a \in T \mid \exists \alpha, \beta,\ S \to^* \alpha X a \beta \}$$

Dans la grammaire des expressions arithmétiques, le symbole + appartient aux suivants de M, puisque nous avons par exemple la dérivation

$$S \to M\, E' \to M + E$$

mais également aux suivants de A puisque nous avons également la dérivation

$$S \to M\, E' \to M + E \to A\, M' + E \to A\, \varepsilon + E = A + E$$

Pour calculer l'ensemble des suivants de X, on regarde toutes les apparitions de X dans un membre droit de règle. Pour chaque règle de la forme $Y \prec \alpha X \beta$, on inclut dans $\mathsf{Suivants}(X)$:
— les symboles terminaux susceptibles d'apparaître en tête d'une séquence dérivée à partir de β, c'est-à-dire $\mathsf{Premiers}(\beta)$,
— dans le cas où il est possible de dériver la séquence vide à partir de β, tous les suivants de Y peuvent également suivre X (note : cela couvre le cas où β *est* la séquence vide, mais aussi le cas où β est une séquence de symboles non terminaux annulables).

On a à nouveau des équations récursives, à résoudre par la même technique que précédemment.

Calcul des suivants. On cherche, pour chaque symbole non terminal, un ensemble de symboles terminaux. La mise en place est donc la même que pour le calcul des premiers. À la première itération, on voit apparaître les symboles suivants qui sont directement visibles dans une règle. Dans les itérations suivantes, les informations se propagent. On considère en outre que le symbole terminal spécial # de fin d'entrée (correspondant à `EOF`) appartient aux suivants du symbole de départ S.

Dans notre exemple des expressions arithmétiques :

$$\begin{aligned}
\text{Suivants}(E) &= \{\#,)\} \cup \text{Suivants}(E') \\
\text{Suivants}(E') &= \text{Suivants}(E) \\
\text{Suivants}(M) &= \{+\} \cup \text{Suivants}(E) \cup \text{Suivants}(M') \\
\text{Suivants}(M') &= \text{Suivants}(M) \\
\text{Suivants}(A) &= \{*\} \cup \text{Suivants}(M)
\end{aligned}$$

Notamment, des informations vont se propager de E à M et de M à A, car M peut apparaître à la fin d'une séquence dérivée de E (après annulation de E'), et car A peut apparaître à la fin d'une séquence dérivée de M (après annulation de M').

	E	E'	M	M'	A
0.	$\{\#\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1.	$\{\#,)\}$	$\{\#\}$	$\{\#, +\}$	$\emptyset$	$\{*\}$
2.	$\{\#,)\}$	$\{\#,)\}$	$\{\#, +,)\}$	$\{\#, +\}$	$\{\#, +, *\}$
3.	$\{\#,)\}$	$\{\#,)\}$	$\{\#, +,)\}$	$\{\#, +\}$	$\{\#, +, *,)\}$
4.	$\{\#,)\}$	$\{\#,)\}$	$\{\#, +,)\}$	$\{\#, +\}$	$\{\#, +, *,)\}$

Une fois calculés les premiers et les suivants, on peut construire une ***table d'analyse LL(1)*** indiquant, pour chaque paire d'un symbole non terminal X en cours d'analyse et d'un symbole terminal a, la règle à appliquer lorsque l'on tente de reconnaître un fragment X et que le prochain lexème de l'entrée est a.

Le principe de construction est le suivant :

- Pour chaque règle $X \prec \beta$ avec $\beta \neq \varepsilon$ et chaque symbole terminal $a \in \text{Premiers}(\beta)$ on indique la règle $X \prec \beta$ dans la case de la ligne X et de la colonne a.
- Pour chaque règle $X \prec \varepsilon$ et chaque symbole terminal $a \in \text{Suivants}(X)$ on indique la règle $X \prec \varepsilon$ dans la case de la ligne X et de la colonne a.

Si chaque case contient au plus une règle, on obtient une table d'analyse déterministe. La grammaire utilisée, qui est donc compatible avec cette technique d'analyse, est dite ***LL(1)*** (***L**eft-to-right scanning, **L**eftmost derivation, using **1** look-ahead symbol*).

C'est le cas par exemple pour la grammaire des expressions arithmétiques dont nous avons calculé les premiers et suivants, pour laquelle nous avons la table d'analyse LL(1) suivante.

	n	+	*	(	)	#
E	$M\,E'$			$M\,E'$		
E'		$+\;E$			ε	ε
M	$A\,M'$			$A\,M'$		
M'		ε	$*\;M$		ε	ε
A	n			$(\;E\;)$		

Si à l'inverse, une case de la table contient plusieurs règles, on dit que la table contient un ***conflit***, et que la grammaire *n'est pas* LL(1). Pour obtenir un analyseur dans ce cas, il faut revoir la grammaire ou utiliser une autre technique.

Notez qu'être ou non compatible avec l'analyse LL(1) est une propriété de la grammaire, et non du langage décrit par la grammaire. Par exemple, la dernière grammaire que nous avons utilisée pour les expressions arithmétiques est LL(1), mais aucune des précédentes ne l'était.

Bilan sur l'analyse récursive descendante. La technique vue à cette section est très facile à programmer, à condition que la grammaire ait une forme compatible. C'est le cas notamment lorsque chaque construction du langage est introduite par un mot-clé ou un symbole spécifique (par exemple : **`if`** ou **`while`**). Dans les autres situations cependant, transformer la grammaire de notre langage de manière à la rendre compatible avec l'analyse LL(1) peut

être compliqué, et le résultat obtenu peut être cryptique, et éloigné de la manière naturelle de décrire le langage.

D'autres techniques existent, avec des caractéristiques différentes.

4.4 Analyse ascendante avec automate et pile

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

On suit donc les principes suivants.

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

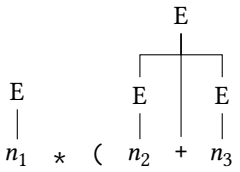
— Analyse de la phrase de bas en haut : les mots lus sont interprétés comme des arbres de dérivation triviaux, et sont regroupés à mesure que la lecture découvre des groupes entiers.

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

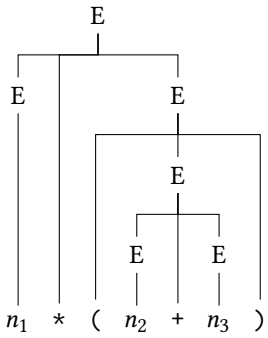
— Après lecture jusqu'à n_1 on identifie une première expression.

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

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



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



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

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

et deux opérations élémentaires, entre lesquelles on va alterner :

— ***progression*** : placer le prochain mot sur la pile, (aussi appelé ***décalage***, ou en anglais ***shift***)
— ***réduction*** : identifier au sommet de la pile une séquence β apparaissant dans une règle (X, β) de la grammaire, la retirer de la pile et la remplacer par X (en anglais ***reduce***).

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

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

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

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

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

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

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

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

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

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

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

sommet de la pile, il est inutile de progresser si le prochain symbole est + : on peut plutôt immédiatement interrompre l'analyse et signaler une erreur de syntaxe.

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

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

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

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

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

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

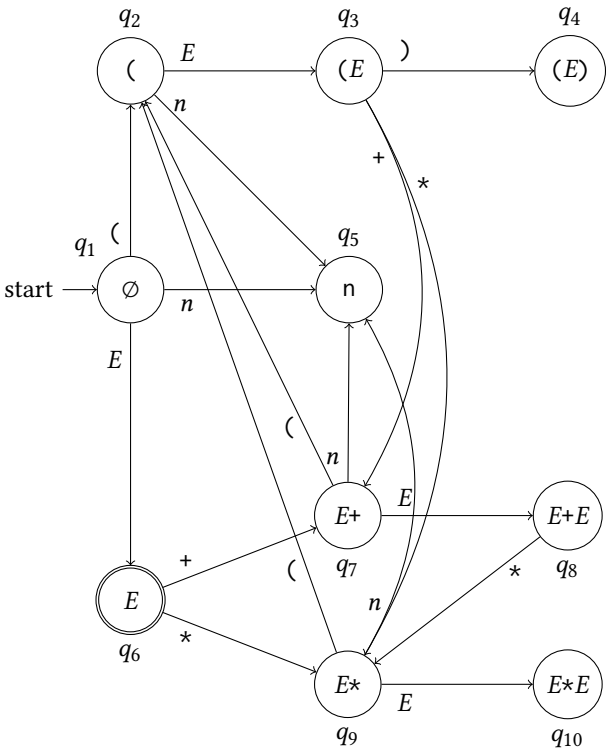
Sommet de pile	**Prochain mot**	**Action**
∅	*n* ou `(`	progression
	autres	échec
n	quelconque	réduction $E \prec n$
`(`	*n* ou `(`	progression
	autres	échec
`(` *E*	`)` ou `+` ou `*`	progression
	autres	échec
`(` *E* `)`	quelconque	réduction $E \prec (E)$
E	`+` ou `*`	progression
	fin de l'entrée	succès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

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

```
type token = Int of int | Plus | Mult | ParO | ParF | EOF
type expr = Cst of int | Add of expr * expr | Mul of expr * expr
type fragment =
  | Token of token
  | Expr of expr

let parse l =
  let rec parse s l = match s, l with
    | [], (Int _ | ParO as t) :: l ->
       parse [Token t] l
    | Token (Int n) :: s, _ ->
       parse (Expr (Cst n) :: s) l
    | Token ParO :: _, (Int _ | ParO as t) :: l ->
       parse (Token t :: s) l
    | Expr _ :: Token ParO :: _, (ParF | Plus | Mult as t) :: l ->
       parse (Token t :: s) l
    | Token ParF :: (Expr _ as e) :: Token ParO :: s, _ ->
       parse (e :: s) l
    | [Expr e], [EOF] ->
       e
```

```
    | [Expr _], (Plus | Mult as t) :: l ->
       parse (Token t :: s) l
    | Token (Mult | Plus) :: Expr _ :: _, (Int _ | Par0 as t) :: l ->
       parse (Token t :: s) l
    | Expr e2 :: Token Mult :: Expr e1 :: s, _ ->
       parse (Expr (Mul (e1, e2)) :: s) l
    | Expr _ :: Token Plus :: Expr _ :: _, Mult :: l ->
       parse s l
    | Expr e2 :: Token Plus :: Expr e1 :: s, _ ->
       parse (Expr (Add (e1, e2)) :: s) l
    | _ ->
       failwith "syntax error"
  in
  parse [] l
```

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



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

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

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

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

Dans les trois cas présentés dans ce tableau, nous étions *avant le début de l'analyse* du motif (E) dans l'un des états q_2, q_7 ou q_8 (motif (ou E + ou E *). Par exemple, dans la troisième ligne nous avons mis en suspens la reconnaissance d'une opération de multiplication le temps de compléter l'analyse de son deuxième opérande, une expression entre parenthèses

potentiellement complexe. L'état obtenu après réduction est l'un des états q_3, q_8 ou q_{10} (motif (E ou E + E ou E * E), chacun accessible par une transition E à partir de l'un des états de départ identifiés. Ce retour à un état précédent correspond, après reconstruction d'un fragment de phrase, à un retour au contexte d'analyse qui englobait ce fragment.

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

Nous construisons donc un automate dont :

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

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

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

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

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

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

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

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

Dans notre exemple.

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

4.5 Génération d'analyseurs syntaxiques avec menhir

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

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

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

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

L'outil menhir émet également un avertissement et un diagnostic en cas de conflit dans l'analyse. Ce point sera abordé dans les prochaines sections.

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

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

Pour un exemple minimaliste, on peut y inclure par exemple la définition d'un AST pour des expressions et des programmes. *Note : en conditions réelles, la syntaxe abstraite serait plutôt définie dans un module dédié et le prélude du fichier* `.mly` *ne ferait que l'importer).*

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

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

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

```
%token nom_du_lexme
```

pour les lexèmes ordinaires sans contenu et

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

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

```
(* Constantes entières *)
%token <int> CONST
(* Quelques symboles arithmétiques *)
%token PAR_O PAR_F PLUS FOIS
(* Fin de fichier *)
%token EOF
```

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

```
%start nom_du_symbole
```

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

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

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

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

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

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

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

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

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

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

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

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

```
%%
```

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

```
<nom_du_symbole>:
```

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

```
$numero_du_symbole
```

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

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

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

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

Menhir propose certaines facilités pour la manipulation des fragments mentionnés dans les règles. Plutôt que de faire référence aux éléments par un numéro, on peut associer des noms à certains éléments avec la notation

```
nom = symbole
```

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

```
expr:
| PAR_O e=expr PAR_F  { e }
```

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

```
| i=CONST  { Cst i }
```

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

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

%%
```

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

Menhir offre également quelques éléments spéciaux pour écrire des règles avec des éléments répétés ou optionnels, que nous aborderons plus tard.

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

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

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

Comme pour l'analyse LL, l'analyse ascendante réalisée par menhir est susceptible de buter sur des conflits. Nous allons voir bientôt que ces conflits sont souvent relativement faciles à régler. Avant cela toutefois, nous allons détailler la manière dont les outils comme menhir construisent leurs tables d'analyse.

4.6 Construction d'automates d'analyse ascendante

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

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

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

où

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

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

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

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

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

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

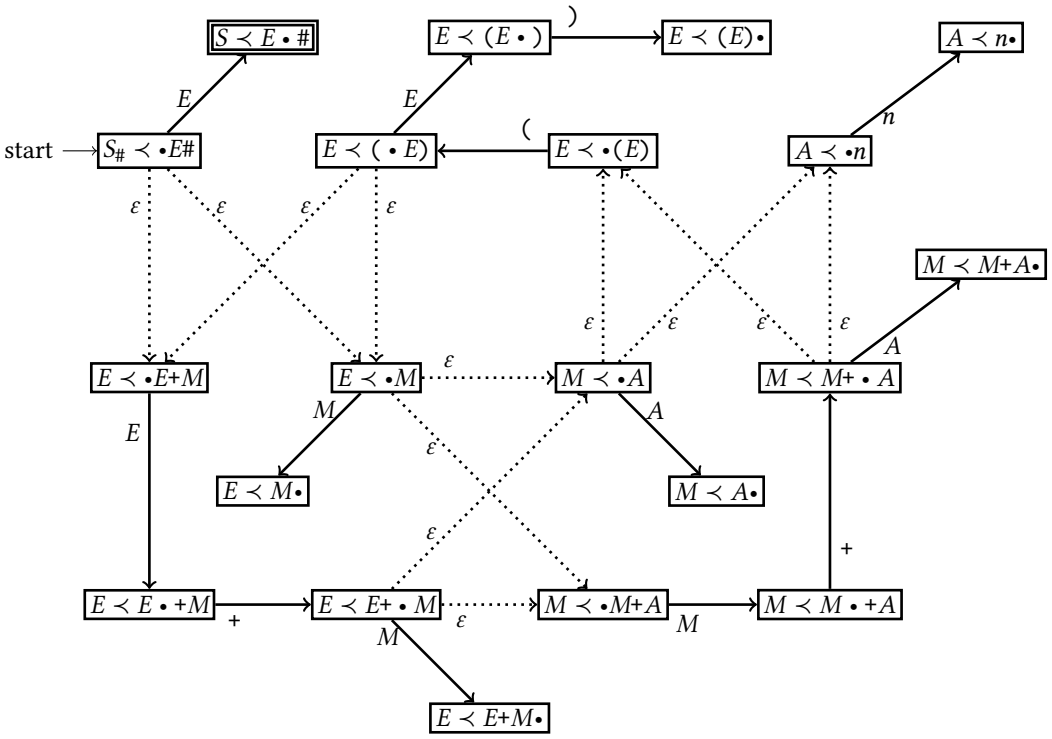
En prenant la grammaire suivante des expressions arithmétiques

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

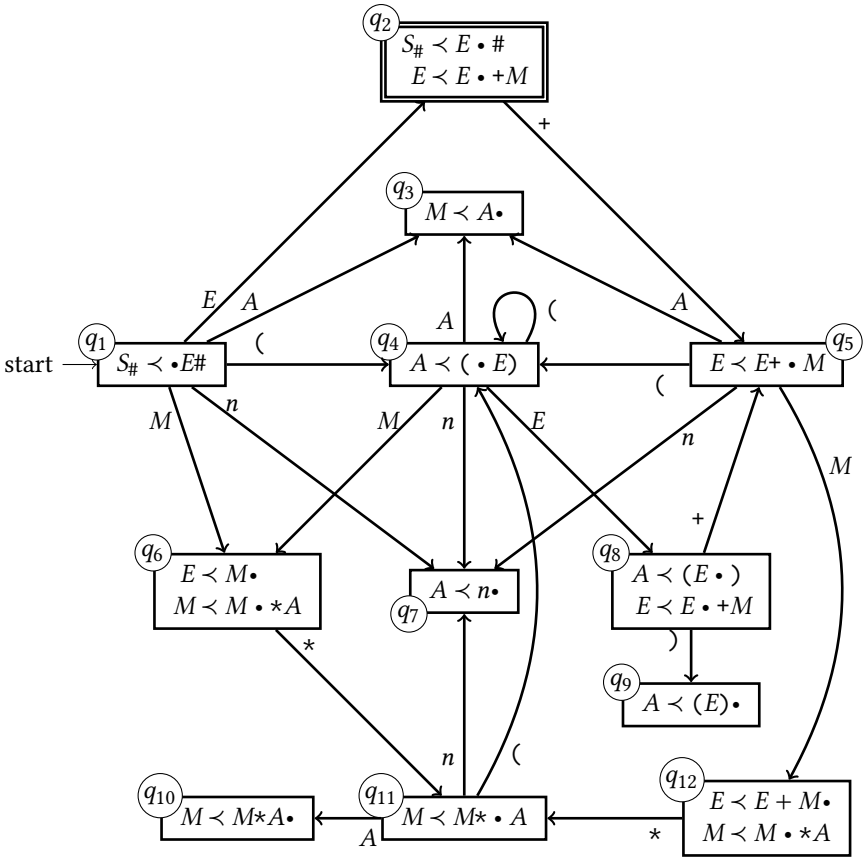
et en y ajoutant la règle de départ

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

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



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



Dans ce dessin, on a gardé une écrire compacte des unions d'états, en n'indiquant que les éléments qui ne peuvent pas être directement déduits par des transitions ε. Ainsi par

exemple, l'état q_{11} correspond à l'union d'états

$$M \prec M\star \bullet A$$
$$A \prec \bullet n$$
$$A \prec \bullet (E)$$

et l'état q_1 correspond à l'union d'états

$$S_\# \prec \bullet E\#$$
$$E \prec \bullet E+M$$
$$E \prec \bullet M$$
$$M \prec \bullet M\star A$$
$$M \prec \bullet A$$
$$A \prec \bullet n$$
$$A \prec \bullet (E)$$

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

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

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

Pour la table des sauts :

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

En appliquant à notre exemple on obtient les tables suivantes.

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

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

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

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

- les conflits entre progression et réduction (*shift/reduce*),
- les conflits entre plusieurs réductions (*reduce/reduce*).

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

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

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

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

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

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

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

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

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

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

$$M \ \prec \ M * A$$

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

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

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

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

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

— aucun symbole n'est annulable,
— le calcul des premiers se déroule ainsi :

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

— et le calcul des suivants donne

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

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

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

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

— Les états sont des quadruplets $[X \prec \alpha \bullet \beta, a]$ avec $X \in N, \alpha, \beta \in (T \cup N)^*$, $(X, \alpha\beta) \in R$ et $a \in \text{Suivants}(X)$. Un tel état s'interprète comme « nous sommes en train de chercher à reconnaître la séquence $\alpha\beta$ pour la regrouper en un fragment X qui devra être suivi du symbole a, nous avons déjà reconnu la partie α et il reste à reconnaître la partie β (et vérifier la présence de a ensuite) ».
— L'unique état initial est $[S_\# \prec \bullet S, \#]$.
— L'unique état acceptant est $[S_\# \prec S \bullet, \#]$.
— Les transitions sont toutes celles qui peuvent être formées de l'une des trois manières suivants :
  — $[Y \prec \alpha \bullet a\beta, b] \xrightarrow{a} [Y \prec \alpha a \bullet \beta, b]$ avec a symbole terminal,
  — $[Y \prec \alpha \bullet X\beta, b] \xrightarrow{a} [Y \prec \alpha X \bullet \beta, b]$ avec X symbole non terminal,
  — $[Y \prec \alpha \bullet X\beta, b] \xrightarrow{\varepsilon} [X \prec \bullet\gamma, c]$ avec (X, γ) une règle et $c \in \text{Premiers}(\beta b)$.

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

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

4.7 Conflits et priorités

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

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

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

Elle est a fortiori compatible avec l'analyse LR(1) faite par menhir.
— Une grammaire encore un peu plus tordue, mais cette fois compatible avec l'analyse

LL(1).

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

Elle a l'avantage de permettre l'écriture d'un analyseur récursif descendant.

— Une grammaire naïve et naturelle.

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

Cette grammaire est ambiguë et sera rejetée par tout outil d'analyse. Mais n'est-ce pas celle-ci que vous souhaiteriez utiliser en pratique ?

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

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

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

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

```
%token CONST PLUS PAR_O PAR_F EOF
%start prog
%type <unit> prog
%%

prog:
| expr EOF {}

expr:
| CONST            {}
| expr PLUS expr   {}
| PAR_O expr PAR_F {}
```

Le bilan donné par menhir est le suivant.

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

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

Dans notre exemple, on y trouve par exemple la description suivante d'un état.

```
State 4:
```

```
## Known stack suffix:
## expr PLUS
## LR(1) items:
expr -> expr PLUS . expr [ PLUS PAR_F EOF ]
## Transitions:
-- On PAR_O shift to state 1
-- On CONST shift to state 2
-- On expr shift to state 5
## Reductions:
```

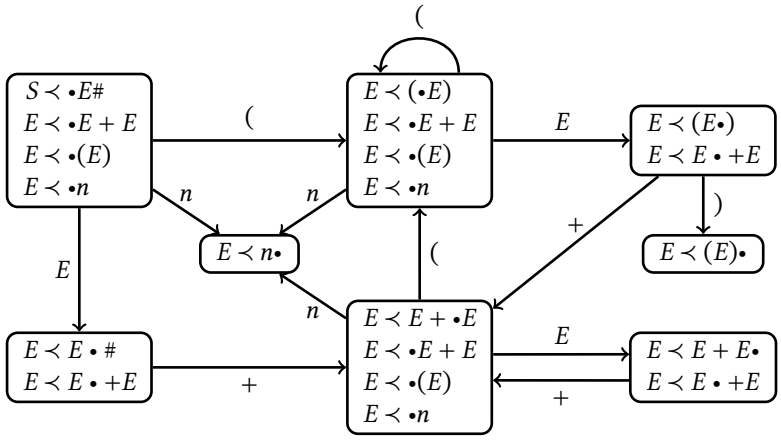
On y voit notamment un numéro pour l'état, un ensemble d'éléments nécessairement présents au sommet de la pile et une règle de réduction $E \prec E{+}E$ en cours de reconnaissance. Le point entre le symbole `PLUS` et la deuxième occurrence de `expr` correspond au niveau de progression dans la règle (c'est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

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

```
State 5:
## Known stack suffix:
## expr PLUS expr
## LR(1) items:
expr -> expr . PLUS expr [ PLUS PAR_F EOF ]
expr -> expr PLUS expr . [ PLUS PAR_F EOF ]
## Transitions:
-- On PLUS shift to state 4
## Reductions:
-- On PLUS PAR_F EOF
--   reduce production expr -> expr PLUS expr
** Conflict on PLUS
```

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

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



En plus de cette description de l'automate, menhir produit un fichier `.conflicts` donnant des détails sur les enjeux de chacun des conflits détectés.

Détaillons le contenu de ce fichier pour notre exemple. On a d'abord une identification de l'état, du prochain lexème de l'entrée pour lequel le conflit existe, et d'un état de la pile correspondant à cet état (ce dernier point peut être vu comme un chemin dans l'automate).

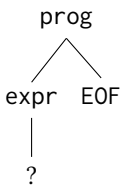
```
** Conflict (shift/reduce) in state 5.
** Token involved: PLUS
** This state is reached from prog after reading:
```

```
expr PLUS expr
```

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

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

prog
expr EOF
(?)
```

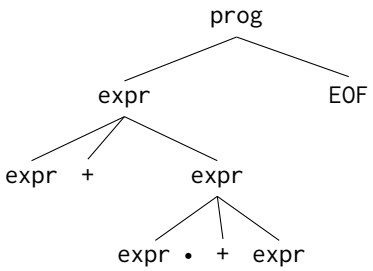
que nous pouvons traduire par le schéma



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

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

expr PLUS expr
          expr . PLUS expr
```

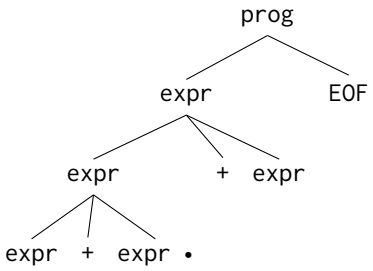
Dans ce cas le prochain symbole `PLUS` est interprété comme faisant partie de l'opérande droit de la première opération. Ceci correspondrait à un arbre de dérivation complété ainsi, où le point • désigne le stade de progression dans la lecture de l'entrée.



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

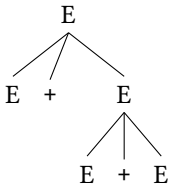
```
** In state 5, looking ahead at PLUS, reducing production
** expr -> expr PLUS expr
** is permitted because of the following sub-derivation:

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

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

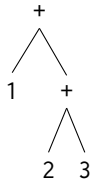


Bilan de ce rapport : à un tel stade de l'analyse de notre entrée, nous avons un choix entre progresser avec le symbole +, ou réduire avec la règle $E \prec E{+}E$.

— Progresser mène à un arbre de dérivation

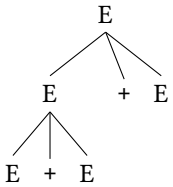


Sur l'entrée concrète `1+2+3` nous aurions l'arbre de syntaxe

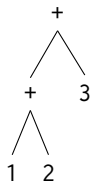


correspondant au parenthésage `1+(2+3)`.

— Réduire mène à un arbre de dérivation



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



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

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

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

```
%left PLUS
```

D'autres conflits plus variés vont intervenir lorsque nous aurons plusieurs symboles dans la grammaire. Ajoutons la multiplication `*`, représentée avec un symbole terminal `STAR` dans menhir.

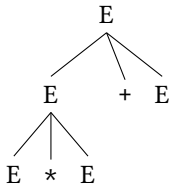
On obtient trois nouveaux conflits :

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

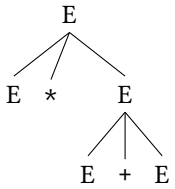
Le premier cas est similaire au précédent, et est relatif à l'associativité de la multiplication (on déclarera donc encore la multiplication comme associative à gauche). Les deux autres en revanche dépendent des priorités relatives données aux deux opérations d'addition et de multiplication.

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

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



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

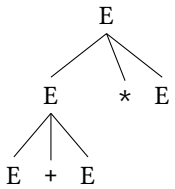


c'est-à-dire interpréter `2*3+4` comme `2*(3+4)`.

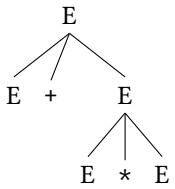
La solution cohérente avec les conventions mathématiques usuelles est la première : il faut donc dans ce cas favoriser la réduction.

À l'inverse, si on a reconnu `expr PLUS expr` est que le prochain symbole est `STAR` on peut :

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



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



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

La solution cohérente avec les conventions mathématiques usuelles est cette fois la seconde : il faut donc dans ce cas favoriser la progression.

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

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

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

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

Le symbole `-` par exemple peut apparaître dans deux situations :

— en tant qu'opérateur binaire, pour l'opération de soustraction, comme dans `1 - 2`,

— en tant qu'opérateur unaire, pour l'opération « opposé », comme dans `-1`.

Dans l'expression `-2 * 3 - 4 * 5` ce symbole apparaît ainsi deux fois, une fois dans chacune des deux situations. Le parenthésage implicite conventionnel est alors `((-2)*3) - (4*5)`. Autrement dit, le symbole - est plus prioritaire que la multiplication lorsqu'il représente une opération uniaire, et moins prioritaire que la multiplication lorsqu'il représente une opération binaire.

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

```
%token PLUS STAR MINUS U_MINUS

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

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

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

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

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

La résolution est alors séparée en deux cas :

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

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

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

4.8 Analyse syntaxique de FUN

Concluons ce chapitre en construisant un analyseur syntaxique complet pour le mini-langage fonctionnel FUN vu à la fin du chapitre 2. Ce langage peut être vu comme le sous-ensemble de caml contenant les éléments suivants :

— les nombres entiers,
— quelques opérateurs arithmétiques et logiques : `+`, `*`, `-`, `=`,
— les variables locales introduites par **`let`** `x = e1` **`in`** `e2`,
— l'expression conditionnelle **`if`** `e1` **`then`** `e2` **`else`** `e3`,
— les fonctions anonymes **`fun`** `x -> e`,
— les fonctions récursives introduites par **`let rec`** `f x1 ... xn = e1` **`in`** `e2`.

On organise ce programme en quatre parties :

— un module `Fun` (fichier `fun.ml`) définit l'AST et d'éventuelles fonctions auxiliaires de manipulation de la syntaxe abstraite,
— un module `Funparser` réalisé avec menhir (fichier `funparser.mly`) définit les lexèmes et l'analyseur syntaxique,

— un module `Funlexer` réalisé avec ocamllex (fichier `funlexer.mll`) définit l'analyseur lexical,
— un module principal `Func` (fichier `func.ml`) définit le programme principal, qui analyse un fichier fourni en entrée.

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

```
type binop = Add | Mul | Sub | Eq
type expr =
  | Cst    of int
  | Binop  of binop * expr * expr
  | Var    of string
  | Let    of string * expr * expr
  | If     of expr * expr * expr
  | Fun    of string * expr
  | App    of expr * expr
  | LetRec of string * expr * expr
```

Analyse syntaxique Définition de l'analyseur syntaxique avec menhir (fichier `funparser.mly`). On a un prélude minimal, qui se contente de charger le module de syntaxe abstraite, suivi immédiatement de la définition des lexèmes.

```
%{
  open Fun
%}

%token <int> CST
%token <string> IDENT
%token PLUS STAR MINUS EQUAL LPAR RPAR
%token FUN ARROW LET REC IN IF THEN ELSE
%token EOF
```

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

$$
\begin{array}{lcl}
P & ::= & E\,\texttt{\#} \\
E & ::= & n \\
 & | & E\,\texttt{+}\,E \\
 & | & E\,\texttt{-}\,E \\
 & | & E\,\texttt{*}\,E \\
 & | & E\,\texttt{=}\,E \\
 & | & \texttt{(}\,E\,\texttt{)} \\
 & | & x \\
 & | & \texttt{let}\ x\ \texttt{=}\ E\ \texttt{in}\ E \\
 & | & \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\
 & | & \texttt{fun}\ x\ \texttt{->}\ E \\
 & | & E\,E \\
 & | & \texttt{let rec?}\ x\,x^\star\ \texttt{=}\ E\ \texttt{in}\ E
\end{array}
$$

où n désigne une constante entière, x un identifiant de variable, `rec?` la présence optionnelle de `rec` et où $x^\star$ une succession d'un nombre quelconque d'identifiants. On souhaite en outre que les conventions de priorité de caml soient respectées.

On peut sans difficulté particulière déjà traduire le symbole de départ P en un symbole non terminal `prog` et définir la règle associée.

```
%start prog
%type <Fun.expr> prog
%%

prog:
| e=expr EOF { e }
```

Pour rapporter à l'utilisateur un minimum d'information en cas d'erreur de syntaxe dans le programme analysé, on ajoute une règle d'erreur, qui récupère la position à laquelle l'analyse

a échoué avec la valeur spéciale de menhir `$startpos`, et qui traduit cette position en un numéro de ligne et un numéro de colonne (voir le module `Lexing` de caml pour le format des positions).

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

Pour la grammaire des expressions, on apporte ensuite trois changements.

— Pour éviter les duplications de code, on factorise les quatre règles

$$\begin{array}{rcl} E & ::= & E + E \\ & | & E - E \\ & | & E * E \\ & | & E = E \end{array}$$

en une seule règle $E ::= E\,O\,E$ en faisant apparaître un nouveau symbole O désignant au sens large un opérateur binaire.

— Pour gérer correctement l'identification des arguments des fonctions, on isole à l'intérieur de E l'ensemble S des expression qui peuvent être reconnues comme des arguments, appelées « expressions simples ». Il s'agit des constantes, des identifiants et des expressions placées entre parenthèses.

— On supprime la règle de définition d'une variable locale

$$E ::= \texttt{let}\ x\ \texttt{=}\ E\ \texttt{in}\ E$$

qui est redondante avec la règle

$$E ::= \texttt{let rec?}\ x\ x^\star\ \texttt{=}\ E\ \texttt{in}\ E$$

de définition d'une fonction. Notez qu'il faudra quand même distinguer les deux situations, mais cela sera fait a posteriori.

On obtient ainsi la nouvelle grammaire

$$\begin{array}{rcl} E & ::= & S \\ & | & E\,O\,E \\ & | & \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\ & | & \texttt{fun}\ x\ \texttt{->}\ E \\ & | & E\,S \\ & | & \texttt{let rec?}\ x\ x^\star\ \texttt{=}\ E\ \texttt{in}\ E \\ S & ::= & n \\ & | & x \\ & | & \texttt{(}\ E\ \texttt{)} \\ O & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{=} \end{array}$$

Ne reste plus alors qu'à traduire cette nouvelle version en menhir. La plupart des règles se traduisent directement.

```
simple_expr:
| n=CST              { Cst n }
| x=IDENT            { Var x }
| LPAR e=expr RPAR   { e     }
;

expr:
| e=simple_expr                          { e                 }
| e1=expr op=binop e2=expr               { Binop(op, e1, e2) }
| IF c=expr THEN e1=expr ELSE e2=expr    { If(c, e1, e2)     }
| FUN x=IDENT ARROW e=expr               { Fun(x, e)         }
| e1=expr e2=simple_expr                 { App(e1, e2)       }
```

La règle du **let** fait intervenir des éléments optionnels et des éléments répétés. On peut utiliser pour cela les primitives `option` et `list` apportées par menhir.

```
  | LET r=option(REC) f=IDENT args=list(IDENT) EQUAL e1=expr IN e2=expr
```

La primitive `option` renvoie une option de caml, c'est-à-dire soit `None` soit un lexème. La primitive `list` renvoie de même une liste caml. On peut donc apporter le traitement suivant, où `mk_fun: string list -> expr -> expr` est une fonction auxiliaire à définir dans le module `Fun` qui permet d'interpréter **let rec** `f x y = e` de la même manière que **let rec** `f = fun x -> fun y -> e` (la première écriture est un ***sucre syntaxique***).

```
      { let fn = mk_fun args e1 in
        if r = None then Let(f, fn, e2) else LetRec(f, fn, e2) }
  ;
```

La fonction `mk_fun` est définie par les lignes suivantes (fichier `fun.ml`). Notez que si aucun argument n'est fourni le résultat est simplement l'expression `e`, et on ne crée donc pas de fonction.

```
  let rec mk_fun args e = match args with
    | []       -> e
    | x::args -> Fun(x, mk_fun args e)
```

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

```
  %inline binop:
  | PLUS  { Add }
  | MINUS { Sub }
  | STAR  { Mul }
  | EQUAL { Eq }
  ;
```

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

```
  | e1=expr op=binop e2=expr  { Binop(op, e1, e2)  }
```

par les quatre règles suivantes.

```
  | e1=expr PLUS  e2=expr     { Binop(Add, e1, e2) }
  | e1=expr MINUS e2=expr     { Binop(Sub, e1, e2) }
  | e1=expr STAR  e2=expr     { Binop(Mul, e1, e2) }
  | e1=expr EQUAL e2=expr     { Binop(Eq, e1, e2)  }
```

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

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

```
  %nonassoc IN ELSE ARROW
  %left EQUAL
  %left PLUS MINUS
  %left STAR
  %left LPAR IDENT CST
```

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

```
  {
    open Lexing
    open Funparser
```

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

```
  let keyword_or_ident =
    let h = Hashtbl.create 17 in
    List.iter
      (fun (s, k) -> Hashtbl.add h s k)
      [ "fun",  FUN;
        "let",  LET;
        "rec",  REC;
        "in",   IN;
        "if",   IF;
        "then", THEN;
        "else", ELSE;
      ] ;
    fun s ->
      try Hashtbl.find h s
      with Not_found -> IDENT(s)
}
```

Le reste de l'analyseur est ensuite conforme à ce que nous avons pu voir au chapitre précédent.

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

rule token = parse
  | ['\n']       { new_line lexbuf; token lexbuf }
  | [' ' '\t' '\r']+   { token lexbuf }
  | "(*"         { comment lexbuf; token lexbuf }
  | digit+ as n  { CST(int_of_string n) }
  | ident as id  { keyword_or_ident id }
  | '+'   { PLUS  }
  | '*'   { STAR  }
  | '-'   { MINUS }
  | '='   { EQUAL }
  | "->"  { ARROW }
  | '('   { LPAR  }
  | ')'   { RPAR  }
  | _ as c { failwith (Printf.sprintf "invalid character: %c" c) }
  | eof   { EOF   }

and comment = parse
  | "*)"  { () }
  | "(*"  { comment lexbuf; comment lexbuf }
  | '\n'  { new_line lexbuf; comment lexbuf }
  | _     { comment lexbuf }
  | eof   { failwith "unterminated comment" }
```

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

```
let () =
  let fichier = Sys.argv.(1) in
  let c = open_in fichier in
  let lexbuf = Lexing.from_channel c in
  let ast = Funparser.prog Funlexer.token lexbuf in
  close_in c;
  ignore(ast);
  exit 0
```

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

5 Sémantique et types

Formalisation de ce qui signifient un programme ou une donnée, de la manière dont il faut les interpréter, et de ce qu'on peut en attendre.

5.1 Valeurs et opérations typées

À l'intérieur de l'ordinateur, une donnée est une séquence de bits. Voici par exemple un mot mémoire de 32 bits.

```
1110 0000 0110 1100 0110 0111 0100 1000
```

Pour faciliter la lecture, on représente souvent un tel mot au format hexadécimal. En l'occurrence, on l'écrirait

```
0x e0 6c 67 48
```

(le `0x` au début indique simplement le format hexadécimal, puis chaque caractère correspond à un groupe de 4 bits).

Que peut signifier cette donnée ? Pour le savoir, il faut une connaissance *très* précise du contexte :

- si la donnée est une adresse mémoire, il s'agira de l'adresse `3 765 200 712`,
- si la donnée est un nombre entier signé 32 bits en complément à 2, ce nombre sera −529 766 584,
- si la donnée est un nombre flottant simple précision de la norme IEEE754, ce nombre sera $15\,492\,936 \times 2^{42}$,
- si la donnée est une chaîne de caractères au format Latin-1, il s'agira de `"Holà"`.

Si on oublie le contexte dans lequel une séquence de bits a du sens, on est susceptible de faire n'importe quoi. *Par exemple : appliquer une opération d'addition entière aux représentations des chaînes de caractères* `"5"` *et* `"37"` *produit la nouvelle chaîne de caractères* `"h7"`.

Opération incohérentes En réalité, toutes les opérations proposées par un langage de programmation sont contraintes.

- L'addition `5 + 37` entre deux entiers est possible en caml
- Les opérations `"5" + 37` ou `5 + (fun x -> 37)` ou `5(37)` ne le sont pas.

On a déjà pu observer ce point dans l'interprète du langage FUN vu à la fin du chapitre 2. L'ensemble des valeurs que pouvait produire une expression y était séparé en deux catégories : les nombres et les fonctions.

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

et on pouvait voir pour certaines opérations des comportements distincts en fonction de la catégorie de valeur manipulée. Une opération arithmétique binaire par exemple était censée s'appliquer à des nombres. Elle produisait un résultat (de sorte `VCst`) si ses deux opérandes avaient bien des valeurs de la sorte `VCst`, et échouait sinon en interrompant l'exécution du programme avec `assert false`.

```
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
```

Les types : classification des valeurs En général, un langage de programmation distingue de nombreuses classes différentes de valeurs, appelées ***types***. La classification dépend de chaque langage, mais on y retrouve souvent de nombreux éléments commun. On a d'une part des types de base du langage, pour les données simples. Par exemple :

- nombres : `int`, `double`,
- valeurs booléennes : `bool`,
- caractères : `char`,
- chaînes : `string`.

D'autre part, des types plus riches peuvent être construits à partir de ces types de base. Par exemple :

— tableaux : `int[]`,
— fonctions : `int -> bool`,
— structures de données : `struct point { int x; int y; };`,
— objets : `class Point { public final int x, y; ... }`.

Une fois cette classification établie, chaque opération va s'appliquer à des éléments d'un type donné.

Dans certains cas, un même opérateur peut s'appliquer à plusieurs types d'éléments, avec des significations différentes à chaque fois. On parle de ***surcharge***. En python et en java par exemple, l'opérateur + peut s'appliquer :

— à deux entiers, et désigne alors l'addition : `5 + 37 = 42`,
— à deux chaînes, et désigne alors la concaténation : `"5" + "37" = "537"`.

Les langages de programmation permettent également parfois le ***transtypage*** (*cast*), c'est-à-dire la conversion d'une valeur d'un type vers un autre. Cette conversion peut même être implicite. Ainsi l'opération `"5" + 37` mélangeant une chaîne et un entier aura comme résultat :

— `42` en php, où la chaîne `"5"` est convertie en le nombre `5`,
— `"537"` en java, où l'entier `37` est converti en la chaîne `"37"`.

Notez qu'une telle conversion peut demander une traduction de la donnée ! Le nombre `5` est représenté par le mot mémoire `0x 00 00 00 05`, mais la chaîne `"5"` par le mot mémoire `0x 00 00 00 35`. De même, le nombre `37` est représenté par le mot mémoire `0x 00 00 00 25`, mais la chaîne `"37"` par le mot mémoire `0x 00 00 37 33`. Dans un sens comme dans l'autre, une conversion d'un type à l'autre demande de calculer la nouvelle représentation.

Bilan. Le type d'une valeur donne une clé d'interprétation de cette donnée, et peut être utile à la sélection des bonnes opérations. En outre, une incohérence dans les types révèle un problème du programme, dont l'exécution doit donc être évitée.

Analyse statique des types Gérer les types des données au moment de l'exécution génère des coûts variés :

— de la mémoire pour accompagner chaque donnée d'une indication de son type,
— des tests pour sélectionner les bonnes opérations,
— des exécutions interrompues en cas de problème, ...

Dans des langages ***typés dynamiquement*** comme Python, ces coûts sont la norme. En revanche, les langages ***typés statiquement*** comme C, java ou caml nous épargnent tout ou partie de ces coûts à l'exécution en gérant autant que possible tout ce qui concerne les types dès la compilation.

Dans l'analyse ***statique*** des types, c'est-à-dire l'analyse des types *à la compilation*, on associe à chaque expression d'un programme un type, qui prédit le type de la valeur qui sera produite par cette expression. Cette prédiction est basée sur des contraintes associées à chaque élément de la syntaxe abstraite. Par exemple, en présence d'une expression d'addition de la forme `Add(e1, e2)` nous pouvons noter deux faits :

— l'expression produira un nombre,
— les deux sous-expressions `e1` et `e2` doivent impérativement produire des valeurs numériques, sans quoi l'ensemble serait mal formé.

On associe de même un type à une variable, pour désigner le type de la valeur référencée par cette variable. Ainsi, dans `let x = e in x + 1` le type de `x` est le type de la valeur produite par l'expression `e` (et on s'attend à ce qu'il s'agisse d'un nombre entier). Enfin, le type d'une fonction va mentionner ce que sont les types attendus pour chacun des paramètres, ainsi que le type du résultat renvoyé.

Le slogan associé à cette vérification de la cohérence des types avant l'exécution des programmes, du à Robin Milner, est

Well-typed programs do not go wrong.

L'objectif du typage statique est ainsi de rejeter les programmes absurdes avant même qu'ils soient exécutés (ou livrés à un client...). Dans l'absolu, on ne peut pas identifier avec certitude tous les programmes problématiques (les questions de ce genre sont généralement algorithmiquement indécidables). On cherche donc à établir des critères décidables qui :

— apportent de la ***sûreté***, c'est-à-dire qui rejettent les programmes absurdes,
— tout en laissant de l'***expressivité***, c'est-à-dire qui ne rejettent pas trop de programmes non-absurdes.

Pour permettre cette analyse des types, on peut être amené à placer un certain nombre d'indications dans le texte d'un programme. Voici quelques possibilités que l'on pourrait imaginer.

1. Annoter toutes les sous-expressions.

```
fun (x : int) ->
  let (y : int) = ((x : int) + (1 : int) : int)
  in (y : int)
```

   Ici, le programmeur fait tout le travail, et le compilateur doit simplement ***vérifier*** la cohérence de l'ensemble. Heureusement, aucun langage n'impose cela.

2. Annoter seulement les variables et les paramètres des fonctions.

```
fun (x : int) -> let (y : int) = x+1 in y
```

   Dans ce cas, le compilateur déduit le type de chaque expression en se référant aux types fournis pour les variables. C'est ce qu'il faut faire en C ou en java.

3. Annoter seulement les paramètres des fonctions.

```
fun (x : int) -> let y = x+1 in y
```

4. Ne rien annoter.

```
fun x -> let y = x+1 in y
```

   Dans ce dernier cas, c'est au compilateur d'***inférer*** le type que doit avoir chaque variable et chaque expression, sans aide du programmeur. C'est ce qui se passe en caml.

Lorsque l'analyse des types est faite à la compilation, la sélection de la bonne instruction en cas d'opérateur surchargé est elle-même faite pendant la compilation, et ne coûte donc plus rien à l'exécution. En outre, la vérification statique de la cohérence des types permet la détection précoce des incohérences du programme : de nombreux problèmes sont corrigés plus tôt.

Dans la suite de ce chapitre, nous allons formaliser la notion de type et les contraintes associées, voir comment réaliser un vérificateur de types ou un programme d'inférence de types, et transformer notre vague notion de sûreté des programmes bien typés en un théorème.

5.2 Jugement de typage et règles d'inférence

Pour caractériser les programmes bien typés, on définit des règles permettant de justifier que « dans un contexte Γ, une expression e est cohérente et admet le type τ ». Cette phrase entre guillemets est appelée un ***jugement de typage*** et est notée

$$\Gamma \vdash e : \tau$$

Le contexte Γ mentionné dans le jugement de typage est l'association d'un type à chaque variable de l'expression e.

Le jugement de typage n'est pas une fonction associant un type à chaque expression, mais simplement une relation entre ces trois éléments : contexte, expression, type. En particulier certaines expressions e *n'ont pas* de type (parce qu'elles sont incohérentes), et dans certaines situations on peut avoir plusieurs types possibles pour une même expression.

Règles de typage Pour illustrer la manière dont on peut formaliser la cohérence et le type d'une expression, concentrons-nous sur un fragment du langage FUN comportant des nombres, des variables et des fonctions :

$$\begin{array}{rcl} e & ::= & n \\ & | & e \texttt{ + } e \\ & | & x \\ & | & \texttt{let } x \texttt{ = } e \texttt{ in } e \\ & | & \texttt{fun } x \texttt{ -> } e \\ & | & e\,e \end{array}$$

Nous aurons donc besoin de manipuler un type de base pour les nombres, et des types de fonctions.

$$\begin{array}{rcl} \tau & ::= & \texttt{int} \\ & | & \tau \to \tau \end{array}$$

Un type de la forme $\tau_1 \to \tau_2$ est le type d'une fonction dont le paramètre attendu a le type τ_1 et le résultat renvoyé a le type τ_2.

À chaque construction du langage, on va associer une règle énonçant
— le type que peut avoir une expression de cette forme, et
— les éventuelles contraintes qui doivent être vérifiées pour que l'expression soit cohérente.

Commençons avec la partie arithmétique. On donnera chaque règle sous deux formes : une description en langue naturelle, et sa traduction comme une ***règle d'inférence*** (voir cours de logique !).

— Une constante entière n admet le type `int`.

$$\frac{}{\Gamma \vdash n \ : \ \texttt{int}}$$

— Si les expressions e_1 et e_2 sont cohérentes et admettent le type `int`, alors l'expression e `+` e est cohérente et admet également le type `int`.

$$\frac{\Gamma \vdash e_1 \ : \ \texttt{int} \qquad \Gamma \vdash e_2 \ : \ \texttt{int}}{\Gamma \vdash e_1 \ \texttt{+} \ e_2 \ : \ \texttt{int}}$$

Dans la règle pour l'addition, les deux jugements $\Gamma \vdash e_1 \ : \ \texttt{int}$ et $\Gamma \vdash e_2 \ : \ \texttt{int}$ sont les prémisses, et le jugement $\Gamma \vdash e_1 \ \texttt{+} \ e_2 \ : \ \texttt{int}$ est la conclusion. Autrement dit, si l'on a pu d'une manière ou l'autre justifier $\Gamma \vdash e_1 \ : \ \texttt{int}$ et $\Gamma \vdash e_2 \ : \ \texttt{int}$, alors la règle permet d'en déduire $\Gamma \vdash e_1 \ \texttt{+} \ e_2 \ : \ \texttt{int}$. À l'inverse, la règle pour la constante entière n'a pas de prémisse (on l'appelle un ***axiome***, ou ***cas de base***). Cela signifie que nous n'avons besoin de rien d'autre que cette règle pour justifier un jugement $\Gamma \vdash n \ : \ \texttt{int}$.

Les règles concernant les variables vont faire intervenir le contexte Γ, aussi appelé ***environnement***, puisque c'est lui qui consigne les types associés à chaque variable.

— Une variable a le type donné par l'environnement.

$$\frac{}{\Gamma \vdash x \ : \ \Gamma(x)}$$

Notez ici que l'on considère Γ comme une fonction : $\Gamma(x)$ désigne le type associé par Γ à la variable x. En outre, l'application de cette règle suppose que $\Gamma(x)$ est bien définie, c'est-à-dire que x appartient au domaine de Γ.

— Une variable locale est associée au type de l'expression qui la définit.

$$\frac{\Gamma \vdash e_1 \ : \ \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 \ : \ \tau_2}{\Gamma \vdash \texttt{let} \ x \ \texttt{=} \ e_1 \ \texttt{in} \ e_2 \ : \ \tau_2}$$

Dans une telle expression, e_2 peut contenir la variable locale x. Le typage de e_2 a donc lieu dans un environnement étendu noté $\Gamma, x \ : \ \tau_1$, qui reprend toutes les associations de Γ et y ajoute l'association du type τ_1 à la variable x. Cette variable x en revanche n'existe pas dans e_1, et n'est pas non plus une variable libre de l'expression complète : elle n'apparaît donc pas dans l'environnement de typage de e_1 ni de `let` x `=` e_1 `in` e_2.

Une fonction a un type de la forme $\alpha \to \beta$, où α désigne le type attendu du paramètre, et β le type du résultat.

— Une fonction doit être appliquée à un paramètre effectif du bon type.

$$\frac{\Gamma \vdash e_1 \ : \ \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 \ : \ \tau_2}{\Gamma \vdash e_1 \, e_2 \ : \ \tau_1}$$

— Dans le corps d'une fonction, le paramètre formel est vu comme une variable dont le type correspond au type attendu pour le paramètre.

$$\frac{\Gamma, x : \tau_1 \vdash e \ : \ \tau_2}{\Gamma \vdash \texttt{fun} \ x \ \texttt{->} \ e \ : \ \tau_1 \to \tau_2}$$

Les règles des ***types simples***, sont donc intégralement contenues dans les six règles d'inférence suivantes.

$$\frac{}{\Gamma \vdash n \ : \ \texttt{int}} \qquad \frac{\Gamma \vdash e_1 \ : \ \texttt{int} \qquad \Gamma \vdash e_2 \ : \ \texttt{int}}{\Gamma \vdash e_1 \ \texttt{+} \ e_2 \ : \ \texttt{int}}$$

$$\frac{}{\Gamma \vdash x \ : \ \Gamma(x)} \qquad \frac{\Gamma \vdash e_1 \ : \ \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 \ : \ \tau_2}{\Gamma \vdash \texttt{let} \ x \ \texttt{=} \ e_1 \ \texttt{in} \ e_2 \ : \ \tau_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash e \ : \ \tau_2}{\Gamma \vdash \texttt{fun} \ x \ \texttt{->} \ e \ : \ \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 \ : \ \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 \ : \ \tau_2}{\Gamma \vdash e_1 \, e_2 \ : \ \tau_1}$$

Expressions typables Pour justifier un jugement de typage pour une expression concrète dans un contexte donné, on enchaîne les déductions à l'aide des règles d'inférence. Ainsi, dans le contexte $\Gamma = \{ x : \texttt{int}, \ f : \texttt{int} \to \texttt{int} \}$ on peut tenir le raisonnement suivant.

1. $\Gamma \vdash x \ : \ \texttt{int}$ est valide, par la règle des variables.
2. $\Gamma \vdash f \ : \ \texttt{int} \to \texttt{int}$ est valide, par la règle des variables.
3. $\Gamma \vdash 1 \ : \ \texttt{int}$ est valide, par la règle des constantes.
4. $\Gamma \vdash f \ 1 \ : \ \texttt{int}$ est valide, par la règle d'application et avec les deux points 2. et 3. déjà justifiés.
5. $\Gamma \vdash x \ \texttt{+} \ f \ 1 \ : \ \texttt{int}$ est valide, par la règle d'addition et avec les deux points 1. et 4. déjà justifiés.

Ce raisonnement, appelé une ***dérivation***, peut également être présenté sous la forme d'un ***arbre de dérivation*** ayant à la racine la conclusion que l'on cherche à justifier.

$$\dfrac{\dfrac{}{\Gamma \vdash x \ : \ \texttt{int}} \qquad \dfrac{\dfrac{}{\Gamma \vdash f \ : \ \texttt{int} \to \texttt{int}} \qquad \dfrac{}{\Gamma \vdash 1 \ : \ \texttt{int}}}{\Gamma \vdash f \ 1 \ : \ \texttt{int}}}{\Gamma \vdash x \ \texttt{+} \ f \ 1 \ : \ \texttt{int}}$$

Dans un tel arbre, chaque barre correspond à une application de règle, et chaque sous-arbre à la justification d'un jugement intermédiaire (une prémisse).

Dans certaines situations, il est possible de dériver plusieurs jugements associant plusieurs types distincts à la même expression. On peut par exemple aussi bien justifier les deux jugements suivants :

$$\vdash \texttt{fun x -> x} \ : \ \texttt{int} \to \texttt{int}$$

$$\vdash \texttt{fun x -> x} \ : \ (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$$

Ici, l'absence de contexte Γ signifie que le typage est effectué dans le contexte vide.

Expressions non typables Si une expression e est incohérente, les règles de typage *ne permettront pas* de justifier de jugements de la forme $\Gamma \vdash e \ : \ \tau$, quels que soient le contexte Γ ou le type τ. On peut le voir en montrant que la construction d'un hypothétique arbre de dérivation justifiant un tel jugement arrive nécessairement à une impasse, c'est-à-dire une situation dans laquelle il est clair que plus aucune règle ne permet de conclure.

Prenons l'exemple de l'expression `5(37)`, que l'on peut encore écrire `5 37`. Il s'agit d'une application. La seule règle permettant de justifier un jugement $\Gamma \vdash \texttt{5 37} \ : \ \tau$ est la règle relative aux applications, qui demande de justifier au préalable les deux prémisses $\Gamma \vdash \texttt{5} \ : \ \tau' \to \tau$ et $\Gamma \vdash \texttt{37} \ : \ \tau'$ (pour un type τ' que l'on peut librement choisir, mais qui doit bien être le même dans les deux jugements). Or il est impossible de justifier une prémisse de la forme $\Gamma \vdash \texttt{5} \ : \ \tau' \to \tau$: aucune règle ne permet d'associer à une constante entière un type de fonction (en effet, la seule règle applicable à une constante entière donnerait $\Gamma \vdash \texttt{5} \ : \ \texttt{int}$).

Considérons le deuxième exemple `fun x -> x x`. De même, une seule règle étant applicable à chaque forme d'expression, un arbre de dérivation d'un jugement $\Gamma \vdash \texttt{fun x -> x x} \ : \ \tau$ aurait nécessairement la forme

$$\dfrac{\dfrac{\Gamma, x : \tau_1 \vdash x \ : \ \tau_1 \to \tau_2 \qquad \Gamma, x : \tau_1 \vdash x \ : \ \tau_1}{\Gamma, x : \tau_1 \vdash x \, x \ : \ \tau_2}}{\Gamma \vdash \texttt{fun x -> x x} \ : \ \tau_2}$$

Or la prémisse $\Gamma, x : \tau_1 \vdash x : \tau_1 \to \tau_2$ est injustifiable : les règles d'inférence ne permettent pas dans ce contexte d'associer à x un autre type que τ_1, et il n'existe pas de types τ_1 et τ_2 vérifiant l'équation $\tau_1 = \tau_1 \to \tau_2$.

Raisonner sur les expressions bien typées Démontrons que pour tout contexte Γ, toute expression e et tout type τ, si $\Gamma \vdash e : \tau$ est valide alors l'ensemble des variables libres de e est inclus dans le domaine de Γ.

Les jugements de typage valides étant définis par un système d'inférence, nous pouvons établir des propriétés vraies pour toutes les expressions bien typées en raisonnant par récurrence sur la structure de la dérivation de typage. Nous avons donc un cas par règle d'inférence, et chaque prémisse de la règle correspondante nous donne une hypothèse de récurrence.

Montrons donc que si $\Gamma \vdash e : \tau$ alors $\mathrm{fv}(e) \subseteq \mathrm{dom}(\Gamma)$, par récurrence sur $\Gamma \vdash e : \tau$.

- Cas $\Gamma \vdash n : \texttt{int}$. On a $\mathrm{fv}(n) = \emptyset$, avec bien sûr $\emptyset \subseteq \mathrm{dom}(\Gamma)$.
- Cas $\Gamma \vdash x : \Gamma(x)$. On a $\mathrm{fv}(x) = \{x\}$, et l'application de la règle suppose justement que $\Gamma(x)$ est bien définie, c'est-à-dire $x \in \mathrm{dom}(\Gamma)$.
- Cas $\Gamma \vdash e_1 + e_2 : \texttt{int}$, avec les deux prémisses $\Gamma \vdash e_1 : \texttt{int}$ et $\Gamma \vdash e_2 : \texttt{int}$. Les deux prémisses nous donnent les deux hypothèses de récurrence $\mathrm{fv}(e_1) \subseteq \mathrm{dom}(\Gamma)$ et $\mathrm{fv}(e_2) \subseteq \mathrm{dom}(\Gamma)$. Or $\mathrm{fv}(e_1 + e_2) = \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2)$. Des deux hypothèses de récurrence on déduit $\mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \subseteq \mathrm{dom}(\Gamma)$, et donc $\mathrm{fv}(e_1 + e_2) \subseteq \mathrm{dom}(\Gamma)$.
- Cas $\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : \tau_2$ avec les deux prémisses $\Gamma \vdash e_1 : \tau_1$ et $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$. Les deux prémisses nous donnent les deux hypothèses de récurrence $\mathrm{fv}(e_1) \subseteq \mathrm{dom}(\Gamma)$ et $\mathrm{fv}(e_2) \subseteq \mathrm{dom}(\Gamma) \cup \{x\}$ (remarquez en effet que la prémisse relative à e_2 est dans un environnement étendu avec la variable x). Or $\mathrm{fv}(\texttt{let } x = e_1 \texttt{ in } e_2) = \mathrm{fv}(e_1) \cup (\mathrm{fv}(e_2) \setminus \{x\})$. Par la première hypothèse de récurrence nous avons $\mathrm{fv}(e_1) \subseteq \mathrm{dom}(\Gamma)$. Par la deuxième hypothèse de récurrence nous avons $\mathrm{fv}(e_2) \subseteq \mathrm{dom}(\Gamma) \cup \{x\}$, dont nous déduisons $\mathrm{fv}(e_2) \setminus \{x\} \subseteq \mathrm{dom}(\Gamma)$. Ainsi on a bien $\mathrm{fv}(\texttt{let } x = e_1 \texttt{ in } e_2) \subseteq \mathrm{dom}(\Gamma)$.
- Les deux cas relatifs aux fonctions sont similaires à ceux déjà traités.

5.3 Vérification de types pour FUN

Si suffisamment d'annotations sont fournies dans le programme source, on peut facilement déduire des règles de typage un ***vérificateur*** de types, c'est-à-dire un (autre) programme qui dit si le programme analysé est cohérent ou non. On va écrire un programme caml pour la vérification des types dans le fragment du langage FUN pour lequel nous venons d'établir des règles de typage. Ce programme prendra la forme d'une fonction `type_expr` qui prend en paramètres une expression e et un environnement Γ et qui :

- renvoie l'unique type qui peut être associé à e dans l'environnement Γ si e est effectivement cohérente dans cet environnement,
- échoue sinon.

On définit un type de données (caml) pour manipuler en caml les types du langage FUN.

```
type typ =
  | TypInt
  | TypFun of typ * typ
```

On ajuste le type caml représentant les arbres de syntaxe abstraite du langage FUN pour y inclure des annotations de type. En l'occurrence on n'introduit cette annotation que pour l'argument d'une fonction : il s'agit du deuxième argument du constructeur `Fun`.

```
type expr =
  | Cst of int
  | Add of expr * expr
  | Var of string
  | Let of string * expr * expr
  | Fun of string * typ * expr
  | App of expr * expr
```

Enfin, on va représenter les environnements comme des tables associatives associant des identifiants de variables (`string`) à des types du langage FUN (`typ`).

```
module Env = Map.Make(String)
type type_env = typ Env.t
```

Le vérificateur est alors une fonction récursive

```
type_expr: expr -> type_env -> typ
```

qui observe la forme de l'expression et traduit la règle d'inférence correspondante.

```
let rec type_expr e env = match e with
```

Une constante entière est toujours cohérente, et de type `int`.

$$\frac{}{\Gamma \vdash n \ : \ \texttt{int}}$$

```
| Cst _ -> TypInt
```

Une variable est considérée comme cohérente si elle existe effectivement dans l'environnement.

$$\frac{}{\Gamma \vdash x \ : \ \Gamma(x)}$$

```
| Var(x) -> Env.find x env
```

Dans le cas contraire, c'est la fonction `Env.find` qui lèvera une exception (en l'occurrence : `Not_found`).

Une addition demande que chaque opérande soit cohérent, et du bon type.

$$\frac{\Gamma \vdash e_1 \ : \ \texttt{int} \qquad \Gamma \vdash e_2 \ : \ \texttt{int}}{\Gamma \vdash e_1 \ \texttt{+} \ e_2 \ : \ \texttt{int}}$$

```
| Add(e1, e2) ->
   let t1 = type_expr e1 env in
   let t2 = type_expr e2 env in
   if t1 = TypInt && t2 = TypInt then
     TypInt
   else
     failwith "type error"
```

Notez que ce code peut échouer à plusieurs endroits différents : pendant la vérification de `e1` ou `e2` si l'une ou l'autre de ces expressions n'est pas cohérente, ou explicitement avec la dernière ligne si `e1` et `e2` sont toutes deux cohérentes mais que l'une n'a pas le type attendu.

Lorsque l'on introduit une variable locale x, on déduit son type de l'expression e_1 définissant cette variable. Le type obtenu peut alors être ajouté à l'environnement pour la vérification du type de la deuxième expression e_2.

$$\frac{\Gamma \vdash e_1 \ : \ \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 \ : \ \tau_2}{\Gamma \vdash \texttt{let} \ x \ \texttt{=} \ e_1 \ \texttt{in} \ e_2 \ : \ \tau_2}$$

```
| Let(x, e1, e2) ->
   let t1 = type_expr e1 env in
   type_expr e2 (Env.add x t1 env)
```

Ce cas n'échoue jamais directement lui-même (les vérifications de `e1` et `e2` peuvent en revanche bien échouer).

Dans le cas d'une fonction, on utilise l'annotation pour fixer le type de l'argument. On peut ensuite vérifier le type du corps de la fonction, en déduire le type du résultat renvoyé, et reconstruire le type complet de la fonction.

$$\frac{\Gamma, x : \tau_1 \vdash e \ : \ \tau_2}{\Gamma \vdash \texttt{fun} \ x \ \texttt{->} \ e \ : \ \tau_1 \to \tau_2}$$

```
| Fun(x, tx, e) ->
   let te = type_expr e (Env.add x tx env) in
   TypFun(tx, te)
```

Dans le cas d'une application, il faut vérifier que le terme de gauche a bien le type d'une fonction, puis que le terme de droite a bien le type attendu par la fonction. Ces deux points donnent deux raisons distinctes d'échouer dans la vérification.

$$\frac{\Gamma \vdash e_1 \ : \ \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 \ : \ \tau_2}{\Gamma \vdash e_1\, e_2 \ : \ \tau_1}$$

```
| App(f, a) ->
   let tf = type_expr f env in
   let ta = type_expr a env in
   begin match tf with
     | TypFun(tx, te) ->
        if tx = ta then
          te
        else
          failwith "type error"
     | _ -> failwith "type error"
   end
```

5.4 Polymorphisme

Avec les types simples que nous avons vu jusqu'ici, une expression comme

```
fun x -> x
```

peut admettre plusieurs types distincts. En revanche, on ne peut lui donner qu'un seul type à la fois. En particulier, dans une expression comme

```
let f = fun x -> x in f f
```

on ne peut donner à `f` qu'un seul des types possibles et l'expression complète ne peut pas être typée. On parle de type ***monomorphe*** (littéralement : *une seule forme*). *Vous pouvez remarquer que caml en revanche n'a pas de difficultés à typer cette expression.*

On s'intéresse dans cette section au ***polymorphisme paramétrique***, c'est-à-dire à la possibilité d'exprimer des types paramétrés, recouvrant plusieurs variantes d'une même forme. On étend la grammaire des types τ en y ajoutant deux éléments :

- des ***variables***, ou ***paramètres***, de type, notées α, β, ... et désignant des types indéterminés,
- une quantification universelle $\forall\alpha.\tau$ décrivant un type ***polymorphe***, où la variable de type α peut, dans τ, désigner n'importe quel type.

Pour notre langage FUN, l'ensemble des types serait donc défini par la grammaire étendue

$$\begin{array}{rcl} \tau & ::= & \texttt{int} \\ & | & \tau \to \tau \\ & | & \alpha \\ & | & \forall\alpha.\tau \end{array}$$

Instanciation L'utilisation d'une expression polymorphe suit le principe suivant : si une expression e admet un type polymorphe $\forall\alpha.\tau$, alors pour tout type τ' on peut encore considérer que e admet le type $\tau[\alpha := \tau']$, c'est-à-dire le type τ dans lequel chaque occurrence du paramètre α a été remplacée par τ'.

$$\frac{\Gamma \vdash e \ : \ \forall\alpha.\tau}{\Gamma \vdash e \ : \ \tau[\alpha := \tau']}$$

La notion de substitution de type $\tau[\alpha := \tau']$ est définie par des équations équivalentes à celles utilisées pour définir la substitution d'expressions au chapitre 2.

$$\begin{array}{rcl} \texttt{int}[\alpha := \tau] & = & \texttt{int} \\ \beta[\alpha := \tau] & = & \begin{cases} \tau & \text{si } \alpha = \beta \\ \beta & \text{si } \alpha \neq \beta \end{cases} \\ (\tau_1 \to \tau_2)[\alpha := \tau] & = & \tau_1[\alpha := \tau] \to \tau_2[\alpha := \tau] \\ (\forall\beta.\tau')[\alpha := \tau] & = & \begin{cases} \forall\beta.\tau' & \text{si } \alpha = \beta \\ \forall\beta.\tau'[\alpha := \tau] & \text{si } \alpha \neq \beta \text{ et } \beta \notin \mathrm{fv}(\tau) \end{cases} \end{array}$$

La notion de variable de type libre est de même définie similairement aux variables libres d'une expression.

$$\begin{array}{rcl} \mathsf{fv}(\texttt{int}) & = & \emptyset \\ \mathsf{fv}(\alpha) & = & \{\alpha\} \\ \mathsf{fv}(\tau_1 \to \tau_2) & = & \mathsf{fv}(\tau_1) \cup \mathsf{fv}(\tau_2) \\ \mathsf{fv}(\forall\alpha.\tau) & = & \mathsf{fv}(\tau) \setminus \{\alpha\} \end{array}$$

Généralisation Lorsqu'une expression admet un type τ contenant un paramètre α, et que ce paramètre *n'est contraint d'aucune manière* par le contexte Γ, alors on peut considérer l'expression e comme polymorphe et lui donner le type $\forall\alpha.\tau$.

$$\frac{\Gamma \vdash e : \tau \qquad \alpha \notin \mathsf{fv}(\Gamma)}{\Gamma \vdash e : \forall\alpha.\tau}$$

Dans la règle d'inférence, notez que la condition demandant que le paramètre α ne soit pas contraint par le contexte est traduite par la non-apparition de α dans le contexte Γ.

Formellement, l'ensemble des variables de type libres d'un environnement $\Gamma = \{ x_1 : \tau_1, \dots, x_n : \tau_n \}$ est défini par l'équation.

$$\mathsf{fv}(x_1 : \tau_1, \dots, x_n : \tau_n) \quad = \quad \bigcup_{1 \leq i \leq n} \mathsf{fv}(\tau_i)$$

Notez que l'on ne parle ici que des variables *de type*. Les x_i, qui sont des variables du langage, ne sont pas concernées.

Exemples et contre-exemples Nous pouvons maintenant donner à la fonction identité `fun x -> x` le type polymorphe $\forall\alpha.\alpha \to \alpha$, exprimant que cette fonction admet un argument de *n'importe quel type* et renvoie un résultat *du même type*.

$$\dfrac{\dfrac{\overline{x : \alpha \vdash x : \alpha}}{\vdash \texttt{fun x -> x} : \alpha \to \alpha} \quad \alpha \notin \mathsf{fv}(\emptyset)}{\vdash \texttt{fun x -> x} : \forall\alpha.\alpha \to \alpha}$$

Notez que la clé de ce raisonnement est la possibilité de donner à `fun x -> x` le type $\alpha \to \alpha$ dans le contexte vide, et donc a fortiori dans un contexte n'imposant aucune contrainte à α.

Il devient alors possibles de typer l'expression `let f = fun x -> x in f f`. En effet, dans la partie de l'arbre de dérivation concernant l'expression `f f`, nous avons un environnement $\Gamma = \{ f : \forall\alpha.\alpha \to \alpha \}$ qui permet de compléter la dérivation ainsi.

$$\dfrac{\dfrac{\overline{\Gamma \vdash f : \forall\alpha.\alpha \to \alpha}}{\Gamma \vdash \texttt{f} : (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})} \quad \dfrac{\overline{\Gamma \vdash f : \forall\alpha.\alpha \to \alpha}}{\Gamma \vdash \texttt{f} : \texttt{int} \to \texttt{int}}}{\Gamma \vdash \texttt{f f} : \texttt{int} \to \texttt{int}}$$

Notez que ce n'est pas la seule solution possible : on aurait également pu laisser à la place du type concret `int` n'importe quelle variable de type β, et même aboutir à la conclusion que le type de `f f` pouvait être généralisé, puisque β n'apparaît pas libre dans Γ.

$$\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash f : \forall\alpha.\alpha \to \alpha}}{\Gamma \vdash \texttt{f} : (\beta \to \beta) \to (\beta \to \beta)} \quad \dfrac{\overline{\Gamma \vdash f : \forall\alpha.\alpha \to \alpha}}{\Gamma \vdash \texttt{f} : \beta \to \beta}}{\Gamma \vdash \texttt{f f} : \beta \to \beta} \quad \beta \notin \mathsf{fv}(\Gamma)}{\Gamma \vdash \texttt{f f} : \forall\beta.\beta \to \beta}$$

Notre système en revanche *ne permet pas* de donner à la fonction identité `fun x -> x` le type $\alpha \to \forall\alpha.\alpha$. En effet, pour cela il faudrait pouvoir, dans un contexte $\Gamma = \{ x : \alpha \}$, donner à x le type $\forall\alpha.\alpha$. Or notre axiome permet seulement de dériver le jugement $\Gamma \vdash x : \alpha$,

dans lequel α ne peut pas être généralisé, puisque justement α apparaît dans Γ. Nous ne pouvons donc (heureusement) pas utiliser le polymorphisme pour typer l'expression mal formée `(fun x -> x) 5 37`.

Démontrons encore que la fonction de composition `fun f -> fun g -> fun x -> g (f x)` admet le type polymorphe $\forall\alpha\beta\gamma.(\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma)$. On note Γ l'environnement $\{f : \alpha \to \beta, g : \beta \to \gamma, x : \alpha\}$. On peut alors produire la dérivation suivante (dans cette dérivation, on a contracté les trois applications successives de la règle de généralisation, ainsi que les trois applications successives de la règle de typage des fonctions).

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \texttt{g} : \beta \to \gamma} \qquad \dfrac{\dfrac{}{\Gamma \vdash \texttt{f} : \alpha \to \beta} \qquad \dfrac{}{\Gamma \vdash \texttt{x} : \alpha}}{\Gamma \vdash \texttt{f x} : \beta}}{\Gamma \vdash \texttt{g (f x)} : \gamma}}{\vdash \texttt{fun f -> fun g -> fun x -> g (f x)} : (\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma)} \; \alpha, \beta, \gamma \notin \mathrm{fv}(\emptyset)$$

$$\vdash \texttt{fun f -> fun g -> fun x -> g (f x)} : \forall\alpha\beta\gamma.(\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma)$$

Exercice : montrer que l'on peut également donner à cette même fonction de composition le type $\forall\alpha\beta.(\alpha \to \beta) \to \forall\gamma.(\beta \to \gamma) \to (\alpha \to \gamma)$.

Système de Hindley-Milner Sans annotations de la part du programmeur, les deux problèmes suivants relatifs au typage polymorphe de FUN sont indécidables :

- l'inférence : c'est-à-dire partant d'une expression e, dire s'il existe un type τ tel que l'on puisse justifier $\Gamma \vdash e : \tau$,
- la vérification : c'est-à-dire partant d'une expression e et d'un type τ, dire si l'on peut ou non justifier $\Gamma \vdash e : \tau$.

Ces résultats d'indécidabilité valent encore évidemment pour tout langage étendant ce noyau.

Pour permettre la vérification algorithmique du bon typage d'un programme, voire l'inférence des types, il faut donc soit imposer un certain nombre d'annotations au programmeur, soit restreindre les possibilités de recours au polymorphisme. Selon les langages, l'équilibre choisi entre la quantité d'annotations requise et l'expressivité du système de types varie.

En caml, le polymorphisme est restreint par le fait que les quantificateurs ne peuvent être qu'implicites : on ne peut pas écrire de quantificateur explicite dans un type, mais en revanche chaque variable de type globalement libre dans le type d'une expression est considérée comme quantifié universellement. Ainsi, le type caml de la première projection d'une paire

```
fst: 'a * 'b -> 'a
```

est en réalité le type universellement quantifié $\forall\alpha\beta.\alpha \times \beta \to \alpha$. De même, le type caml de l'itérateur de liste

```
List.fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

doit être lu comme $\forall\alpha\beta.(\alpha \to \beta \to \alpha) \to \alpha \to \beta\,\texttt{list} \to \alpha$.

Ce polymorphisme restreint, partagé par tous les langages de la famille ML, est le ***système de Hindley-Milner***. Il autorise le quantificateur universel $\forall$ uniquement « en tête » et correspond à séparer la notion de ***type*** τ sans quantificateur de la notion de ***schéma de type*** σ qui est un type devant lequel on a éventuellement ajouté un ou plusieurs quantificateurs.

Ainsi pour FUN :

$$\begin{array}{rcl} \tau & ::= & \texttt{int} \\ & | & \tau \to \tau \\ & | & \alpha \\ \sigma & ::= & \forall\alpha_1 \ldots \forall\alpha_n.\tau \end{array}$$

Dans ce système, on peut ainsi exprimer les schémas de type $\forall\alpha.\alpha \to \alpha$ et $\forall\alpha\beta\gamma.(\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma)$, mais pas un type de la forme $(\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$.

Dans le système de Hindley-Milner, on adapte les notions de contexte et de jugement de typage pour autoriser l'association d'un schéma de types à une variable ou une expression :

$$x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash e : \sigma$$

Notez cependant qu'un schéma avec zéro quantificateur n'est rien d'autre qu'un type simple : cette forme autorise donc de manière générale aussi bien les schémas que les types simples.

Les règles de typage sont également ajustées, de manière à n'autoriser les schémas de type qu'aux endroits où ils sont effectivement acceptables.

$$\frac{}{\Gamma \vdash n \ : \ \texttt{int}} \qquad \frac{\Gamma \vdash e_1 \ : \ \texttt{int} \qquad \Gamma \vdash e_2 \ : \ \texttt{int}}{\Gamma \vdash e_1 \ \texttt{+} \ e_2 \ : \ \texttt{int}}$$

$$\frac{}{\Gamma \vdash x \ : \ \Gamma(x)} \qquad \frac{\Gamma \vdash e_1 \ : \ \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash e_2 \ : \ \sigma_2}{\Gamma \vdash \texttt{let} \ x \ \texttt{=} \ e_1 \ \texttt{in} \ e_2 \ : \ \sigma_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash e \ : \ \tau_2}{\Gamma \vdash \texttt{fun} \ x \ \texttt{->} \ e \ : \ \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 \ : \ \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 \ : \ \tau_2}{\Gamma \vdash e_1 \, e_2 \ : \ \tau_1}$$

$$\frac{\Gamma \vdash e \ : \ \forall \alpha . \sigma}{\Gamma \vdash e \ : \ \sigma[\alpha := \tau]} \qquad \frac{\Gamma \vdash e \ : \ \sigma \qquad \alpha \notin \text{fv}(\Gamma)}{\Gamma \vdash e \ : \ \forall \alpha . \sigma}$$

En l'occurrence la généralisation du type d'une expression est acceptée à deux endroits :

— à la racine, et

— pour l'argument d'un `let`.

Vous pouvez observer ceci dans la règle du typage du `let`. À l'inverse, vous pouvez également observer que les règles de typage d'une fonction ou d'une application imposent que les types de l'argument et du résultat soient tous les deux des types simples.

Le système de types de Hindley-Milner possède deux propriétés intéressantes :

— la vérification *et* l'inférence de types sont décidables (voir section suivante),

— le système est ***sûr***, c'est-à-dire que l'exécution d'un programme bien typé ne peut pas buter sur une incohérence (voir fin du chapitre).

À la fin de ce chapitre, nous verrons comment une telle notion de sûreté peut être formalisée et démontrée. Avant cela, nous allons formaliser la manière dont s'exécute un programme : c'est la question de la ***sémantique***.

5.5 Inférence de types

L'écriture de notre vérificateur de types simples pour FUN était relativement simple grâce à deux caractéristiques de ce système :

— les règles d'inférence étaient purement *syntaxiques*, c'est-à-dire que pour chaque forme d'expression il n'y avait qu'une seule règle d'inférence potentiellement applicable (en anglais on dit *syntax-directed*),

— on avait demandé une annotation pour aider le typage au seul endroit où on n'avait pas de manière simple de choisir le type d'un élément (en l'occurrence, le paramètre d'une fonction).

Dans le système de Hindley-Milner en revanche, les deux règles d'instanciation et de généralisation sont applicables à n'importe quelle forme d'expression a priori, et brisent donc la première propriété. En outre on vise l'***inférence complète***, c'est-à-dire sans aucune annotation.

Système de Hindley-Milner syntaxique Dans une première étape, nous allons donc donner une variante syntaxique du système de Hindley-Milner en restreignant les possibilités d'appliquer les règles de généralisation et d'instanciation.

— On n'autorise l'instanciation d'une variable de type qu'au moment de récupérer dans l'environnement le schéma de type associé à une variable. On obtient cet effet en supprimant la règle d'instanciation et en remplaçant l'axiome $\Gamma \vdash x \ : \ \Gamma(x)$ par une règle qui combine les deux effets suivants :

  1. récupérer le schéma de type σ associé à x dans Γ,
  2. instancier *toutes* les variables universelles de σ (on obtient donc un type simple).

— Symétriquement, on n'autorise la généralisation d'une variable de type qu'au moment d'une définition `let`. On obtient cet effet en supprimant la règle de généralisation et en remplaçant la règle d'inférence pour $\Gamma \vdash \texttt{let} \ x \ \texttt{=} \ e_1 \ \texttt{in} \ e_2 \ : \ \tau_2$ par une variante qui combine les effets suivants :

  1. typer l'expression e_1 dans l'environnement Γ, on note τ_1 le type obtenu,

2. obtenir un schéma σ_1 en généralisant *toutes* les variables de τ_1 qui peuvent l'être,
3. typer e_2 dans l'environnement étendu où x est associé à σ_1.

Notez au passage que l'on n'autorise plus les schémas de types ailleurs que dans le contexte.

Voici une dérivation de typage dans ce système, avec $\Gamma_1 = \{x : \alpha \to \alpha, y : \alpha\}$ et $\Gamma_2 = \{f : \forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash x : \alpha \to \alpha} \quad \dfrac{}{\Gamma_1 \vdash y : \alpha}}{\Gamma_1 \vdash x\,y : \alpha}}{\vdash x\,(x\,y) : \alpha}}{\vdash \texttt{fun}\ y \texttt{->}\ x\,(x\,y) : \alpha \to \alpha}}{\vdash \texttt{fun}\ x \texttt{->}\ \texttt{fun}\ y \texttt{->}\ x\,(x\,y) : (\alpha \to \alpha) \to (\alpha \to \alpha)} \qquad \dfrac{\dfrac{}{\Gamma_2 \vdash f : (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma_2, z : \texttt{int} \vdash z : \texttt{int}} \quad \dfrac{}{\Gamma_2, z : \texttt{int} \vdash 1 : \texttt{int}}}{\Gamma_2, z : \texttt{int} \vdash z\texttt{+}1 : \texttt{int}}}{\Gamma_2 \vdash \texttt{fun}\ z \texttt{->}\ z\texttt{+}1 : \texttt{int} \to \texttt{int}}}{f : \forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha) \vdash f\,(\texttt{fun}\ z \texttt{->}\ z\texttt{+}1) : \texttt{int} \to \texttt{int}}$$

$$\vdash \texttt{let}\ f = \texttt{fun}\ x \texttt{->}\ \texttt{fun}\ y \texttt{->}\ x\,(x\,y)\ \texttt{in}\ f\,(\texttt{fun}\ z \texttt{->}\ z\texttt{+}1) : \texttt{int} \to \texttt{int}$$

On peut démontrer que cette variante syntaxique du système de Hindley-Milner est équivalente à la version d'origine, c'est-à-dire qu'elle permet de dériver les mêmes jugements de typage.

Génération de contraintes et unification Reste donc à réussir à se passer totalement d'annotations. L'algorithme W utilise pour cela la stratégie suivante.

- À chaque fois que l'on doit introduire un type que l'on ne sait pas calculer immédiatement, on introduit à la place une nouvelle variable de type. Cela vaut pour le type du paramètre d'une fonction, mais aussi pour les types instanciant les variables universelles d'un schéma $\Gamma(x)$.
- On détermine la valeur de ces variables de types *plus tard*, au moment de résoudre les contraintes associées aux règles de typage de l'application ou de l'addition.

Lorsqu'une règle de typage impose une égalité entre deux types τ_1 et τ_2 contenant des variables de types α_1, ..., α_n, on cherche à ***unifier*** ces deux types, c'est-à-dire à trouver une instanciation f des variables α_i telle que $f(\tau_1) = f(\tau_2)$.

Exemples d'unification :

- Si $\tau_1 = \alpha \to \texttt{int}$ et $\tau_2 = (\texttt{int} \to \texttt{int}) \to \beta$, on peut unifier τ_1 et τ_2 avec l'instanciation $[\alpha \mapsto \texttt{int} \to \texttt{int}, \beta \mapsto \texttt{int}]$.
- Si $\tau_1 = (\alpha \to \texttt{int}) \to (\alpha \to \texttt{int})$ et $\tau_2 = \beta \to \beta$, on peut unifier τ_1 et τ_2 avec l'instanciation $[\beta \mapsto \alpha \to \texttt{int}]$.
- On ne peut pas unifier $\alpha \to \texttt{int}$ et $\texttt{int}$.
- On ne peut pas unifier $\alpha \to \texttt{int}$ et α.

Critères d'unification :

- τ est toujours unifié à lui-même,
- pour unifier $\tau_1 \to \tau_1'$ et $\tau_2 \to \tau_2'$ on unifie τ_1 avec τ_2 et on unifie τ_1' avec τ_2',
- pour unifier τ et une variable α, lorsque α n'apparaît pas dans τ, on substitue α par τ partout (si α apparaît dans τ l'unification est impossible),
- dans tous les autres cas l'unification est impossible.

Algorithme W sur un exemple On regarde l'expression $\texttt{let}\ f = \texttt{fun}\ x \texttt{->}\ \texttt{fun}\ y \texttt{->}\ x\,(x\,y)\ \texttt{in}\ f\,(\texttt{fun}\ z \texttt{->}\ z\texttt{+}1)$. Pour inférer son type, on s'intéresse d'abord à $\texttt{fun}\ x \texttt{->}\ \texttt{fun}\ y \texttt{->}\ x\,(x\,y)$. On procède ainsi.

- À x on donne le type α, avec α une nouvelle variable de type,
- à y on donne le type β, avec β une nouvelle variable de type,
- on type ensuite l'expression $x\,(x\,y)$.
  - L'application $x\,y$ demande que le type α de x soit un type fonctionnel, dont le paramètre correspond au type β de y. Il faut donc unifier α avec $\beta \to \gamma$, pour γ une nouvelle variable de type. On fixe ainsi $\alpha = \beta \to \gamma$.
  - L'application $x\,y$ a donc le type γ.
  - L'application $x\,(x\,y)$ demande que le type $\alpha = \beta \to \gamma$ de x soit un type fonctionnel dont le paramètre correspond au type γ de $x\,y$. Il faut donc unifier $\beta \to \gamma$ avec $\gamma \to \delta$, pour δ une nouvelle variable de type. On fixe ainsi $\gamma = \delta = \beta$.

  On en déduit que l'application $x\,(x\,y)$ a le type β.
- Finalement, $\texttt{fun}\ x \texttt{->}\ \texttt{fun}\ y \texttt{->}\ x\,(x\,y)$ reçoit le type $\alpha \to (\beta \to \beta)$, c'est-à-dire $(\beta \to \beta) \to (\beta \to \beta)$, et dans le contexte de typage vide on peut généraliser ce type en $\forall\beta.(\beta \to \beta) \to (\beta \to \beta)$.

Dans une deuxième étape, on s'intéresse à l'expression f (`fun` z `->` z`+1`), dans un contexte où f a le type généralisé $\forall\beta.(\beta \to \beta) \to (\beta \to \beta)$. Pour inférer le type de cette application on commence par typer les deux sous-expressions puis on résoud les contraintes.

- Pour f on récupère le schéma de type $\forall\beta.(\beta \to \beta) \to (\beta \to \beta)$ dans le contexte, et on instantie la variable universelle β avec un nouvelle variable de type ζ. On obtient donc pour f le type $(\zeta \to \zeta) \to (\zeta \to \zeta)$.
- Typage de `fun` z `->` z`+1`.
  - À z on donne le type η, avec η une nouvelle variable de type,
  - puis on type l'addition z`+1`.
    - z a le type η, qu'il faut unifier avec `int`. On fixe donc $\eta =$ `int`.
    - 1 a le type `int`, qu'il faut unifier avec `int` : rien à faire.

    Donc z`+1` a le type `int`.

  Donc `fun` z `->` z`+1` a le type $\eta \to$ `int`, c'est-à-dire `int` $\to$ `int`.
- Pour résoudre l'application elle-même, il faut unifier le type $(\zeta \to \zeta) \to (\zeta \to \zeta)$ de f avec le type (`int` $\to$ `int`) $\to \theta$ d'une fonction prenant un paramètre du type `int` $\to$ `int` (le type de `fun` z `->` z`+1`), avec θ une nouvelle variable de type. On fixe ainsi $\zeta =$ `int` et θ, c'est-à-dire `int` $\to$ `int`.

Finalement, l'expression f (`fun` z `->` z`+1`) a le type $\theta =$ `int` $\to$ `int`.

On conclut donc bien

$$\vdash \texttt{let } f = \texttt{fun } x \texttt{ -> fun } y \texttt{ -> } x\ (x\ y) \texttt{ in } f\ (\texttt{fun } z \texttt{ -> } z\texttt{+1}) \ :\ \texttt{int} \to \texttt{int}$$

Algorithme W, en caml On conserve pour les expressions FUN la syntaxe abstraite suivante, sans annotation de types.

```
type expression =
  | Var of string
  | Cst of int
  | Add of expression * expression
  | Fun of string * expression
  | App of expression * expression
  | Let of string * expression * expression
```

On ajoute aux types simples une notion de variable de type, et on définit un schéma de type par une structure avec un type simple `typ` et un ensemble `vars` de variables de types quantifiées universellement.

```
type typ =
  | Tint
  | Tarrow of typ * typ
  | Tvar   of string

module VSet = Set.Make(String)
type schema = { vars: VSet.t; typ: typ }
```

Un environnement de typage associe un schéma de type à chaque variable de programme.

```
module SMap = Map.Make(String)
type env = schema SMap.t
```

On peut alors définir une fonction `type_inference: expression -> typ` calculant un type aussi général que possible pour l'expression donnée en argument. Cette fonction embarque une fonction auxiliaire `new_var: unit -> string` pour la création de nouvelles variables de types.

```
let type_inference t =

  let new_var =
    let cpt = ref 0 in
    fun () -> incr cpt; Printf.sprintf "tvar_%i" !cpt
  in
```

À mesure que des contraintes seront découvertes et analysées, les variables de types introduites avec `new_var` sont destinées à être associées à des types concrets (ou a minima à des types plus précis). Plutôt que de faire les remplacements partout à chaque fois qu'une

nouvelle association est trouvée, on mémorise ces associations dans une table de hachage `subst` qui va grandir progressivement.

```
    let subst = Hashtbl.create 32 in
```

En conséquence, les types manipulés lors de l'inférence vont contenir des variables de types, dont certaines seront associées à une définition dans `subst`. Pour décoder un tel type, on se donne des fonctions auxiliaires de dépliage `unfold` et `unfold_full`, qui remplacent dans un type τ donné en argument les variables de types pour lesquelles on a une définition dans `subst`. La fonction `unfold` fait ce remplacement « en surface », pour découvrir la forme superficielle du type et permettre de distinguer entre les cas `Tint`, `Tarrow` ou `Tvar`. La fonction `unfold_full` fait un remplacement intégral pour connaître le type complet (cette dernière ne servira que pour afficher le verdict à la fin).

```
    let rec unfold t = match t with
      | Tint | Tarrow(_, _) -> t
      | Tvar a ->
         if Hashtbl.mem subst a then
           unfold (Hashtbl.find subst a)
         else
           t
    in

    let rec unfold_full t = match unfold t with
      | Tarrow(t1, t2) -> Tarrow(unfold_full t1, unfold_full t2)
      | t -> t
    in
```

Exemple d'utilisation du dépliage : pour déterminer si une variable de type α apparaît dans un type τ, on raisonne comme d'habitude sur la forme du type τ, mais en intercalant un appel à la fonction de dépliage pour réaliser au préalable les substitutions enregistrées dans `subst`.

```
    let rec occur a t = match unfold t with
      | Tint -> false
      | Tvar b -> a=b
      | Tarrow(t1, t2) -> occur a t1 || occur a t2
    in

    ...
```

Le cœur de l'algorithme W travaille classiquement sur la forme de l'expression analysée. Dans le cas d'une constante, on se contente de renvoyer le type de base `Tint`. Dans le cas d'une addition on infère un type pour chacun des opérandes et on vérifie que les types `t1` et `t2` obtenus sont bien compatibles avec le type `Tint` attendu. Cette vérification de compatibilité est faite par une fonction auxiliaire `unify` qui, éventuellement, enregistrera de nouvelles associations entre des variables de types et des types concrets.

```
    let rec w e env = match e with
      | Cst _ ->
         Tint

      | Add(e1, e2) ->
         let t1 = w e1 env in
         let t2 = w e2 env in
         unify t1 Tint; unify t2 Tint;
         Tint
```

Dans le cas d'une variable, on instancie le schéma de types obtenu dans l'environnement à l'aide d'une fonction auxiliaire `instantiate` qui remplace chaque variable quantifiée universellement par une variable de types fraîche.

```
      | Var x ->
         instantiate (SMap.find x env)
```

Inversement, dans le cas d'un `let` on généralise le type inféré pour l'expression `e1` à l'aide d'une fonction auxiliaire `generalize` qui produit un schéma de type où chaque variable qui peut l'être est quantifiée universellement.

```
    | Let(x, e1, e2) ->
       let t1 = w e1 env in
       let st1 = generalize t1 env in
       let env' = SMap.add x st1 env in
       w e2 env'
```

Pour typer une fonction, on introduit une nouvelle variable pour le type du paramètre. Le type d'un paramètre ne pouvant pas être généralisé, on fixe immédiatement que l'ensemble des variables quantifiées universellement est vide. On infère alors le type du corps de la fonction dans cet environnement étendu.

```
    | Fun(x, e) ->
       let v = new_var() in
       let env = SMap.add x { vars = VSet.empty; typ = Tvar v } env in
       let t = w e env in
       Tarrow(Tvar v, t)
```

Pour typer une application on infère d'abord un type pour chacune des deux sous-expressions. Il faut ensuite résoudre la contrainte de la règle d'application, à savoir que le type `t1` du membre gauche `e1` doit être un type de fonction, et que le type `t2` du membre droit doit être le type attendu du paramètre de cette fonction.

```
    | App(e1, e2) ->
       let t1 = w e1 env in
       let t2 = w e2 env in
       let v = Tvar (new_var()) in
       unify t1 (Tarrow(t2, v));
       v

in
unfold_full (w t SMap.empty)
```

Définition des fonctions auxiliaires citées ci-dessus. Les contraintes sont résolues par un algorithme d'unification, qui prend en paramètres deux types τ_1 et τ_2 et cherche à donner des définitions aux variables de types contenues dans τ_1 et τ_2 de sorte que ces deux types deviennent égaux. Si les deux types ont la même forme, alors il n'y a rien de particulier à faire, si ce n'est poursuivre récursivement l'unification sur les éventuels sous-termes.

```
let rec unify t1 t2 = match unfold t1, unfold t2 with
  | Tint, Tint ->
     ()
  | Tarrow(t1, t1'), Tarrow(t2, t2') ->
     unify t1 t2; unify t1' t2'
```

Lorsqu'apparaît une variable en revanche, on a plusieurs issues possibles :

- Si τ_1 et τ_2 sont la même variable, il n'y a rien à faire : les deux types sont déjà identiques.
- Si l'un des types est une variable α, et si l'autre est une variable différente ou un type d'une autre forme τ, alors on va ajouter une association $[\alpha \mapsto \tau]$ dans `subst`. À noter cependant : si la variable α apparaît dans τ, alors on échoue à la place car il n'est pas possible que α fasse partie de sa propre définition.

```
  | Tvar a, Tvar b when a=a ->
     ()
  | Tvar a, t | t, Tvar a ->
     if occur a t then
       failwith "unification error"
     else
       Hashtbl.add subst a t
```

Enfin, dans tous les autres cas l'unification échoue.

```
  | _, _ ->
     failwith "unification error"
in
```

La fonction auxiliaire d'instanciation génère une nouvelle variable de type pour chaque variable universelle du schéma de types donné en argument, et opère un remplacement.

```
  let instantiate s =
    let renaming = VSet.fold
              (fun v r -> SMap.add v (Tvar(new_var())) r)
              s.vars
              SMap.empty
    in
    let rec rename t = match unfold t with
      | Tvar a as t -> (try SMap.find a renaming with Not_found -> t)
      | Tint -> Tint
      | Tarrow(t1, t2) -> Tarrow(rename t1, rename t2)
    in
    rename s.typ
  in
```

La fonction auxiliaire de généralisation prend en paramètres un type τ et un environnement Γ, et calcule l'ensemble des variables libres de τ qui n'apparaissent pas dans l'environnement Γ. Ces variables sont alors indiquées comme quantifiées universellement.

```
  let rec fvars t = match unfold t with
    | Tint -> VSet.empty
    | Tarrow(t1, t2) -> VSet.union (fvars t1) (fvars t2)
    | Tvar x -> VSet.singleton x
  in
  let rec schema_fvars s =
    VSet.diff (fvars s.typ) s.vars
  in

  let generalize t env =
    let fvt = fvars t in
    let fvenv = SMap.fold
                  (fun _ s vs -> VSet.union (schema_fvars s) vs)
                  env
                  VSet.empty
    in
    { vars = VSet.diff fvt fvenv; typ=t }
  in
```

5.6 Sémantique naturelle

La ***sémantique*** décrit la signification et le comportement des programmes. Un langage de programmation est généralement accompagné d'une description plus ou moins informelle de ce qu'il faut attendre du comportement d'un programme écrit dans ce langage. Voici un extrait de la spécification de Java :

> *The Java programming language guarantees that the operands of operators appear to be evaluated in a specific order, namely, from left to right. It is recommended that code do not rely crucially on this specification.*

Ce genre de document comporte souvent une quantité plus ou moins grande d'imprécisions voire d'ambiguïtés. À l'inverse, on peut également donner à un langage une ***sémantique formelle***, c'est-à-dire une caractérisation mathématique des calculs décrits par un programme. Ce cadre plus rigoureux permet notamment de *raisonner* sur l'exécution des programmes.

Sémantique en appel par valeur Au chapitre 2 nous avons déjà pu voir comment définir une fonction d'interprétation des expressions d'un langage de programmation, c'est-à-dire une fonction eval qui, étant donnés une expression e et un environnement ρ associant des valeurs aux variables libres de e, renvoie le résultat de l'évaluation de e.

Pour notre fragment du langage FUN, une telle fonction peut être définie par les équations

suivantes.

$$\begin{aligned}
\text{eval}(n, \rho) &= n \\
\text{eval}(e_1 \texttt{ + } e_2, \rho) &= \text{eval}(e_1, \rho) + \text{eval}(e_2, \rho) \\
\text{eval}(x, \rho) &= \rho(x) \\
\text{eval}(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2, \rho) &= \text{eval}(e_2, \rho[x = \text{eval}(e_1, \rho)]) \\
\text{eval}(\texttt{fun } x \texttt{ -> } e, \rho) &= \text{Clos}(x, e, \rho) \\
\text{eval}(e_1\ e_2, \rho) &= \text{eval}(e, \rho'[x = \text{eval}(e_2, \rho)]) \\
&\quad \text{si eval}(e_1, \rho) = \text{Clos}(x, e, \rho')
\end{aligned}$$

Notez dans l'équation concernant l'addition que le symbole + dans e_1 `+` e_2 est un élément de syntaxe du langage FUN reliant deux expressions, tandis que l'opérateur + dans $\text{eval}(e_1, \rho) + \text{eval}(e_2, \rho)$ est l'addition mathématique des valeurs v_1 et v_2 produites par l'évaluation des deux expressions e_1 et e_2. Rappelons également que la forme $\text{Clos}(x, e, \rho)$ désigne une fermeture, c'est-à-dire une fonction accompagnée de son environnement.

Notez que l'environnement ρ manipulé par cette fonction d'évaluation, et la nécessité qui en découle d'introduire une notion de fermeture pour représenter les fonctions comme des valeurs, est un dispositif qui était avant tout destiné à produire un interprète efficace. Pour une spécification mathématique de la valeur que doit produire l'évaluation d'une expression, et dans un tel cadre purement fonctionnel, on peut ne manipuler que des expressions dans lesquelles chaque variable est remplacée par sa valeur, et ainsi contourner cette notion d'environnement. On aurait ainsi une définition comme

$$\begin{aligned}
\text{eval}(n) &= n \\
\text{eval}(e_1 \texttt{ + } e_2) &= \text{eval}(e_1) + \text{eval}(e_2) \\
\text{eval}(x) &= \text{indéfini} \\
\text{eval}(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2) &= \text{eval}(e_2[x := \text{eval}(e_1)]) \\
\text{eval}(\texttt{fun } x \texttt{ -> } e) &= \texttt{fun } x \texttt{ -> } e \\
\text{eval}(e_1\ e_2) &= \text{eval}(e[x = \text{eval}(e_2)]) \\
&\quad \text{si eval}(e_1) = \texttt{fun } x \texttt{ -> } e
\end{aligned}$$

où la substitution $e[x := e']$ dans l'expression e de chaque occurrence de la variable x par l'expression e' est définie selon les lignes habituelles par

$$\begin{aligned}
n[x := e'] &= n \\
(e_1 \texttt{ + } e_2)[x := e'] &= e_1[x := e'] \texttt{ + } e_2[x := e'] \\
y[x := e'] &= \begin{cases} e' & \text{si } x = y \\ y & \text{sinon} \end{cases} \\
(\texttt{let } y \texttt{ = } e_1 \texttt{ in } e_2)[x := e'] &= \begin{cases} \texttt{let } y \texttt{ = } e_1[x := e'] \texttt{ in } e_2 & \text{si } x = y \\ \texttt{let } y \texttt{ = } e_1[x := e'] \texttt{ in } e_2[x := e'] & \text{si } x \neq y \text{ et } y \notin \text{fv}(e') \end{cases} \\
(\texttt{fun } y \texttt{ -> } e)[x := e'] &= \begin{cases} \texttt{fun } y \texttt{ -> } e & \text{si } x = y \\ \texttt{fun } y \texttt{ -> } e[x := e'] & \text{si } x \neq y \text{ et } y \notin \text{fv}(e') \end{cases} \\
(e_1\ e_2)[x := e'] &= e_1[x := e']\ e_2[x := e']
\end{aligned}$$

Cette approche de la sémantique d'un programme est surtout adaptée à la description de programmes déterministes et dont l'exécution de passe bien.

Une approche limitant ces présupposés consiste à définir la sémantique comme une relation entre les expressions et les valeurs. On noterait ainsi

$$e \Longrightarrow v$$

pour toute paire d'une expression e et d'une valeur v telle que l'expression e peut s'évaluer en la valeur v.

Cette relation, appelée ***sémantique naturelle***, ou ***sémantique à grands pas***, est définie par des règles d'inférence et spécifie les évaluations possibles des expressions. Un compilateur est tenu de respecter la sémantique de son langage source.

Pour asseoir notre formalisation, précisons l'ensemble des valeurs que nous considérons : les nombres entiers et les fonctions.

$$\begin{aligned}
v &::= n \\
&\mid \texttt{fun } x \texttt{ -> } e
\end{aligned}$$

Les règles d'inférence vont ensuite traduire les équations définissant notre précédente fonction eval.

— eval(n) = n. Une constante entière est sa propre valeur. La règle associée est un axiome.

$$\frac{}{n \implies n}$$

— eval(e_1 `+` e_2) = eval(e_1) + eval(e_2). La valeur d'une expression d'addition est obtenue en ajoutant les valeurs de chacune des deux sous-expressions.

$$\frac{e_1 \implies n_1 \qquad e_2 \implies n_2}{e_1 \texttt{ + } e_2 \implies n_1 + n_2}$$

Notez que cette règle ne peut s'appliquer que si les valeurs n_1 et n_2 associées à e_1 et e_2 sont bien des nombres.

— eval(`let` x `=` e_1 `in` e_2) = eval($e_2[x$ `:=` eval(e_1)$]$). La valeur d'une expression e_2 comportant une variable locale x est obtenue en évaluant e_2 après substitution de x par la valeur de l'expression e_1 associée.

$$\frac{e_1 \implies v_1 \qquad e_2[x := v_1] \implies v}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \implies v}$$

— eval(`fun` x `->` e) = `fun` x `->` e. Une fonction est sa propre valeur. Comme pour les constantes, la règle associée est un axiome.

$$\frac{}{\texttt{fun } x \texttt{ -> } e \implies \texttt{fun } x \texttt{ -> } e}$$

— eval(e_1 e_2) = eval($e[x$ = eval(e_2)$]$) si eval(e_1) = `fun` x `->` e. Pour que la valeur d'une application soit bien définie, il faut que son membre gauche e_1 ait pour valeur une fonction. La valeur de l'application est alors obtenue en substituant le paramètre formel dans le corps de la fonction par la valeur de l'argument e_2, puis en évaluant l'expression obtenue.

$$\frac{e_1 \implies \texttt{fun } x \texttt{ -> } e \qquad e_2 \implies v_2 \qquad e[x := v_2] \implies v}{e_1\ e_2 \implies v}$$

Nous obtenons donc pour notre fragment du langage FUN cinq règles d'inférence, et nous pouvons justifier un jugement sémantique de la forme $e \implies v$ à l'aide d'une dérivation.

$$\dfrac{\dfrac{}{\texttt{fun } x\texttt{->}x\texttt{+}x \implies \texttt{fun } x\texttt{->}x\texttt{+}x} \qquad \dfrac{\dfrac{\dfrac{}{20 \implies 20} \qquad \dfrac{}{1 \implies 1}}{20\texttt{+}1 \implies 21} \qquad \dfrac{\dfrac{}{21 \implies 21} \qquad \dfrac{}{21 \implies 21}}{21\texttt{+}21 \implies 42}}{(\texttt{fun } x \texttt{ -> } x \texttt{ + } x)\,(20 \texttt{ + } 1) \implies 42}}{\texttt{let } f \texttt{ = fun } x \texttt{ -> } x \texttt{ + } x \texttt{ in } f(20 \texttt{ + } 1) \implies 42}$$

Notez que la règle proposée pour l'application de fonction évalue l'argument e_2 avant de le substituer dans le corps de la fonction. Ce comportement vient de la fonction d'interprétation qui nous a servi de base, et qui réalisait la stratégie d'*appel par valeur*.

Sémantique en appel par nom On pourrait définir une variante de cette sémantique basée sur la stratégie d'*appel par nom*. Cette variante consiste essentiellement à remplacer la règle relative à l'application par la version plus simple suivante

$$\frac{e_1 \implies \texttt{fun } x \texttt{ -> } e \qquad e[x := e_2] \implies v}{e_1\ e_2 \implies v}$$

où l'argument e_2 est substitué tel quel.

Dans une sémantique en appel par nom on peut également, optionnellement, utiliser la variante suivante de la règle de `let`.

$$\frac{e_2[x := e_1] \implies v}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \implies v}$$

La sémantique en appel par nom est *presque* équivalente à la sémantique en appel par valeur : elles permettent en grande partie de dériver les mêmes jugements $e \Longrightarrow v$ (la forme de la dérivation peut changer, mais le seul fait qui compte est qu'une dérivation *existe*). En l'occurrence, on peut dériver

$$\texttt{let } f \texttt{ = fun } x \texttt{ -> } x \texttt{ + } x \texttt{ in } f(20 \texttt{ + } 1) \Longrightarrow 42$$

en appel par nom de la manière suivante.

$$\dfrac{\dfrac{}{\texttt{fun } x\texttt{->}x\texttt{+}x \Longrightarrow \texttt{fun } x\texttt{->}x\texttt{+}x} \qquad \dfrac{\dfrac{\dfrac{}{20 \Longrightarrow 20} \quad \dfrac{}{1 \Longrightarrow 1}}{20\texttt{+}1 \Longrightarrow 21} \qquad \dfrac{\dfrac{}{20 \Longrightarrow 20} \quad \dfrac{}{1 \Longrightarrow 1}}{20\texttt{+}1 \Longrightarrow 21}}{(20\texttt{+}1)\texttt{+}(20\texttt{+}1) \Longrightarrow 42}}{(\texttt{fun } x \texttt{ -> } x \texttt{ + } x)\,(20 \texttt{ + } 1) \Longrightarrow 42}$$
$$\texttt{let } f \texttt{ = fun } x \texttt{ -> } x \texttt{ + } x \texttt{ in } f(20 \texttt{ + } 1) \Longrightarrow 42$$

Question : comment l'appel par nom se traduit-il dans la forme de l'arbre ?

Insistons cependant sur le fait que ces deux sémantiques ne sont *pas totalement équivalentes* : il existe des jugements $e \Longrightarrow v$ qui peuvent être dérivés dans l'une et non dans l'autre. *Question : pouvez-vous en trouver ?*

Raisonner sur la sémantique Du fait que la sémantique naturelle est définie par un système d'inférence, nous pouvons démontrer des propriétés des programmes et de leur sémantique en raisonnant par récurrence sur la dérivation d'un jugement $e \Longrightarrow v$. Comme nous l'avons déjà vu avec la récurrence sur les jugements de typage, on a alors un cas de preuve par règle d'inférence de la sémantique, et les prémisses des règles fournissent à chaque fois des hypothèses de récurrence.

Utilisons la sémantique naturelle en appel par nom de FUN,

$$\frac{}{n \Longrightarrow n} \qquad \frac{e_1 \Longrightarrow n_1 \qquad e_2 \Longrightarrow n_2}{e_1 \texttt{ + } e_2 \Longrightarrow n_1 + n_2} \qquad \frac{e_2[x := e_1] \Longrightarrow v}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \Longrightarrow v}$$

$$\frac{}{\texttt{fun } x \texttt{ -> } e \Longrightarrow \texttt{fun } x \texttt{ -> } e} \qquad \frac{e_1 \Longrightarrow \texttt{fun } x \texttt{ -> } e \qquad e[x := e_2] \Longrightarrow v}{e_1\, e_2 \Longrightarrow v}$$

et démontrons que si $e \Longrightarrow v$ alors v est une valeur et $\mathrm{fv}(v) \subseteq \mathrm{fv}(e)$, par récurrence sur la dérivation de $e \Longrightarrow v$.

- Cas $n \Longrightarrow n$: immédiat, car n est bien une valeur, et $\mathrm{fv}(n) \subseteq \mathrm{fv}(n)$.
- Cas $\texttt{fun } x \texttt{ -> } e \Longrightarrow \texttt{fun } x \texttt{ -> } e$ immédiat de même.
- Cas $e_1 \texttt{ + } e_2 \Longrightarrow n_1 + n_2$ avec $e_1 \Longrightarrow n_1$ et $e_2 \Longrightarrow n_2$. Par définition $n_1 + n_2$ est une valeur entière. En outre $\mathrm{fv}(n_1 + n_2) = \emptyset \subseteq \mathrm{fv}(e_1 \texttt{ + } e_2)$. *Note : les hypothèses de récurrence concernant e_1 et e_2 ne sont même pas utiles dans ce cas.*
- Cas $\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \Longrightarrow v$ avec $e_2[x := e_1] \Longrightarrow v$. La prémisse donne comme hypothèse de récurrence que $\mathrm{fv}(v) \subseteq \mathrm{fv}(e_2[x := e_1])$ (et que v est une valeur).
  On a besoin ici d'un lemme sur les variables libres d'un terme soumis à une substitution. On utilisera l'égalité suivante (démontrée juste après, par récurrence sur l'expression e).

  $$\mathrm{fv}(e[x := e']) = (\mathrm{fv}(e) \setminus \{x\}) \cup \mathrm{fv}(e')$$

  Avec le lemme, on a $\mathrm{fv}(v) \subseteq (\mathrm{fv}(e_2) \setminus \{x\}) \cup \mathrm{fv}(e_1)$. Or par définition

  $$\mathrm{fv}(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2) = \mathrm{fv}(e_1) \cup (\mathrm{fv}(e_2) \setminus x)$$

  On a donc bien $\mathrm{fv}(v) \subseteq \mathrm{fv}(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2)$.
- Cas $e_1\, e_2 \Longrightarrow v$ avec $e_1 \Longrightarrow \texttt{fun } x \texttt{ -> } e$ et $e[x := e_2] \Longrightarrow v$. Les deux prémisses donnent comme hypothèses de récurrence que v est une valeur, que $\mathrm{fv}(\texttt{fun } x \texttt{ -> } e) \subseteq \mathrm{fv}(e_1)$ et que $\mathrm{fv}(v) \subseteq \mathrm{fv}(e[x := e_2])$. Avec le lemme précédent on a donc

  $$\begin{array}{rcl} \mathrm{fv}(v) & \subseteq & \mathrm{fv}(e[x := e_2]) \\ & = & (\mathrm{fv}(e) \setminus \{x\}) \cup \mathrm{fv}(e_2) \\ & = & \mathrm{fv}(\texttt{fun } x \texttt{ -> } e) \cup \mathrm{fv}(e_2) \\ & \subseteq & \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \\ & = & \mathrm{fv}(e_1\, e_2) \end{array}$$

Premier lien entre typage et sémantique Avec cette formalisation de la sémantique naturelle, on peut démontrer l'énoncé suivant liant le typage et la sémantique d'une expression.

$$\text{Si } \Gamma \vdash e : \tau \text{ et } e \Longrightarrow v \text{ alors } \Gamma \vdash v : \tau.$$

Cette propriété exprime que l'évaluation d'un programme préserve sa cohérence et son type.

Notez cependant que cette propriété part de l'hypothèse que l'évaluation du programme aboutit. Autrement dit, elle ne démontre pas que l'évaluation d'un programme bien typé aboutit, et ne dit rien des programmes qui plantent ni des programmes qui bouclent. Il va falloir préciser la formalisation de la sémantique pour permettre d'énoncer une véritable propriété de sûreté.

5.7 Sémantique opérationnelle à petits pas

Nous avons vu que la sémantique naturelle ne parle que des calculs qui aboutissent. En particulier, elle ne dit rien des calculs qui échouent à cause d'un blocage, comme l'évaluation de `5(37)`, ni des calculs qui ne terminent jamais, comme l'évaluation de `(fun x -> x x) (fun x -> x x)`. Elle n'est pas même capable de distinguer ces deux situations.

La ***sémantique à petits pas*** nous donne plus de précision en décomposant la relation d'évaluation $e \Longrightarrow v$ en une série d'étapes de calcul $e \to e_1 \to e_2 \to \ldots \to v$. On obtient alors les moyens de distinguer les trois situations suivantes :

— un calcul qui, après un nombre fini d'étapes, aboutit à un résultat :

$$e \to e_1 \to e_2 \to \ldots \to v$$

où v est une valeur,

— un calcul qui, après un certain nombre d'étapes, aboutit à un blocage :

$$e \to e_1 \to e_2 \to \ldots \to e_n$$

où e_n n'est pas valeur mais ne peut plus être évaluée,

— un calcul qui se poursuit indéfiniment :

$$e \to e_1 \to e_2 \to \ldots$$

où les étapes s'enchaînent à l'infini sans jamais aboutir à une expression finale.

Règles de calculs La sémantique à petits pas est définie par la relation $e \to e'$, appelée ***relation de réduction*** et désignant l'application d'un unique pas de calcul. Cette relation est à nouveau définie par un système de règles d'inférence. On fournit d'une part des règles de calcul élémentaires, formant des cas de base, et d'autre part des règles d'inférence permettant d'appliquer les règles de calcul dans des sous-expressions.

Commençons par détailler les axiomes pour notre fragment de FUN. Ils correspondent aux règles de calcul de base, appliquées directement à la racine d'une expression.

— Axiome pour l'application de fonction en appel par valeur. Si une fonction `fun` x `->` e est appliquée à une valeur v, alors on peut substituer v à chaque occurrence du paramètre formel x dans le corps de la fonction e.

$$\frac{}{(\texttt{fun } x \texttt{ -> } e)\ v \to e[x := v]}$$

L'application de cette règle suppose que l'argument de l'application a été évalué lors des étapes précédentes du calcul.

— Axiome pour le remplacement d'une variable locale par sa valeur.

$$\frac{}{\texttt{let } x \texttt{ = } v \texttt{ in } e \to e[x := v]}$$

Comme pour l'application de fonction, cette règle n'est applicable que si la valeur v associée à la variable x a déjà été calculée.

— Axiome pour l'addition.

$$\frac{n_1 + n_2 = n}{n_1 \texttt{ + } n_2 \to n}$$

Attention au jeu d'écriture ici : on passe de l'expression n_1 `+` n_2, où le symbole + est un élément de syntaxe, au résultat n de l'addition mathématique des deux nombres n_1 et n_2. Encore une fois en revanche, l'application de cette règle présuppose que les deux opérandes ont déjà été évalués, et que les valeurs obtenues sont bien des nombres.

Les règles d'inférence décrivent ensuite la manière dont les règles de base peuvent être appliquées à des sous-expressions.

— Règles d'inférence pour l'addition. Dans une expression de la forme e_1 `+` e_2, il est possible d'effectuer un pas de calcul dans l'une ou l'autre des deux sous-expressions e_1 et e_2. On traduit cela par deux règles d'inférences, une concernant chaque expression.

$$\frac{e_1 \to e_1'}{e_1 \texttt{ + } e_2 \to e_1' \texttt{ + } e_2} \qquad \frac{e_2 \to e_2'}{e_1 \texttt{ + } e_2 \to e_1 \texttt{ + } e_2'}$$

Avec ces règles, on peut dériver le fait qu'une étape de calcul mène de l'expression `((1+2)+3)+(4+5)` à l'expression `(3+3)+(4+5)`.

$$\frac{\dfrac{\dfrac{1 + 2 = 3}{\texttt{1 + 2} \to \texttt{3}}}{\texttt{(1 + 2) + 3} \to \texttt{3 + 3}}}{\texttt{((1 + 2) + 3) + (4 + 5)} \to \texttt{(3 + 3) + (4 + 5)}}$$

Notez que ces règles n'imposent rien sur l'ordre dans lequel évaluer les deux sous-expressions e_1 et e_2. Elles permettent même d'alterner des étapes de calcul de manière arbitraire entre les deux côtés. On peut ainsi dériver la suite d'étapes de calcul suivante.

```
((1+2)+3)+(4+5) → (3+3)+(4+5) → (3+3)+9 → 6+9 → 15
```

Si on voulait forcer l'évaluation des opérandes de gauche à droite, il faudrait remplacer la deuxième règle par la variante suivante, qui autorise l'application d'une étape de calcul dans l'opérande de droite sous condition que l'opérande de gauche soit déjà une valeur.

$$\frac{e_2 \to e_2'}{v_1 \texttt{ + } e_2 \to v_1 \texttt{ + } e_2'}$$

— Règles d'inférence pour une définition de variable locale. La règle ci-dessous autorise l'application d'un pas de calcul dans l'expressions e_1 définissant la valeur d'une variable locale x.

$$\frac{e_1 \to e_1'}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \to \texttt{let } x \texttt{ = } e_1' \texttt{ in } e_2}$$

— Règles d'inférence pour les applications. Les deux règles ci-dessous autorisent l'application d'un pas de calcul du côté gauche d'une application sans condition, et du côté droit d'une application dont le côté gauche a déjà été évalué.

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ e_2'}$$

Notez qu'aucune règle n'autorise l'application d'un pas de calcul à l'intérieur du corps e d'une fonction `fun` x `->` e. En effet, une telle fonction est déjà une valeur, et n'a pas à être évaluée plus à ce stade ! Une fois que cette fonction aura reçu un argument v en revanche, et que cet argument aura été substitué à x dans e, alors le calcul pourra se poursuivre à cet endroit.

Bilan du système d'inférence définissant une sémantique à petits pas pour FUN, en appel par valeur avec évaluation de gauche à droite des opérandes.

$$\frac{e_1 \to e_1'}{e_1 \texttt{ + } e_2 \to e_1' \texttt{ + } e_2} \qquad \frac{e_2 \to e_2'}{v_1 \texttt{ + } e_2 \to v_1 \texttt{ + } e_2'} \qquad \frac{n_1 + n_2 = n}{n_1 \texttt{ + } n_2 \to n}$$

$$\frac{e_1 \to e_1'}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \to \texttt{let } x \texttt{ = } e_1' \texttt{ in } e_2} \qquad \frac{}{\texttt{let } x \texttt{ = } v \texttt{ in } e \to e[x := v]}$$

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ e_2'} \qquad \frac{}{(\texttt{fun } x \texttt{ -> } e)\ v \to e[x := v]}$$

Notez qu'avec ce niveau de détail dans le déroulement des calculs, on peut très facilement imaginer des variantes correspondant à d'autres stratégies d'évaluation.

Exercice : définir une sémantique à petits pas pour FUN en appel par nom.

Séquences de réduction La relation $e \to e'$ décrit les pas de calcul élémentaires. On note donc

- $e \to e'$ lorsque 1 pas de calcul mène de e à e', et
- $e \to^* e'$ lorsque qu'un calcul mène de e à e' en 0, 1 ou plusieurs pas (on parle alors d'une ***séquence*** de calcul ou d'une séquence de réduction).

Une expression ***irréductible*** est une expression e à partir de laquelle on ne peut plus effectuer de pas de calcul, c'est-à-dire pour laquelle il n'existe pas d'expression e' telle que $e \to e'$. Une expression irréductible peut être deux choses :

- une valeur, c'est-à-dire le résultat attendu d'un calcul,
- une expression bloquée, c'est-à-dire une expression décrivant un calcul qui n'est pas terminé, mais pour laquelle aucune règle ne permet de poursuivre.

Exemple de réduction aboutissant à une valeur.

```
let f = fun x -> x + x in f (20 + 1)
→  (fun x -> x + x) (20 + 1)
→  (fun x -> x + x) 21
→  21 + 21
→  42
```

Exemple de réduction aboutissant à un blocage.

```
let f = fun x -> fun y -> x + y in 1 + f 2
→  1 + (fun x -> fun y -> x + y) 2
→  1 + (fun y -> 2 + y)
```

5.8 Équivalence petits pas et grands pas

Les sémantiques à grands pas et à petits pas donnent des points de vue légèrement différents : la première donne une vision directe du résultat qui peut être attendu d'un programme, alors que la deuxième donne une vision plus précise des différentes opérations effectuées. Ces deux modes de présentation de la sémantique sont cependant *équivalents*, dans le sens qu'ils spécifient les mêmes relations d'évaluation. Autrement dit,

$$e \Longrightarrow v \qquad \text{si et seulement si} \qquad e \to^* v$$

Démontrons-le pour les deux versions de la sémantique en appel par valeur de FUN. Nous prenons la sémantique à grands pas définie par les règles d'inférence

$$\frac{}{n \Longrightarrow n} \qquad \frac{}{\texttt{fun } x \texttt{ -> } e \Longrightarrow \texttt{fun } x \texttt{ -> } e}$$

$$\frac{e_1 \Longrightarrow n_1 \qquad e_2 \Longrightarrow n_2}{e_1 \texttt{ + } e_2 \Longrightarrow n_1 + n_2} \qquad \frac{e_1 \Longrightarrow v_1 \qquad e_2[x := v_1] \Longrightarrow v}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \Longrightarrow v}$$

$$\frac{e_1 \Longrightarrow \texttt{fun } x \texttt{ -> } e \qquad e_2 \Longrightarrow v_2 \qquad e[x := v_2] \Longrightarrow v}{e_1\ e_2 \Longrightarrow v}$$

et la sémantique à petits pas définie par les règles d'inférence suivantes.

$$\frac{e_1 \to e_1'}{e_1 \texttt{ + } e_2 \to e_1' \texttt{ + } e_2} \qquad \frac{e_2 \to e_2'}{v_1 \texttt{ + } e_2 \to v_1 \texttt{ + } e_2'} \qquad \frac{n_1 + n_2 = n}{n_1 \texttt{ + } n_2 \to n}$$

$$\frac{e_1 \to e_1'}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \to \texttt{let } x \texttt{ = } e_1' \texttt{ in } e_2} \qquad \frac{}{\texttt{let } x \texttt{ = } v \texttt{ in } e \to e[x := v]}$$

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ e_2'} \qquad \frac{}{(\texttt{fun } x \texttt{ -> } e)\ v \to e[x := v]}$$

$e \Longrightarrow v$ **implique** $e \to^* v$ Démontrons cette implication par récurrence sur la dérivation de $e \Longrightarrow v$.

- Cas $n \Longrightarrow n$. On a bien $n \to^* n$ avec une séquence de 0 pas.
- Cas $\texttt{fun } x \texttt{ -> } e \Longrightarrow \texttt{fun } x \texttt{ -> } e$ immédiat de même.
- Cas $e_1 \texttt{ + } e_2 \Longrightarrow n$ avec $e_1 \Longrightarrow n_1$, $e_2 \Longrightarrow n_2$ et $n = n_1 + n_2$. Les deux prémisses nous donnent les hypothèses de récurrence $e_1 \to^* n_1$ et $e_2 \to^* n_2$. De $e_1 \to^* n_1$ on déduit $e_1 \texttt{ + } e_2 \to^* n_1 \texttt{ + } e_2$ (à strictement parler, il s'agit d'un lemme, à démontrer par récurrence sur la longueur de la séquence de réduction $e_1 \to^* n_1$). De même, de $e_2 \to^* n_2$ on déduit $n_1 \texttt{ + } e_2 \to^* n_1 \texttt{ + } n_2$. En ajoutant une dernière étape avec la règle de réduction de base de l'addition nous obtenons la séquence

  $$\begin{array}{lll} e_1 \texttt{ + } e_2 & \to^* & n_1 \texttt{ + } e_2 \\ & \to^* & n_1 \texttt{ + } n_2 \\ & \to & n \end{array}$$

  qui conclut.
- Cas $\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \Longrightarrow v$ avec $e_1 \Longrightarrow v_1$ et $e_2[x := v_1] \Longrightarrow v$. Les deux prémisses nous donnent les hypothèses de récurrence $e_1 \to^* v_1$ et $e_2[x := v_1] \to^* v$. On en déduit la séquence de réduction

  $$\begin{array}{lll} \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 & \to^* & \texttt{let } x \texttt{ = } v_1 \texttt{ in } e_2 \\ & \to & e_2[x := v_1] \\ & \to^* & v \end{array}$$

- Cas $e_1\ e_2 \Longrightarrow v$ avec $e_1 \Longrightarrow \texttt{fun } x \texttt{ -> } e$, $e_2 \Longrightarrow v_2$ et $e[x := v_2] \Longrightarrow v$. Les trois prémisses nous donnent les hypothèses de récurrence $e_1 \to^* \texttt{fun } x \texttt{ -> } e$, $e_2 \to^* v_2$ et $e[x := v_2] \to^* v$. On en déduit la séquence de réduction

  $$\begin{array}{lll} e_1\ e_2 & \to^* & (\texttt{fun } x \texttt{ -> } e)\ e_2 \\ & \to^* & (\texttt{fun } x \texttt{ -> } e)\ v_2 \\ & \to & e[x := v_2] \\ & \to^* & v \end{array}$$

$e \to^* v$ **implique** $e \Longrightarrow v$ Notez que dans cet énoncé, en écrivant $e \to^* v$ on suppose que v est une valeur. Démontrons cette implication par récurrence sur la longueur de la séquence de réduction $e \to^* v$.

- Cas d'une séquence de longueur 0. L'expression e est donc déjà une valeur, c'est-à-dire de la forme n ou de la forme $\texttt{fun } x \texttt{ -> } e'$. On conclut donc immédiatement avec l'un des axiomes

  $$\frac{}{n \Longrightarrow n} \qquad \frac{}{\texttt{fun } x \texttt{ -> } e' \Longrightarrow \texttt{fun } x \texttt{ -> } e'}$$

- Cas d'une séquence $e \to^* v$ de longueur $n+1$, en supposant que pour toute séquence de réduction $e' \to^* v$ de longueur n on a bien $e' \Longrightarrow v$ (c'est notre hypothèse de récurrence). Notons

  $$e \to e' \to \ldots \to v$$

  notre séquence de réduction $e \to^* v$ de $n+1$ étapes, avec e' l'expression obtenue au terme de la première étape. Nous avons donc $e' \to^* v$ en n étapes, et donc par hypothèse de récurrence $e' \Longrightarrow v$.

Pour conclure, on démontre que de manière générale, si $e \to e'$ et $e' \implies v$ alors $e \implies v$.

Lemme : si $e \to e'$ et $e' \implies v$ alors $e \implies v$. Démonstration par récurrence sur la dérivation de $e \to e'$.

— Cas $n_1 + n_2 \to n$ avec $n = n_1 + n_2$. On a dans ce cas également $n \implies v$, qui n'est possible qu'avec $v = n$. On conclut donc avec la dérivation

$$\frac{\overline{n_1 \implies n_1} \qquad \overline{n_2 \implies n_2}}{n_1 + n_2 \implies n}$$

— Cas $\texttt{let } x \texttt{ = } w \texttt{ in } e \to e[x := w]$, avec w une valeur et où l'hypothèse $e' \implies v$ s'écrit $e' = e[x := w] \implies v$. On conclut donc avec la dérivation

$$\frac{w \implies w \qquad e[x := w] \implies v}{\texttt{let } x \texttt{ = } w \texttt{ in } e \implies v}$$

Notez que nous n'avons pas utilisé d'hypothèse de récurrence ici !

— Cas $e_1 + e_2 \to e_1' + e_2$ avec $e_1 + e_1'$ et où l'hypothèse $e' \implies v$ s'écrit $e_1' + e_2 \implies v$. La prémisse $e_1 + e_1'$ nous donne comme hypothèse de récurrence « pour toute valeur v_1 telle que $e_1' \implies v_1$ on a $e_1 \implies v_1$ ».
Par inversion du jugement $e_1' + e_2 \implies v$ on déduit que v s'exprime $n_1 + n_2$ avec n_1 et n_2 tels que $e_1' \implies n_1$ et $e_2 \implies n_2$. Avec l'hypothèse de récurrence on déduit donc $e_1 \implies n_1$. Grâce à ce jugement, on peut construire la dérivation suivante.

$$\frac{e_1 \implies n_1 \qquad e_2 \implies n_2}{e_1 + e_2 \implies n}$$

— Autres cas similaires.

Ainsi, nos deux présentations de la sémantique associent les mêmes expressions aux mêmes valeurs. La sémantique à petits pas en revanche nous dit des choses plus précises des évaluations qui n'aboutissent pas. Elle va permettre un énoncé satisfaisant des propriétés de sûreté, c'est-à-dire d'absence de problèmes d'évaluation, des programmes bien typés.

5.9 Sûreté du typage

Le slogan associé au typage était *well-typed programs do not go wrong*. Avec une sémantique à petits pas, ce slogan peut se traduire plus formellement par le fait suivant : l'évaluation d'un programme bien typé ne bloque jamais.

On traduit ce résultat par la conjonction des deux lemmes suivants.

— Lemme de ***progression*** : une expression bien typée n'est pas bloquée. Autrement dit, si une expression e bien typée n'est pas déjà une valeur, alors on peut encore effectuer au moins un pas de calcul à partir de e.

$$\textit{Si } \Gamma \vdash e : \tau \textit{ alors } v \textit{ est une valeur ou il existe } e' \textit{ telle que } e \to e'.$$

— Lemme de ***préservation*** : la réduction préserve les types. Si une expression e est cohérente, alors toute expression e' obtenue en calculant à partir de e est encore cohérente et de même type.

$$\textit{Si } \Gamma \vdash e : \tau \textit{ et } e \to e' \textit{ alors } \Gamma \vdash e' : \tau.$$

Historiquement, le lemme de préservation est appelé en anglais ***subject reduction*** (explication : e est le « sujet » du prédicat $\Gamma \vdash e : \tau$).

Ces deux lemmes rassemblés, et appliqués de manière itérée, promettent le comportement suivant de l'évaluation d'une expression e_1 cohérente et de type τ : si e_1 n'est pas déjà une valeur, alors elle se réduit en e_2, qui est encore cohérente et de type τ et qui donc, si elle n'est pas déjà une valeur, se réduit en e_3 cohérente et de type τ, etc.

$$e_1 : \tau \to e_2 : \tau \to e_3 : \tau \to \ldots$$

À l'extrémité droite de cette séquence, deux scénarios sont possibles : soit on aboutit à une valeur v (accessoirement : cohérente et de type τ), soit la réduction se poursuit indéfiniment. Il est en revanche impossible d'aboutir à un blocage.

Progression Si $\Gamma \vdash e : \tau$, alors e est une valeur, ou il existe e' telle que $e \to e'$. Reprenons les types simples du langage FUN définis par les règles suivantes

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \tau_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}$$

et démontrons le lemme par récurrence sur la dérivation de $\Gamma \vdash e : \tau$.

- Cas $\Gamma \vdash n : \texttt{int}$. Alors n est une valeur.
- Cas $\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau_1 \to \tau_2$. Alors de même $\texttt{fun } x \texttt{ -> } e$ est une valeur.
- Cas $\Gamma \vdash e_1\, e_2 : \tau_1$, avec $\Gamma \vdash e_1 : \tau_2 \to \tau_1$ et $\Gamma \vdash e_2 : \tau_2$. Les hypothèses de récurrence sur les deux prémisses nous donnent les deux disjonctions suivantes :
  1. e_1 est une valeur ou $e_1 \to e_1'$,
  2. e_2 est une valeur ou $e_2 \to e_2'$.

  On raisonne par cas sur ces disjonctions.
  - Si $e_1 \to e_1'$, alors $e_1\, e_2 \to e_1'\, e_2$: conclusion.
  - Sinon, e_1 est une valeur v_1.
    - Si $e_2 \to e_2'$, alors $v_1\, e_2 \to v_1\, e_2'$: conclusion.
    - Sinon, e_2 est une valeur v_2. Analysons la forme de la valeur v_1. Les deux formes possibles pour une valeur sont : n ou $\texttt{fun } x \texttt{ -> } e$. Or nous avons l'hypothèse de typage $\Gamma \vdash v_1 : \tau_2 \to \tau_1$, donc v_1 ne peut avoir que la forme $\texttt{fun } x \texttt{ -> } e$ (lemme de ***classification*** détaillé plus bas). Nous avons donc

      $$e_1\, e_2 = (\texttt{fun } x \texttt{ -> } e)\, v_2 \to e[x := v_2]$$

      ce qui conclut.
- Autres cas similaires.

Lemme de classification des valeurs typées. Soit v une valeur telle que $\Gamma \vdash v : \tau$. Alors :
- si $\tau = \texttt{int}$, alors v a la forme n,
- si $\tau = \tau_1 \to \tau_2$, alors v a la forme $\texttt{fun } x \texttt{ -> } e$.

Preuve par cas sur la dernière règle de la dérivation du jugement $\Gamma \vdash v : \tau$.

Préservation Si $\Gamma \vdash e : \tau$ et $e \to e'$ alors $\Gamma \vdash e' : \tau$. Preuve par récurrence sur la dérivation de $e \to e'$.

- Cas $n_1 \texttt{ + } n_2 \to n$ avec $n = n_1 + n_2$. De l'hypothèse $\Gamma \vdash n_1 \texttt{ + } n_2 : \texttt{int}$ on a nécessairement $\tau = \texttt{int}$ (lemme d'***inversion*** détaillé plus bas), et en outre $\Gamma \vdash n : \texttt{int}$.
- Cas $e_1 \texttt{ + } e_2 \to e_1' \texttt{ + } e_2$ avec $e_1 \to e_1'$. La prémisse nous donne l'hypothèse de récurrence « si $\Gamma \vdash e_1 : \tau'$, alors $\Gamma \vdash e_1' : \tau'$ ».
  De l'hypothèse $\Gamma \vdash e_1 \texttt{ + } e_2 : \tau$ on a nécessairement $\tau = \texttt{int}$, $\Gamma \vdash e_1 : \texttt{int}$ et $\Gamma \vdash e_2 : \texttt{int}$ (lemme d'inversion). Donc par hypothèse de récurrence $\Gamma \vdash e_1' : \texttt{int}$, dont on déduit encore la dérivation

  $$\frac{\Gamma \vdash e_1' : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}}$$

- Cas $(\texttt{fun } x \texttt{ in } e)\, v \to e[x := n]$. *Note : pas de prémisse à cette règle de réduction, et donc pas d'hypothèse de récurrence non plus.*
  De l'hypothèse $\Gamma \vdash (\texttt{fun } x \texttt{ in } e)\, v : \tau$ il existe τ' tel que $\Gamma \vdash \texttt{fun } x \texttt{ in } e : \tau' \to \tau$ et $\Gamma \vdash v : \tau'$ (lemme d'inversion) et de $\Gamma \vdash \texttt{fun } x \texttt{ in } e : \tau' \to \tau$ on a encore $\Gamma, x : \tau' \vdash e : \tau$ (lemme d'inversion toujours).
  On a donc d'une part $\Gamma, x : \tau' \vdash e : \tau$ et d'autre part $\Gamma \vdash v : \tau'$, ce dont on peut déduire que $\Gamma \vdash e[x := v] : \tau$, par le lemme de ***substitution*** détaillé ci-dessous.
- Autres cas similaires.

Lemme d'inversion.

- Si $\Gamma \vdash e_1 \texttt{ + } e_2 \texttt{ : } \tau$ alors $\tau = \texttt{int}, \Gamma \vdash e_1 \texttt{ : int}$ et $\Gamma \vdash e_2 \texttt{ : int}$.
- Si $\Gamma \vdash e_1\, e_2 \texttt{ : } \tau$ alors il existe τ' tel que $\Gamma \vdash e_1 \texttt{ : } \tau' \to \tau$ et $\Gamma \vdash e_2 \texttt{ : } \tau'$.
- Si $\Gamma \vdash \texttt{fun } x \texttt{ -> } e \texttt{ : } \tau$ alors il existe τ_1 et τ_2 tels que $\tau = \tau_1 \to \tau_2$ et $\Gamma, x \texttt{ : } \tau_1 \vdash e \texttt{ : } \tau_2$.

Preuve par cas sur la dernière règle de la dérivation de typage.

Lemme de substitution : remplacer une variable typée par une expression du même type préserve le typage.

$$\text{Si } \Gamma, x \texttt{ : } \tau' \vdash e \texttt{ : } \tau \text{ et } \Gamma \vdash e' \texttt{ : } \tau' \text{ alors } \Gamma \vdash e[x := e'] \texttt{ : } \tau.$$

Preuve par récurrence sur la dérivation de typage $\Gamma, x \texttt{ : } \tau' \vdash e \texttt{ : } \tau$.

Théorème de sûreté du typage Des lemmes de progression et de préservation du typage, on déduit l'énoncé suivant.

$$\textit{Si } \Gamma \vdash e \texttt{ : } \tau \textit{ et } e \to^* e' \textit{ avec } e' \textit{ irréductible, alors } e' \textit{ est une valeur.}$$

La preuve est par récurrence sur la longueur de la séquence de réduction $e \to^* e'$.

Bilan : la propriété de sûreté du typage est un lien entre une propriété statique d'un programme (sa cohérence du point de vue des types) et une propriété dynamique de ce programme (l'absence de blocage à l'exécution). Il reste possible que le programme boucle, ou diverge. De manière générale, un langage de programmation doté d'une discipline de types stricte va permettre une détection précoce de nombreuses erreurs (à la compilation), et donc éviter que certains bugs n'apparaissent que plus tard (à l'exécution).

6 Compilation

L'analyse préalable du programme source est terminée. Le vrai travail va pouvoir commencer.

6.1 *Back-end*

Les premières étapes de la compilation, sur lesquelles nous nous sommes concentrés jusque là, sont des étapes d'***analyse***, visant à extraire du sens du programme source reçu en entrée. Partant d'un programme source écrit dans un certain langage et donné sous la forme d'un fichier texte, nous avions donc :

1. une analyse *lexicale*, extrayant du texte source une séquence de lexèmes,
2. une analyse *grammaticale*, transformant cette séquence en un arbre de syntaxe, et
3. une analyse *sémantique*, vérifiant la cohérence du programme que l'on s'apprête à compiler.

Chacune de ces étapes est susceptible de s'interrompre et de renvoyer un message d'erreur à l'utilisateur, lorsque l'on détecte un problème dans le programme à compiler. Erreur lexicale par exemple si le texte source contient des symboles qui n'appartiennent pas au langage, erreur grammaticale par exemple si la forme du programme source est mauvaise, ou erreur sémantique si le programme source présente des incohérences de type. Toutes ces erreurs signalent un programme invalide qu'il est impossible (ou inutile) de compiler et demandent à l'utilisateur de corriger son programme avant de le soumettre à nouveau.

Phase de synthèse Après cette série d'analyses, lorsque qu'elles passent toutes avec succès, vient une phase de ***synthèse***. Cette dernière phase vise à produire un programme cible équivalent au programme source, à nouveau sous la forme d'un fichier écrit dans un certain langage cible. Cette phase de synthèse, ou ***back-end***, du compilateur prend comme point de départ l'arbre de syntaxe abstraite du programme source, et enchaîne des phases d'optimisation et de traduction visant à produire un programme efficace écrit dans un langage cible. Cette étape n'est plus censée échouer : toute erreur dans le programme source susceptible de faire échouer la compilation doit avoir été remontée à l'utilisateur lors des phases d'analyse.

Le langage cible peut être par exemple :

— un autre langage de haut niveau, par exemple C ou javascript, et on pourra alors utiliser l'infrastructure de ce langage pour exécuter le programme obtenu,
— du ***bytecode***, c'est-à-dire du code bas niveau pour une machine virtuelle, par exemple les machines virtuelles de java, caml ou python,
— un langage ***assembleur***, pour exécution directe sur une architecture matérielle donnée, par exemple Intel x86, ARM ou MIPS (après assemblage et édition de liens).

La phase de synthèse comprend généralement plusieurs étapes, voire plusieurs dizaines d'étapes dans un compilateur industriel, allant progressivement du langage source vers le langage cible, en passant par plusieurs langages intermédiaires. On appelle ***représentations intermédiaires*** toutes les représentations à l'intérieur du compilateur de ces langages intermédiaires par lesquels va passer la traduction. Les représentations intermédiaires ont des formes variées, couvrant un spectre avec à une extrémité des arbres de syntaxe décrivant un programme structuré dans un langage encore d'assez haut niveau, et à l'autre extrémité des séquences d'instructions de bas niveau de type assembleur ou *bytecode*. Entre les deux, on peut trouver des représentations plus exotiques, par exemple basées sur des graphes.

Une petite optimisation Une étape d'*optimisation* d'un compilateur travaille sur un programme exprimé dans l'une des représentations intermédiaires, et produit un programme équivalent mais « plus efficace ». L'optimisation elle-même est généralement la combinaison :

— d'une analyse, pour détecter ce qui peut être amélioré tout en respectant la sémantique du programme d'origine, et
— de la transformation elle-même, pour produire un programme résultat.

Le programme résultat peut être exprimé dans la même représentation intermédiaire que le programme d'origine.

Le compromis fondamental est de dépenser *une* fois du temps lors de la compilation pour en gagner *n* fois lors de l'exécution du programme. Les optimisations sont donc d'autant plus rentables qu'elles concernent des programmes ou des fragments de code destinés à être exécutés de nombreuses fois.

Une optimisation particulièrement simple consiste à simplifier les expressions constantes. En effet : ce qu'on peut calculer dès maintenant ne sera plus à calculer à l'exécution. En prenant la syntaxe abstraite du langage IMP par exemple, une expression `Add(Cst 4, Cst 7)` peut être simplifiée en l'expression `Cst 11`, et ce principe peut être appliqué récursivement pour simplifier autant que possible une expression complexe.

Un fragment d'une telle fonction de simplification pour IMP pourrait ainsi être :

```
let rec simplify = function
  | Add(e1, e2) ->
     let e1' = simplify e1 in
     let e2' = simplify e2 in
     begin match e1', e2' with
       | Cst n1, Cst n2 -> Cst (n1 + n2)
       | _, _ -> Add(e1', e2')
     end
```

Exercice : compléter cette fonction pour simplifier autant de choses que possible tout en préservant la sémantique du programme source.

Pour aller plus loin, on pourrait aussi imaginer propager les valeurs des variables lorsqu'elles sont connues. Ainsi dans un code comme

```
x = 1;
y = x+1;
```

on peut savoir à la deuxième ligne que la valeur de `x` est `1`, et en déduire une simplification de la deuxième affectation en `y = 2;`. Cependant, les simplifications de ce type peuvent demander des analyses beaucoup plus fines. Peut-on de même simplifier l'affectation à `y` dans un programme de la forme suivante ?

```
x = 1;
while (...) {
  y = x+1;
  x = 1/x;
}
```

Nous n'irons pas jusque là dans ce cours (rendez-vous l'année prochaine pour de vraies optimisations).

Fil conducteur Dans ce cours, nous allons dans un premier temps prendre comme langage source le langage IMP, et développer un schéma simple de compilation vers le langage assembleur MIPS, en trois étapes.

1. Nous venons de voir une mini-étape d'optimisation, sur l'arbre de syntaxe IMP (première représentation)
2. Traduction vers le code d'une machine virtuelle de bas niveau (deuxième représentation).
3. Traduction du code obtenu à l'étape précédente vers du code assembleur MIPS (troisième représentation).

À la fin du chapitre nous ajouterons de nouvelles phases préalables, partant d'un langage source d'un peu plus haut niveau inspiré de C, qui sera traduit vers IMP.

6.2 Représentation intermédiaire bas niveau

Voici un exemple de code bas niveau, exprimé dans ce qui sera notre deuxième représentation intermédiaire pour notre compilateur IMP.

```
function fact {
  nb_params: 1
  nb_locals: 0
  start: fact_4

  fact_4:
    cst 2; push; get param[0]; lt; cjump fact_2, fact_3

  fact_3:
```

```
    cst -1; push; get param[0]; add; push; call fact;
    push; get param[0]; mul; return

  fact_2:
    cst 1; return
}
```

Nous allons bientôt détailler cette représentation, mais vous pouvez déjà déceler que l'ensemble de ce code exemple représente une fonction, dont le corps est séparé en trois blocs, chacun introduit par une étiquette de la forme `fact_*`. Chaque bloc est ensuite constitué d'une séquence d'instructions élémentaires, avec des opérations arithmétiques élémentaires (`cst`, `add`, `mul`), des accès à des variables (`get`), des appels de fonctions (`call`, `return`), des sauts (`cjump`), des manipulations d'une pile (`push`)...

Cette représentation intermédiaire, que nous appellerons LLIR, peut être vue comme un code pour une ***machine virtuelle***, c'est-à-dire un ordinateur fictif, dont l'architecture serait centrée autour de deux éléments :

— un registre de travail, contenant la valeur de l'expression courante, et
— une pile, stockant les valeurs intermédiaires « en attente ».

On pourra représenter un état de la machine virtuelle par un tableau

v_1	v_2	…	v_n	r

où les v_i sont les valeurs intermédiaires stockées sur la pile, avec v_1 la plus ancienne et v_n la plus récente, et où r est la valeur contenue dans la registre.

Un programme LLIR est constitué d'un ensemble de déclarations de variables globales, et d'un ensemble de définitions de fonctions.

Arithmétique Cette architecture est facilement exploitable pour réaliser des calculs arithmétiques. On peut par exemple produire la valeur de l'expression `(1+2)+(4+8)` avec la séquence d'actions suivante.

1. Partir de l'état « vide ». | . |
2. Calculer la valeur de `1+2`.
   (a) Placer la valeur `1` dans le registre. | 1 |
   (b) Sauvegarder le registre sur la pile. | 1 | 1 |
   (c) Placer la valeur `2` dans le registre. | 1 | 2 |
   (d) Ajouter au registre la dernière valeur de la pile. | 3 |
3. Sauvegarder la valeur de `1+2` sur la pile. | 3 | 3 |
4. Calculer la valeur de `4+8`.
   (a) Placer la valeur `4` dans le registre. | 3 | 4 |
   (b) Sauvegarder le registre sur la pile. | 3 | 4 | 4 |
   (c) Placer la valeur `8` dans le registre. | 3 | 4 | 8 |
   (d) Ajouter au registre la dernière valeur de la pile. | 3 | 12 |
5. Ajouter au registre la dernière valeur de la pile. | 15 |

Notre représentation intermédiaire fournit donc des instructions élémentaires pour chacune des opérations effectuées dans cet exemple.

— L'instruction `cst` n place dans le registre une valeur numérique n.

p	r	→	p	n

— L'instruction `push` sauvegare sur la pile la valeur courante du registre.

p	r	→	p	r	r

— Les instructions `add`, `mul`, `lt`, ... appliquent une opération aux valeurs contenues dans le registre et au sommet de la pile.

p	v	r	→	p	$r \star v$

Notez que ces instructions consomment la valeur présente au sommet de la pile, et que leur comportement n'est pas défini dans le cas où la pile serait vide.

Dans ce cadre, *compiler* une expression c'est produire un fragment de code LLIR qui calcule la valeur de cette expression et laisse le résultat dans le registre. Ainsi on compile l'expression e_1 `+` e_2 en produisant l'enchaînement suivant :

1. commencer avec le code évaluant la sous-expression e_1,
2. ajouter une instruction `push` pour sauvergarder le résultat sur la pile,
3. poursuivre avec le code évaluant la sous-expression e_2,
4. conclure avec une instruction `add`.

Remarquez que l'instruction `add` est placée un peu après une instruction `push`, et s'exécutera donc bien avec une pile non vide. Ou du moins, c'est le cas si l'on suppose que la séquence d'instructions relative à e_2 ne fait pas n'importe quoi. Nous en reparlerons dans la section suivante.

Variables On a dans LLIR trois catégories de variables.

— Les variables globales, désignées par un identifiant. Par exemple : `x`.
— Les variables locales, qui sont numérotées, avec la notation `local[`k`]`.
— Dans le cadre d'un appel de fonction, les paramètres de cet appel, qui sont numérotés également, avec la forme `param[`k`]`.

Ces trois catégories de variables sont manipulées de la même manière dans le code, à l'aide de deux instructions.

— L'instruction d'accès en lecture `get` v place la valeur de la variable v dans le registre.
— L'instruction d'accès en écriture `set` v définit la nouvelle valeur de la variable v avec la valeur courante du registre.

Dans le langage IMP, ces trois sortes de variables existent mais elles sont toutes désignées exactement de la même façon : par des identifiants. La seule distinction entre les trois versions est la manière dont elles sont déclarées :

— les variables globales sont déclarées en tête du fichier,
— les variables locales sont déclarées en tête du corps d'une fonction,
— les paramètres d'une fonction sont déclarés dans la signature de la fonction elle-même.

Pour permettre la traduction des identifiants IMP vers des variables LLIR on se base sur une ***table des symboles***, c'est-à-dire une table associant des informations à chaque identifiant du programme source IMP. Les informations contenues dans une table des symboles peuvent être de différentes natures. Ici on se contente de renseigner la manière dont chaque variable est déclarée : globale, locale ou comme paramètre de fonction. Dans les deux derniers cas, on ajoute un numéro localement unique, en commençant par exemple à 1 la première variable locale et pour le premier paramètre d'une fonction. Cette table peut être déduite simplement des informations contenues dans l'AST.

La compilation des accès aux variables peut alors suivre les schémas suivants.

— On traduit une expression IMP `x` par une simple instruction `get` v, où v est la désignation de la variable `x` au format LLIR, obtenue en consultant la table des symboles pour `x`.
— On traduit une instruction IMP `x = e;` par l'enchaînement suivant :
  1. d'abord le code évaluant `e`,
  2. puis l'instruction `set` v, avec v la notation LLIR associée à l'identifiant `x`.

Contrôle Le code d'une fonction LLIR est donné par un ensemble de ***blocs***, qui forment l'unité la plus fine d'organisation du code. Un bloc est une séquence d'instructions qui :

— a un *nom unique*, donné par une étiquette,
— est *linéaire*, c'est-à-dire qui ne contient aucun branchement,
— *termine par un saut* vers le code à exécuter ensuite, avec éventuellement plusieurs suites possibles, ou par une instruction `return`.

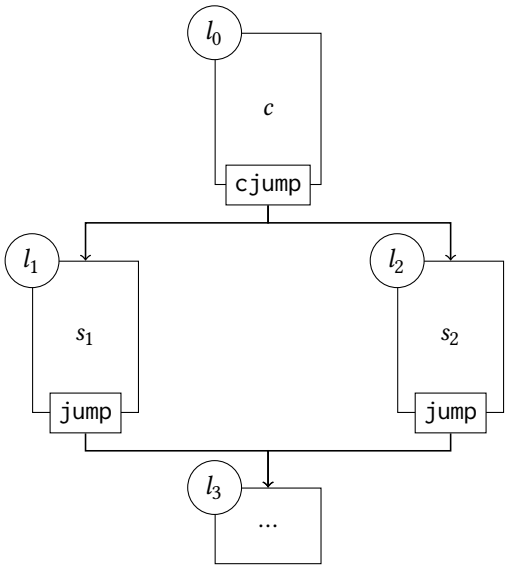
On a deux instructions de saut possibles pour conclure un bloc, en plus de `return`.

— Le saut inconditionnel `jump` l désigne comme bloc suivant le bloc d'étiquette l.
— Le saut conditionnel `cjump` l_1 l_2 désigne comme bloc suivant l'un des blocs d'étiquette l_1 ou l_2. Le choix ira vers l_1 si la valeur courante du registre est non nulle, ou vers l_2 sinon.

Les structures de contrôle du langage IMP sont traduites par des combinaisons de blocs LLIR. On peut les représenter à l'aide de graphes dont les nœuds représentent des blocs d'instructions et les arcs les instructions de saut.

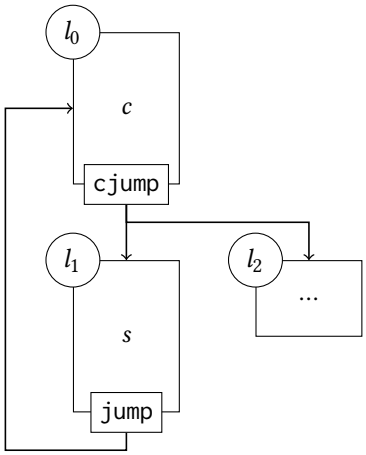
— Une instruction de branchement conditionnel `if (`c`) { `s_1` } else { `s_2` }` introduit un graphe comportant un branchement au niveau du test, et un point de jonction

auquel on saute après l'exécution de l'une ou l'autre des séquences s_1 ou s_2.



Selon le contenu des séquences s_1 et s_2, les parties correspondantes peuvent elles-mêmes être composées de plusieurs blocs. Le bloc de jonction peut être directement le premier bloc de l'instruction suivant le branchement.

— Une boucle `while (`c`) { `s` }` donne un graphe contenant un cycle, où après l'exécution de la séquence s on saute à nouveau au bloc effectuant le test de la condition.



Le bloc de sortie peut être directement le premier bloc de l'instruction suivant la boucle.

Fonctions Une fonction LLIR est définie par :

— son nom,
— son nombre de paramètres,
— son nombre de variables locales,
— un ensemble de blocs de code,
— l'identification du bloc de départ.

Notez que les blocs n'ont pas d'ordre particulier : l'identification du bloc de départ suffit à commencer l'exécution, puis les sauts à la fin de chaque bloc donneront la succession à utiliser. Les identifiants des paramètres et des variables locales ne sont pas non plus utiles : on accède à ces variables uniquement par leur numéro, et connaître leur nombre suffira donc.

On a deux instructions pour la gestion des appels de fonction.

— L'instruction `return` déjà mentionnée permet d'arrêter l'exécution de la fonction. La valeur renvoyée par la fonction est celle qui se trouve dans le registre.
— L'instruction `call` f déclenche un appel à la fonction de nom f. Les paramètres sont pris sur la pile et en sont retirés. Le résultat renvoyé est placé dans le registre.

Chaque appel de fonction s'exécute en isolation des autres, avec sa propre pile, son propre tableau de paramètres et son propre tableau de variables locales. Seules les variables globales sont partagées.

Un appel de fonction `f(`e_1`, ..., `e_n`)` du langage IMP est compilé en l'enchaînement suivant :

— évaluer e_n et empiler le résultat,
— ...
— évaluer e_1 et empiler le résultat,
— conclure avec `call f` pour réaliser l'appel lui-même.

Avec tout ceci en main, vous pouvez relire et interpréter le code de la fonction factorielle en LLIR vu plus haut.

6.3 Raisonner sur les transformations

On a décrit un compilateur comme un programme qui :
— prend en entrée un programme source P_1, et qui
— produit un programme cible P_2 *équivalent.*

Par « équivalent » on entend informellement « qui a le même comportement en toutes circonstances », ou autrement dit « qui a le même comportement pour chacune des entrées possibles ». Plus précisément, voici ce qu'on attend de l'exécution du programme P_2 sur des entrées e_1, ..., e_n en fonction du comportement de P_1 sur ces mêmes entrées.

— Si P_1 est défini et s'exécute sans erreur, alors P_2 doit produire le même résultat et les mêmes effets (et ne pas faire d'erreur). Petite subtilité : si P_1 n'est pas déterministe, alors on peut demander à P_2 de produire l'un des ensembles résultat/effets possibles pour P_1.
— Si P_1 produit une erreur, alors P_2 doit produire une erreur.
— Si le comportement de P_1 est indéfini, alors P_2 peut avoir un comportement arbitraire (il peut produire un résultat, ou produire une erreur, ou être lui aussi indéfini).

Ce contrat est relatif aux sémantiques du langage source et du langage cible.

Cette notion de ***correction*** d'un compilateur peut s'appliquer à toute transformation de programme, et en particulier à chacune des étapes d'optimisation et de traduction de la phase de synthèse. Autrement dit, on peut justifier qu'un compilateur donné est correct en justifiant que chacune des transformations qu'il effectue préserve bien le comportement des programmes.

Pour énoncer la correction d'une transformation de programme, il faut au préalable :
— une sémantique du langage source,
— une sémantique du langage cible (si différent du langage source),
— une fonction de traduction.

Ainsi, démontrer la correction d'un compilateur complet en analysant les étapes de transformation l'une après l'autre demande d'avoir, en plus d'une sémantique pour le langage source : une sémantique du langage cible, et une sémantique de chaque représentation intermédiaire !

Cas : optimisation des expressions constantes On a mentionné une optimisation de simplification des expressions constantes pour le langage IMP. Il s'agit d'une transformation dont le langage source et le langage cible sont tous les deux IMP. On justifie donc sa correction en fonction d'une sémantique de IMP.

Supposons par exemple les règles simples suivantes pour un fragment des expressions du langage. On utilise les notations caml de la syntaxe abstraite pour éviter les confusions entre syntaxe et sémantique.

$$\frac{}{\texttt{Cst}\ n, \rho \implies n} \qquad \frac{}{\texttt{Var}\ x, \rho \implies \rho(x)} \qquad \frac{e_1, \rho \implies n_1 \qquad e_2, \rho \implies n_2}{\texttt{Add}(e_1, e_2), \rho \implies n_1 + n_2}$$

La simplification des expressions constantes réalise une fonction f satisfaisant les équations suivantes.

$$\begin{aligned} f(\texttt{Cst}\ n) &= \texttt{Cst}\ n \\ f(\texttt{Var}\ x) &= \texttt{Var}\ x \\ f(\texttt{Add}(e_1, e_2)) &= \begin{cases} \texttt{Cst}\ (n_1 + n_2) & \text{si } f(e_1) = n_1 \text{ et } f(e_2) = n_2 \\ \texttt{Add}(f(e_1), f(e_2)) & \text{sinon} \end{cases} \end{aligned}$$

On se donne l'énoncé de correction suivant :

$$\text{Pour tous } e, \rho, \text{ si } e, \rho \implies n \text{ alors } f(e), \rho \implies n.$$

On le démontre par récurrence sur la dérivation de $e, \rho \implies n$.

- Cas $n, \rho \implies n$ et $x, \rho \implies \rho(x)$. Dans ces cas la fonction f est l'identité, et la conclusion est immédiate.
- Cas $\texttt{Add}(e_1, e_2) \implies n_1 + n_2$ avec $e_1, \rho \implies n_1$ et $e_2, \rho \implies n_2$. Les deux prémisses nous donnent les hypothèses de récurrence $f(e_1), \rho \implies n_1$ et $f(e_2), \rho \implies n_2$.
  La définition de $f(\texttt{Add}(e_1, e_2))$ dépend des formes de $f(e_1)$ et de $f(e_2)$.
  - Si $f(e_1) = n_1'$ et $f(e_2) = n_2'$, alors $f(\texttt{Add}(e_1, e_2)) = \texttt{Cst}(n_1' + n_2')$. Dans ce cas, l'hypothèse de récurrence $f(e_1), \rho \implies n_1$ s'écrit en outre $n_1', \rho \implies n_1$, ce qui n'est possible qu'avec $n_1' = n_1$. De même on a $n_2' = n_2$, et donc $f(\texttt{Add}(e_1, e_2)) = \texttt{Cst}(n_1' + n_2') = \texttt{Cst}(n_1 + n_2)$.
  - Sinon $f(\texttt{Add}(e_1, e_2)) = \texttt{Add}(f(e_1), f(e_2))$, et à l'aide des hypothèse de récurrence on peut déduire

$$\frac{f(e_1), \rho \implies n_1 \qquad f(e_2), \rho \implies n_2}{\texttt{Add}(f(e_1), f(e_2)), \rho \implies n_1 + n_2}$$

Cas : traduction des expressions arithmétiques Considérons maintenant la traduction des expressions arithmétiques du langage IMP en en séquences d'instructions LLIR. La fonction de traduction satisfait les équations suivantes.

$$\begin{array}{rcl} f(\texttt{Cst}\ n) & = & \texttt{cst}\ n \\ f(\texttt{Var}\ x) & = & \begin{cases} \texttt{get}\ x & \text{si } x \text{ variable globale} \\ \texttt{get local[}k\texttt{]} & \text{si } x \text{ est la } k\text{ème variable locale} \\ \texttt{get param[}k\texttt{]} & \text{si } x \text{ est le } k\text{ème paramètre} \end{cases} \\ f(\texttt{Add}(e_1, e_2)) & = & f(e_1);\ \texttt{push};\ f(e_2);\ \texttt{add} \end{array}$$

La correction de cette transformation exprime un lien entre l'évaluation de l'expression IMP selon la sémantique déjà connue, et l'exécution des instructions LLIR. La sémantique de ces dernières doit encore être formalisée !

Les instructions LLIR sont exécutées par une machine virtuelle utilisant un registre, une pile et une mémoire. Chaque instruction exécutée modifie certains de ces éléments, qui forment ensemble l'***état*** de la machine. Ainsi pour une séquence $i_1\,;\ i_2\,;\ \ldots i_n$ on a les évolutions successives

$$S_0 \xrightarrow{i_1} S_1 \xrightarrow{i_2} S_2 \ldots \xrightarrow{i_n} S_n$$

où chaque état S_k est décrit par un triplet

$$[\rho \mid r \mid \pi]$$

où

- ρ est un environnement, c'est-à-dire une représentation de la mémoire sous la forme d'une fonction associant des valeurs aux variables,
- r est la valeur courante du registre,
- π est une liste de valeurs représentant la pile.

On formalise la sémantique en définissant les transitions $S \xrightarrow{i} S'$ d'un état S à un état S' sous l'effet d'une instruction i.

état de départ	instruction	état d'arrivée
$[\rho \mid r \mid \pi]$	$\texttt{cst}\ n$	$[\rho \mid n \mid \pi]$
$[\rho \mid r \mid \pi]$	$\texttt{get}\ v$	$[\rho \mid \rho(v) \mid \pi]$
$[\rho \mid r \mid \pi]$	push	$[\rho \mid r \mid r::\pi]$
$[\rho \mid r \mid n::\pi]$	add	$[\rho \mid n + r \mid \pi]$

Note : par v on désigne ici l'une quelconque des formes x, $\texttt{local[}k\texttt{]}$ ou $\texttt{param[}k\texttt{]}$.

L'énoncé de correction de la traduction des expressions exprime un lien entre la relation d'évaluation $e, \rho \implies n$ et l'évolution de l'état de la machine virtuelle LLIR sous l'effet de l'exécution d'une séquence d'instructions.

$$\text{Pour tous } e, \rho, r \text{ et } \pi, \text{ si } e, \rho \implies n \text{ alors } [\rho \mid r \mid \pi], f(e) \to^* [\rho \mid n \mid \pi].$$

Notez dans cet énoncé que le contenu initial du registre et de la pile n'ont aucune influence. La pile doit en revanche, à la fin de l'exécution, avoir retrouvé son état initial.

On démontre cet énoncé par récurrence sur la dérivation de $e, \rho \implies n$.

- Cas $n, \rho \implies n$. Alors $f(n) = \texttt{cst}\ n$ et la règle de transition correspondante donne directement le résultat souhaité : $[\rho \mid r \mid \pi], \texttt{cst}\ n \to [\rho \mid n \mid \pi]$.
- Cas $x, \rho \implies \rho(x)$ similaire.
- Cas $\texttt{Add}(e_1, e_2), \rho \implies n_1 + n_2$ avec $e_1, \rho \implies n_1$ et $e_2, \rho \implies n_2$. Les deux prémisses nous donnent les hypothèses de récurrence « $\forall r, \pi,\ [\sigma \mid r \mid \pi], f(e_1) \to^* [\sigma \mid n_1 \mid \pi]$ » et « $\forall r, \pi,\ [\sigma \mid r \mid \pi], f(e_2) \to^* [\sigma \mid n_2 \mid \pi]$ ».
  On rappelle que $f(\texttt{Add}(e_1, e_2)) = f(e_1);\ \texttt{push};\ f(e_2);\ \texttt{add}$, dont on déduit la séquence d'exécution suivante.

$$
\begin{array}{llll}
[\sigma \mid r \mid \pi] & , f(e_1) & \to^* & [\sigma \mid n_1 \mid \pi] \\
[\sigma \mid n_1 \mid \pi] & , \texttt{push} & \to & [\sigma \mid n_1 \mid n_1 :: \pi] \\
[\sigma \mid n_1 \mid n_1 :: \pi] & , f(e_1) & \to^* & [\sigma \mid n_2 \mid n_1 :: \pi] \\
[\sigma \mid n_2 \mid n_1 :: \pi] & , \texttt{add} & \to & [\sigma \mid n_1 + n_2 \mid \pi]
\end{array}
\qquad \begin{array}{l} \\ \\ \\ \\ \end{array}
\begin{array}{l} \text{par hyp. de récurrence} \\ \\ \text{par hyp. de récurrence} \\ \\ \end{array}
$$

  À la fin de la séquence, on a bien obtenu la valeur $n_1 + n_2$ dans le registre, et la pile a retrouvé son état d'origine π. Notez que la deuxième application de l'hypothèse de récurrence est faite dans un état où la pile a été étendue par rapport à son état initial. C'est possible grâce à la quantification universelle sur π dans l'hypothèse de récurrence.

6.4 Traduction IMP vers LLIR

Dans cette section, nous allons présenter le code complet d'une traduction de IMP vers LLIR.

Syntaxe abstraite IMP Un programme IMP contient des variables globales et des définitions de fonctions.

```
type prog = {
  globals: string list;
  functions: function_def list;
}
```

Chaque définition de fonction comporte un nom, une liste de paramètres formels, une liste de variables locales et un code.

```
type function_def = {
  name: string;
  params: string list;
  locals: string list;
  code: seq;
}
```

Le code est une séquence d'instructions comportant notamment des opérations d'affectation et les instructions de contrôle classiques. On y ajoute une primitive `putchar` d'affichage d'un caractère, une instruction `return` pour la fin d'exécution des fonctions, et la possibilité d'utiliser une expression à la place d'une instruction.

```
type instr =
  | Set of string * expr     (* x = e; *)
  | If of expr * seq * seq   (* if (e) { s1 } else { s2 } *)
  | While of expr * seq      (* while (e) { s } *)
  | Putchar of expr          (* putchar(e); *)
  | Return of expr           (* return(e); *)
  | Expr of expr             (* e; *)
and seq = instr list         (* i1; i2; ...; iN *)
```

Les expressions sont formées avec quelques opérations arithmétiques déjà manipulées, des variables, et des appels de fonctions.

```
type expr =
  | Cst of int                  (* 17 *)
```

```
  | Add of expr * expr          (* e1 + e2 *)
  | Mul of expr * expr          (* e1 * e2 *)
  | Lt of expr * expr           (* e1 < e2 *)
  | Get of string               (* x *)
  | Call of string * expr list  (* f(e1, ..., eN) *)
```

Représentation intermédiaire LLIR Un programme au format LLIR est composé, comme au format IMP, d'un ensemble de variables globales et d'un ensemble de fonctions. Les définitions de fonction en revanche évoluent. D'une part, les variables locales et les paramètres formels des fonctions sont identifiés en LLIR par leur numéro. Il n'est donc plus nécessaire de retenir leurs identifiants, leur nombre suffit. D'autre part, le code d'une fonction LLIR est donné par un ensemble de blocs d'instructions, chacun associé à une étiquette. Il n'y a plus ici de notion d'ordre entre les différents blocs. À la place, chaque bloc termine par une instruction de saut permettant d'identifier le bloc à exécuter ensuite. On peut voir ce code comme un *graphe* dont les sommets sont des blocs d'instructions et dont les arêtes correspondent aux successions possibles entre blocs. On représente cet ensemble par une table de hachage où les clés sont les étiquettes et les valeurs sont les blocs associés.

```
type function_def = {
  name: string;
  nb_params: int;
  nb_locals: int;
  code: (string, seq) Hashtbl.t;
  start: string;
}

type prog = {
  globals: string list;
  functions: function_def list;
}
```

On introduit un type commun pour représenter les différentes sortes de variables des programmes LLIR (variables globales, paramètres formels de fonctions, variables locales de fonctions).

```
type var =
  | Global of string
  | Param of int
  | Local of int
```

Les instructions enfin sont maintenant restreintes à des opérations élémentaires, agissant sur le registre, sur la pile ou sur la mémoire. On y retrouve quelques opérations élémentaires d'arithmétique,

```
type instr =
  | Cst of int  (* r <- n *)
  | Push        (* push r *)
  | Add         (* r <- r + pop *)
  | Mul         (* r <- r * pop *)
  | Lt          (* r <- (pop < r)?1:0 *)
```

des instructions de lecture ou d'écriture des variables,

```
  | Get of var  (* r <- var *)
  | Set of var  (* var <- r *)
```

ou des instructions de saut et de saut conditionnel.

```
  | Jump of string            (* exec l *)
  | CJump of string * string  (* exec r?l1:l2 *)
```

L'instruction d'appel de fonction prend les paramètres de la fonction sur la pile, et le résultat est renvoyé via le registre.

```
  | Call of string  (* r <- f(pop, ..., pop) *)
  | Return          (* return r *)
```

On conserve une opération primitive d'affichage (prenant en paramètre le registre), ainsi qu'une notion de séquence.

```
    | Putchar     (* print r *)
type seq = instr list
```

Traduction de IMP vers LLIR Pour traduire un programme IMP en programme LLIR, on traduit indépendamment chacune des fonctions à l'aide d'une fonction `tr_fdef` définie ci-après.

```
let tr_prog prog = {
  Llir.globals = Imp.(prog.globals);
  Llir.functions = List.map tr_fdef Imp.(prog.functions);
}
```

La principale tâche de la fonction caml traduisant les fonctions IMP en fonctions LLIR consiste à produire le graphe des blocs d'instructions LLIR. En l'occurrence, on produira au moins un bloc LLIR pour chaque instruction IMP. On définit pour cela une table de hachage pour les blocs et deux fonctions auxiliaires : `new_name` produit de nouveaux noms de blocs (préfixés par le nom de la fonction) et `new_block` crée un nouveau bloc à partir d'une séquence d'instructions LLIR.

```
let tr_fdef fdef =
  let code = Hashtbl.create 32 in
  let new_name =
    let f = Imp.(fdef.name) in
    let cpt = ref 0 in
    fun () -> incr cpt; Printf.sprintf "%s_%i" f !cpt
  in
  let new_block b =
    let block_name = new_name() in
    Hashtbl.add code block_name b;
    block_name
  in

  ...
```

Notez que `new_block` génère à la volée un nom pour le nouveau bloc `b`, et renvoie ce nom. En outre, on aura besoin de connaître ce nom pour former la séquence d'instructions du bloc « précédent » du programme, puisque cette séquence « précédente » devra terminer par une instruction de saut vers le bloc `b`. Autrement dit, les blocs LLIR correspondant à une séquence IMP vont être créés à rebours, en commençant par la dernière instruction IMP et en revenant progressivement à la première. Ainsi, la fonction `tr_instr` traduisant une instruction IMP en un bloc LLIR prend en paramètre supplémentaire le nom du bloc suivant. De même, la fonction `tr_seq` de traduction d'une séquence IMP prend en paramètre supplémentaire le nom du bloc suivant, et renvoie le nom du premier bloc. Pour traduire une séquence `i::s` on traduit donc d'abord sa queue `s`, après quoi on récupère le nom `next'` du premier bloc de la queue, qui sera désigné comme successeur du bloc de tête.

```
  let rec tr_seq s next = match s with
    | i::s -> let next' = tr_seq s next in
              new_block (tr_instr i next')
```

Notez que la fonction `tr_seq`, ainsi que toutes les fonctions restant à venir dans ce traducteur, sont définies comme des fonctions internes de `tr_fdef`. Elles ont ainsi bien accès à la table de hachage `code` et à la fonction `new_block`.

On complète la fonction `tr_seq` en traitant à part le cas d'une séquence d'une unique instruction et le cas d'une séquence vide, pour remplacer ce dernier cas par une instruction anodine.

```
    | [i] -> new_block (tr_instr i next)
    | []  -> new_block (tr_instr Imp.(Expr(Cst 0)) next)
  in
```

En poussant cette logique à l'échelle de la fonction IMP complète, le premier bloc créé va être le bloc final de la fonction, puis on progressera petit à petit en revenant vers le début du texte de la fonction. Le code principal de `tr_fdef` crée donc d'abord un bloc final avant de traduire le reste du code.

```
    let end_name = new_block [Llir.Cst 0; Llir.Return] in
    let start = tr_seq Imp.(fdef.code) end_name in
```

Ici, le bloc final correspond à une instruction `return(0);` ajoutée à la fin du programme, qui sera exécutée si aucune instruction `return` n'est rencontrée plus tôt. Une fois passé cet appel à `tr_seq`, la table de hachage `code` contient tous les blocs de code LLIR traduisant le code IMP de la fonction, et `start` désigne le bloc de tête. Il ne reste plus qu'à renseigner les champs de la structure représentant la fonction LLIR.

```
    {
      Llir.name = Imp.(fdef.name);
      Llir.nb_params = Imp.(List.length fdef.params);
      Llir.nb_locals = Imp.(List.length fdef.locals);
      Llir.code = code;
      Llir.start = start;
    }
```

On crée un nouveau bloc pour chaque instruction IMP traduite, voire plusieurs blocs pour les instructions impliquant des branchements. La fonction de traduction `tr_instr` prend en paramètre supplémentaire le nom du bloc suivant, et renvoie une séquence d'instructions LLIR destinée à former un nouveau bloc. La fonction `tr_instr` fait appel à une fonction `tr_expr` dédiée à la traduction des expressions. La fonction `tr_expr` prend en paramètre, outre une expression `e` à traduire, une liste `k` d'instructions LLIR à exécuter *après* l'évaluation de `e`. Le résultat d'un appel `tr_expr e k` est donc une liste d'instructions LLIR contenant d'abord les instructions évaluant `e`, puis la liste « de suite » `k`.

Ainsi par exemple, on traduit une instruction IMP `putchar(e);` précédant un bloc de nom `next` en faisant suivre les instructions d'évaluation de `e` de l'instruction LLIR `putchar` et de l'instruction de saut `jump next`.

```
    let rec tr_instr i next = match i with
      | Imp.Putchar(e) ->
        tr_expr e [Llir.Putchar; Llir.Jump next]
```

On suit la même logique pour les instructions d'affectation, d'évaluation d'une expression seule, ou de retour.

```
      | Imp.Set(x, e) ->
        tr_expr e [Llir.Set (convert_var x); Llir.Jump next]
      | Imp.Expr e ->
        tr_expr e [Llir.Jump next]
      | Imp.Return e ->
        tr_expr e [Llir.Return]
```

Les spécificités de ces nouveaux cas sont, pour l'instruction d'affectation l'utilisation d'une fonction auxiliaire traduisant une variable (définie plus bas), et pour l'instruction de retour le fait que l'on néglige l'étiquette `next`, qui n'a plus d'utilité.

Les instructions de contrôle apportent des comportements un peu plus riches, puisqu'il faut des graphes avec des branchements, voire des boucles. Dans le cas d'un branchement conditionnel `if (`c`) { `s_1` } else { `s_2` }`, les deux branches s_1 et s_2 sont traduites avec la même étiquette de suite `next`, qui est le point de jonction. Le bloc évaluant la condition c se conclut par un saut conditionnel, choisissant l'une des deux branches (rappel : `tr_seq` renvoie l'étiquette du bloc de tête de la séquence traduite).

```
      | Imp.If(c, s1, s2) ->
        let then_name = tr_seq s1 next in
        let else_name = tr_seq s2 next in
        tr_expr c [Llir.CJump(then_name, else_name)]
```

Dans le cas d'une boucle, on produit un bloc pour le test de la condition, et un ensemble de blocs pour le corps de la boucle. Pour résoudre la circularité dans le graphe à créer, on génère à la main le nom du bloc réalisant le test, puis on l'enregistre de même manuellement dans

la table de hachage (pour ce bloc, on n'utilise donc pas notre fonction auxiliaire `new_block`). Une fois ceci mis en place, on produit un bloc contenant une unique instruction de saut vers le bloc réalisant le test, puisque c'est par là qu'il faudra commencer.

```
    | Imp.While(c, s) ->
      let test_name = new_name() in
      let loop_name = tr_seq s test_name in
      Hashtbl.add code test_name (tr_expr c [Llir.CJump(loop_name, next)]);
      [Llir.Jump test_name]
  in
```

La fonction `tr_expr` traduit une expression IMP `e` en une séquence d'instructions LLIR. Cette fonction est récursive : pour traduire une expression `Add(`e_1`, `e_2`)` il faudra traduire les deux sous-expressions e_1 et e_2 et rassembler les deux séquences obtenues. En revanche, on veut éviter d'utiliser l'opérateur `@` de concaténation à cet endroit (il coûte trop cher). À la place on donne à la fonction `tr_expr` un paramètre supplémentaire : une liste `k` d'instructions LLIR qui doit être placée *après* la séquence obtenue en traduisant `e`. Cela revient à fournir en deuxième paramètre à `tr_expr` la liste avec laquelle le résultat de la traduction de `e` doit être « concaténée » (sans utiliser `@`)[9]. Dans des cas comme celui d'une constante ou d'une variable, où la traduction de `e` est réduite à une unique instruction, il suffit donc de placer cette instruction en tête de `k`.

```
  let rec tr_expr e k = match e with
    | Imp.Cst(n) -> Llir.Cst n :: k
    | Imp.Get(x) -> Llir.Get (convert_var x) :: k
```

Dans le cas d'un opérateur binaire la situation est un peu plus riche : nous avons deux appels récursifs à `tr_expr`, et il faut donner à chacun la suite `k` adaptée. Considérons le cas de `Add(`e_1, e_2`)`. Sa traduction était décrite par l'équation $f($`Add(`e_1, e_2`)`$) =$ `push;` $f(e_1)$`;` $f(e_2)$`;` `add`. Ainsi, la suite à donner à $f(e_2)$ est `add` et la suite à donner à $f(e_1)$ est `push;` $f(e_2)$`;` `add` (et dans un cas comme dans l'autre, on ajoute encore la suite globale `k`). On traduit donc d'abord l'expression e_2 en lui donnant la suite `add; k`, ce qui nous donne une liste `k'`, puis on obtient la séquence complète en incluant cette liste dans la suite à donner à la traduction de e_1.

```
    | Imp.Add(e1, e2) ->
      let k' = tr_expr e2 (Llir.Add :: k) in
      tr_expr e1 (LLir.Push :: k')
```

Les autres opérateurs binaires sont traités de même (ici sans prendre la peine de nommer la séquence intermédiaire `k'`).

```
    | Imp.Mul(e1, e2) ->
      tr_expr e1 (Llir.Push :: tr_expr e2 (Llir.Mul :: k))
    | Imp.Lt(e1, e2) ->
      tr_expr e1 (Llir.Push :: tr_expr e2 (Llir.Lt :: k))
```

Pour les appels de fonction on passe par une fonction auxiliaire `tr_args` pour produire une séquence d'instructions évaluant et plaçant sur la pile une liste d'arguments. Comme `tr_expr`, cette fonction prend en deuxième paramètre une séquence de suite.

```
    | Imp.Call(f, args) -> tr_args args (Llir.Call f :: k)

  and tr_args args k = match args with
    | [] -> k
    | a::args -> tr_args args (tr_expr a (Llir.Push :: k))
  in
```

Notez l'appel récursif à `tr_args` pour traduire la queue de la liste d'arguments, qui prend comme suite le code évaluant le premier argument. Autrement dit, l'argument de tête est évalué et placé sur la pile en dernier : les arguments seront bien empilés dans l'ordre du dernier au premier.

Ne manque que la fonction auxiliaire `convert_var` associant chaque identifiant de variable IMP à une description de variable au format LLIR qui distingue variables globales, paramètres

9. Une autre issue consisterait à utiliser une structure de données autre que la liste, avec concaténation en temps constant.

formels et variables locales, et dans les deux derniers cas précise un numéro. La seule chose notable dans cette dernière fonction est l'ordre dans lequel on consulte les tables : en cas de collision de noms, les variables locales masquent les paramètres, qui eux-mêmes masquent les variables globales.

```
    (* Fonction auxiliaire : premier indice auquel [x] apparaît dans [l] *)
    let get_i_opt x l =
      let rec scan i = function
        | [] -> None
        | y::l -> if x = y then Some i else scan (i+1) l
      in
      scan 0 l
    in

    let convert_var x =
      match get_i_opt x Imp.(fdef.locals) with
        | Some i -> Llir.Local i
        | None -> match get_i_opt x Imp.(fdef.params) with
            | Some i -> Llir.Param i
            | None -> Global x
    in
```

La traduction de IMP vers LLIR est maintenant achevée. Nous allons pouvoir poursuivre dans les sections suivantes en allant depuis LLIR vers un langage assembleur réel.

Bonus : interprétation de LLIR Avec les éléments en place, on peut écrire un interprète pour les programmes LLIR. Cette étape n'est pas utile pour l'écriture de notre compilateur IMP, mais elle donne une manière de tester le traducteur IMP vers LLIR que nous venons de réaliser. En outre, l'écriture d'un interprète est une manière alternative de décrire la sémantique de notre représentation intermédiaire.

La fonction principale d'interprétation s'applique au programme à interpréter ainsi qu'à une liste d'arguments qui sont les arguments à donner à la fonction `main`. Dans cette fonction principale, on crée une table pour les variables globales (ici, elles ne sont pas initialisées, mais on pourrait aussi choisir de toutes les initialiser à zéro), et on déclenche l'appel initial à la fonction `main`. Cet appel utilise une fonction auxiliaire `exec_call` que l'on va définir à l'intérieur de `exec_prog` et qui sera à nouveau utilisée pour chaque appel de fonction ordinaire.

```
  let exec_prog prog args =
    let globals = Hashtbl.create (List.length prog.globals) in

    ...

    let main = List.find (fun fdef -> fdef.name = "main") prog.functions in
    exec_call main args
```

La fonction `exec_call` s'applique à la description d'une fonction LLIR et une liste de valeurs à lui passer en arguments, et interprète l'appel de fonction correspondant. `exec_call` est définie comme une fonction interne à `exec_prog`, et a donc accès notamment à la table des variables globales.

Comme `exec_prog`, cette nouvelle fonction commence par mettre en place des tables pour la gestion des variables associées à l'appel, en l'occurrence avec un tableau pour les paramètres effectifs et un tableau pour les variables locales. On crée également à ce niveau une pile (rappel : on a dit dans la description du langage LLIR que chaque appel de fonction travaillait avec sa propre pile) et deux fonctions auxiliaires pour ajouter ou retirer un élément de la pile. Une fois cette mise en place effectuée, la fonction entame l'interprétation du code de la fonction avec une nouvelle fonction auxiliaire `exec_seq`, en commençant pas le bloc identifié comme bloc d'entrée, et en donnant arbitrairement la valeur initiale `0` au registre.

```
    let rec exec_call f args =
      let params = Array.of_list args in
      let locals = Array.make fdef.nb_locals 0 in
      let stack = Stack.create () in
      let push v = Stack.push v stack in
      let pop () = Stack.pop stack in
```

```
    ...

    let fdef = List.find (fun fdef -> fdef.name = f) prog.functions in
    let s = Hashtbl.find fdef.code fdef.start in
    exec_seq s 0
  in
```

La fonction `exec_seq` d'interprétation d'un bloc d'instructions est définie avec une fonction `exec_instr` d'interprétation des instructions élémentaires. Toutes deux sont définies comme des fonctions internes à `exec_call`, et ont donc accès à toutes les structures définies ci-dessus pour les variables locales ou globales. Notez de plus que toutes les structures introduites pour les variables sont mutables, et pourront donc être modifiées par `exec_instr`.

La fonction `exec_instr` interprète une instruction élémentaire. Elle prend en paramètre supplémentaire la valeur courante du registre, et renvoie la valeur du registre mise à jour après exécution (il ne faut pas oublier de renvoyer cette valeur même lorsqu'elle n'est pas modifiée).

```
    let exec_instr i r = match i with
      | Cst n   -> n
      | Push    -> push r; r
      | Add     -> r + pop()
      | Mul     -> r * pop()
      | Lt      -> if r < pop() then 1 else 0
      | Putchar -> print_char(char_of_int r); r
```

Cette fonction est également susceptible d'accéder aux variables ou de les modifier. On traite un cas de lecture et un cas d'écriture pour chaque sorte de variable, pour tenir compte des différentes structures utilisées.

```
      | Get(Global x) -> Hashtbl.find globals x
      | Get(Param k) -> params.(k)
      | Get(Local k) -> locals.(k)
      | Set(Global x) -> Hashtbl.replace globals x r; r
      | Set(Param k) -> params.(k) <- r; r
      | Set(Local k) -> locals.(k) <- r; r
```

Pour traiter un nouvel appel de fonction, on va chercher la définition de fonction correspondant à l'identifiant `f` donné. Cette définition nous permet de connaître la fonction exécuter, mais aussi le nombre de paramètres à lire sur la pile. Une fonction auxiliaire `pop_args` récupère le bon nombre d'arguments sur la pile. On déclenche ensuite l'appel à proprement parler par un appel (récursif) à la première fonction auxiliaire `exec_call`.

```
      | Call(f) ->
        let fdef = List.find (fun fdef -> fdef.name = f) prog.functions in
        let rec pop_args = function
          | 0 -> []
          | n -> pop() :: pop_args (n-1)
        in
        let args = pop_args fdef.nb_params in
        exec_call fdef args
```

On laisse de côté les cas relatifs aux instructions de saut, qui seront interceptés en amont par la fonction `exec_seq` décrite ci-après.

```
      | _ -> assert false
    in
```

Notez une asymétrie de traitement entre le registre d'une part, et la pile ou les variables d'autre part : la fonction `exec_instr` prend en paramètre la valeur du registre et renvoie la valeur mise à jour, alors qu'elle gère les autres éléments à l'aide de structures mutables. On aurait cependant très bien pu faire du registre une nouvelle donnée mutable, ou prendre en argument et renvoyer les autres structures (qui auraient alors pu être immuables). La version asymétrique présentée ici est une coquetterie produisant un code plus compact.

La fonction auxiliaire `exec_seq` est définie au même niveau que `exec_instr`. Comme elle, `exec_seq` prend en paramètre supplémentaire la valeur courante du registre et renvoie la

valeur mise à jour. Cette fonction `exec_seq` se concentre sur la gestion de la séquence et sur les instructions `jump`, `cjump` et `return`. Point commun entre ces trois instructions : on ignore les éventuelles instructions venant après, car l'exécution continue avec un autre bloc, voire dans le cas de `return` revient directement au contexte appelant. Notez que si le programme LLIR est bien formé, cette liste d'instructions suivantes ignorées est vide.

```
    let rec exec_seq seq r = match seq with
      | Jump(lab) :: _ ->
        let s = Hashtbl.find fdef.code lab in
        exec_seq s r
      | CJump(lab1, lab2) :: _ ->
        let lab = if r <> 0 then lab1 else lab2 in
        let s = Hashtbl.find fdef.code lab in
        exec_seq s r
      | Return :: _ -> r
```

Pour toutes les autres instructions, `exec_seq` fait appel à `exec_instr`. On suppose en outre que la séquence à exécuter n'est jamais vide.

```
      | i::s -> exec_instr i r |> exec_seq s
      | [] -> assert false
    in
```

6.5 Architecture cible

La cible ultime de la compilation est la production d'un programme directement exécutable par un ordinateur physique. La nature d'un tel programme est intimement liée à l'architecture de l'ordinateur cible.

Architecture et boucle d'exécution Dans ce cours on utilisera une architecture simple mais réelle appelée MIPS, qu'on trouve notamment dans des puces embarquées. Les deux principaux composants de l'architecture sont :

— un *processeur*, comportant un petit nombre de registres qui permettent de stocker quelques données directement accessibles, ainsi que des unités de calcul qui opèrent sur les registres, et

— une grande quantité de *mémoire*, où sont stockées à la fois des données et le programme à exécuter lui-même.

On a deux unités de base pour la mémoire : l'*octet*, une unité universelle désignant un groupe de 8 bits, et le ***mot mémoire***, qui représente l'espace alloué à une donnée « ordinaire », comme un nombre entier. En MIPS, le mot mémoire vaut 4 octets : les données manipulées sont donc généralement représentées par des séquences de 32 bits. En conséquence, un ***entier machine***, c'est-à-dire un nombre entier primitif manipulé par ce processeur est compris entre -2^{31} et $2^{31}-1$ (ou entre 0 et $2^{32}-1$ dans le cas d'entiers « non signés »).

L'architecture MIPS comporte 32 registres, pouvant chacun stocker un mot mémoire. Ces données stockées dans les registres sont directement accessibles à l'unité de calcul.

La mémoire principale a une étendue de 2^{32} octets. Chaque octet de la mémoire est associé à une ***adresse***, qui est un entier entre 0 et $2^{32}-1$, et c'est exclusivement en utilisant ces adresses que l'on accède aux éléments stockés en mémoire. Les données étant organisées sur des mots mémoire de 4 octet, l'architecture impose en outre que les adresses des données ordinaires soient des multiples de 4 (il existe quelques exceptions). L'accès à la mémoire est relativement coûteux : une seule opération de lecture ou d'écriture en mémoire a un coût nettement supérieur à une opération arithmétique travaillant directement sur les registres.

Le programme à exécuter est une séquence d'instructions directement interprétables par le processeur. Chaque instruction est codée sur 4 octets, et l'ensemble est stocké de manière contiguë dans la mémoire.

Un registre spécial `pc` (qui ne fait pas partie des 32) contient l'adresse de l'instruction courante (*program counter*, ou *pointeur de code*). L'exécution d'un programme procède en répétant le cycle suivant :

1. lecture d'un mot mémoire à l'adresse `pc` (*fetch*),
2. interprétation des octets lus comme une instruction (*decode*),
3. exécution de l'instruction reconnue (*execute*),

4. mise à jour de pc pour passer à l'instruction suivante (par défaut : incrémenter de 4 octets pour passer au mot mémoire suivant).

Instructions On distingue trois catégories principales d'instructions :
— des instructions arihtmétiques, appliquant des opérations élémentaires à des valeurs stockées dans des registres,
— des instructions d'accès à la mémoire, pour transférer des valeurs de la mémoire vers les registres ou inversement,
— des instructions de contrôle, pour gérer le pointeur de code pc.

Dans les 32 bits utilisés pour coder une instruction, les 6 premiers désignent l'opération à effectuer (***opcode***), et les suivants donnent les paramètres ou des précisions éventuelles sur l'opération. Ci-dessous, quelques exemples avant d'aborder à la section suivante la manière de programmer avec ces instructions.

Par exemple : l'instruction numéro 9 prend en paramètres un registre source r_s, un registre de destination r_t, et un nombre n non signé de 16 bits, et place dans r_t la valeur $r_s + n$. Il suffit de 5 bits pour désigner l'un des 32 registres. Si r_s est le registre numéro 5, r_t le registre numéro 17, et n la valeur 42, alors l'instruction est représentée par un mot mémoire décomposé ainsi :

op	r_s	r_t	n
001000	00101	10001	0000000000101010

Autre exemple : l'instruction numéro 15 prend en paramètres un registre de destination r_t et un nombre n de 16 bits, et place n dans les deux octets supérieurs de r_t. Si r_t est le registre numéro 17 et n la valeur 42, alors l'instruction est représentée par le mot mémoire suivant, où 5 bits ne sont pas utilisés.

op		r_t	n
001111	00000	10001	0000000000101010

Quizz : comment utiliser les instructions précédentes pour charger la valeur `0xeff1cace` *dans le registre numéro 2 ?*

La plupart des opérations arithmétiques binaires utilisent l'*opcode* 0, l'opération précise étant alors précisée dans les 6 derniers bits de l'instruction (cette dernière partie est appelée la *fonction*). Ces instructions prennent invariablement trois paramètres : deux registres r_s et r_t pour les valeurs auxquelles appliquer l'opération, et un registre de destination r_d. Ainsi l'opération d'addition, qui est identifiée par l'*opcode* 0 et la fonction 32. Ci-dessous, opération d'addition avec r_s, r_t et r_d les registres de numéros respectifs 1, 2 et 3. Notez que 5 bits sont inutilisés.

op	r_s	r_t	r_d		f
000000	00001	00010	00011	00000	100000

Certaines opérations ignorent le paramètre r_s et fournissent à la place une donnée constante à l'aide des 5 bits qui étaient restés libres ci-dessus. Ainsi, l'opération de décalage vers la gauche prend trois paramètres : un registre r_t à qui appliquer un décalage, un registre r_d de destination, et une constante k sur 5 bits indiquant de combien de bits vers décaler r_t. C'est opération correspond toujours à l'*opcode* 0, mais cette fois avec la fonction 0. Ainsi, pour décaler de le registre numéro 4 de 5 bits vers la gauche et placer le résultat dans le registre numéro 6, on a :

op		r_t	r_d	k	f
000000	00000	00100	00110	00101	000000

Ce schéma général de découpage vaut encore pour les instructions d'accès à la mémoire. L'instruction de lecture d'un mot mémoire par exemple prend en paramètres deux registres r_s et r_t et un entier signé n sur 16 bits. Le registre r_s est supposé contenir une adresse, et l'entier n est appelé *décalage*. L'instruction de lecture calcule une adresse a en additionnant l'adresse de base r_s et le décalage n, et transfère vers le registre r_t le mot mémoire lu à l'adresse a. Le code de cette opération est 35. Ainsi, pour placer dans le registre numéro 3 la donnée trouvée à l'adresse obtenue en ajoutant 8 octets à l'adresse donnée par le registre numéro 5, on a la séquence :

op	r_s	r_t	n
100011	00101	00011	0000000000001000

Les instructions de contrôle utilisent encore une variante de ce même découpage. On a par exemple une instruction de saut conditionnel, qui prend en paramètres un registre r_s et un entier n signé sur 16 bits, et qui avance de n instructions dans le code du programme si la valeur de r_s est strictement positive. Le code de cette instruction est 7. Pour reculer de 5 instructions lorsque la valeur du registre 3 est strictement positive, on a donc l'instruction :

op	r_s	r_t	n
000111	00011	00000	1111111111111011

Enfin, l'instruction numéro 2 prend pour unique paramètre un entier sur 26 bits interprété comme l'adresse a d'une instruction i, et va faire se poursuivre l'exécution à partir de cette instruction i. Autrement dit, on écrase la valeur du pointeur `pc` pour la remplacer par a. Pour aller à l'instruction d'adresse `0x00efface` on aura donc l'instruction :

op	a
000010	00111011111111101011001110

Nous verrons à la section suivante qu'en pratique, on ne compose pas directement ces codes représentant chaque instruction. On utilise à la place une forme textuelle appelée le ***langage assembleur***, qui est ensuite automatiquement traduite en la version codée par des mots binaires.

Autres familles d'architectures De nombreux aspects de l'architecture MIPS que nous venons de décrire sont partagés avec les autres architectures majeures. Pour commencer, la boucle d'exécution *fetch/decode/execute* avec un programme stocké en mémoire est une réalisation directe du modèle de von Neumann, à la base de tous les ordinateurs depuis l'origine de l'informatique.

D'une famille d'architectures à l'autre, on observe des variations par exemple sur la taille d'un mot mémoire, avec notamment les architectures 64 bits où le mot mémoire est de 8 octets. Le nombre de registre peut également varier, ainsi que l'ensemble des instructions disponibles ou les manières d'accéder à la mémoire. Les différences les plus significatives ne sont en revanche pas toujours les plus visibles pour le programmeur.

Dans l'architecture MIPS, les instructions sont codées suivant un schéma uniforme qui limite le nombre d'instructions qui peuvent être proposées. Cette approche est typique des architectures de la famille *RISC* (*Reduced Instruction Set Computer*), qui comprennent notamment les architectures ARM.

À l'inverse, les architectures de la famille *CISC* (*Complex Instruction Set Computer*) proposent un codage variable des instructions, par exemple allant de 1 à 8 octets. Ceci permet entre autres choses de proposer un nombre beaucoup plus importants d'instructions. Cette famille comprend notamment les architectures Intel.

Dans une représentation variable de type *CISC*, on dispose d'instructions riches permettant un code optimisé. En outre, le format variable permet de compresser le code, en donnant une représentation compacte aux instructions les plus fréquemment utilisées. Dans une représentation uniforme de type *RISC*, le décodage des instructions est plus simple, et l'ensemble du processeur est de même plus simple et utilise moins de transistors, ce qui permet une moindre consommation d'énergie. En conséquence les architectures ARM sont omniprésentes dans les *smartphones*, où la gestion de la batterie est critique. Les architectures Intel, réputées plus puissantes, ont à l'inverse longtemps été hégémoniques dans le domaine des ordinateurs grand public, mais cela pourrait basculer. En particulier, le format uniforme des architectures RISC facilite grandement le traitement de plusieurs instructions en parallèle, ce qui est une source très importante d'optimisation des architectures. La montée en puissance de ces optimisations rend maintenant les architectures RISC compétitives avec les architectures CISC en termes de puissance de calcul, et les puces ARM commencent à prendre une place dans le monde des ordinateurs grands publics [10].

6.6 Langage assembleur MIPS

En pratique, on n'écrit pas directement les suites de bits du langage machine. On utilise à la place un langage d'assemblage, ou assembleur, qui est quasiment isomorphe au langage machine mais permet une écriture plus agréable. En langage assembleur on a en particulier :

10. Ce commentaire est écrit en 2021. Apple est en train de remplacer les processeurs Intel par des processeurs ARM dans l'ensemble de ses ordinateurs, avec succès jusque là (gros gain énergétique sans perte notable de puissance de calcul), et des rumeurs suggèrent que Google préparerait une évolution similaire.

— des écritures textuelles pour les instructions,
— la possibilité d'utiliser des étiquettes symboliques plutôt que des adresses explicites,
— une allocation statique des données globales,
— quelques *pseudo*-instructions, qui correspondent à une combinaison simple d'instructions réelles.

Dans cette section, on présente le langage assembleur MIPS.

Pour tester nos programmes MIPS, on utilisera le simulateur MARS. Ce simulateur peut être appelé directement depuis la ligne de commande pour exécuter un programme assembleur donné, mais dispose aussi d'un mode graphique dans lequel on peut suivre pas à pas l'exécution d'un programme et l'évolution des registres et de la mémoire.

Registres Dans l'assembleur MIPS, les 32 registres sont désignés par leur numéro, de `$0` jusqu'à `$31`. Alternativement à son numéro, chaque registre possède en outre un nom reflétant sa fonction.

n°	*nom*
`$0`	`$zero`
`$1`	`$at`
`$2`	`$v0`
`$3`	`$v1`
`$4`	`$a0`
`$5`	`$a1`
`$6`	`$a2`
`$7`	`$a3`

n°	*nom*
`$8`	`$t0`
`$9`	`$t1`
`$10`	`$t2`
`$11`	`$t3`
`$12`	`$t4`
`$13`	`$t5`
`$14`	`$t6`
`$15`	`$t7`

n°	*nom*
`$16`	`$s0`
`$17`	`$s1`
`$18`	`$s2`
`$19`	`$s3`
`$20`	`$s4`
`$21`	`$s5`
`$22`	`$s6`
`$23`	`$s7`

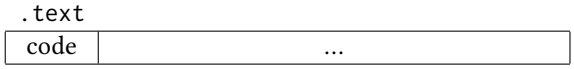
n°	*nom*
`$24`	`$t8`
`$25`	`$t9`
`$26`	`$k0`
`$27`	`$k1`
`$28`	`$gp`
`$29`	`$sp`
`$30`	`$fp`
`$31`	`$ra`

Les 24 registres `$v*`, `$a*`, `$t*` et `$s*` sont des registres ordinaires, que l'on peut librement utiliser pour des calculs (on verra les spécificités de ces quatre familles plus tard). Les 8 autres registres ont des rôles particuliers : `$gp`, `$sp`, `$fp` et `$ra` contiennent des adresses utiles, `$zero` contient toujours la valeur 0 et ne peut pas être modifié, et les 3 registres `$at` et `$k*` sont réservés respectivement pour l'assembleur et pour le système (il ne faut donc pas les utiliser).

Instructions et pseudo-instructions Un programme en langage assembleur MIPS prend la forme d'un fichier texte, où chaque ligne contient une instruction ou une annotation. On introduit une liste d'instructions avec l'annotation

```
.text
```

Les instructions ainsi introduites sont placées dans une zone de la mémoire réservée au programme, tout en bas de la mémoire (c'est-à-dire aux adresses les plus basses).



Chaque instruction est construite avec un mot clé appelé ***mnémonique***, désignant l'opération à effectuer, et un certain nombre de paramètres séparés par des virgules. Voici par exemple une instruction qui initialise le registre `$t0` avec la valeur 42

```
li $t0, 42
```

et une instruction qui copie la valeur présente dans le registre `$t0` vers le registre `$t1`.

```
move $t1, $t0
```

Comme le montrent ces exemples, les paramètres donnés aux instructions peuvent être, selon les instructions, des registres désignés par leur nom et/ou des constantes entières écrites de manière traditionnelle.

Une mnémonique décrit l'opération à effectuer, en général à l'aide d'un mot ou d'un acronyme. On a ici `move` pour le « déplacement » d'une valeur ou `li` pour *Load Immediate*, c'est-à-dire pour le chargement d'une valeur constante (on appelle valeur *immédiate* une valeur constante directement fournie dans le code assembleur). Certaines de ces mnémoniques correspondent directement à des instructions machines, et peuvent être associées à un *opcode* et le cas échéant à une fonction. D'autres mnémoniques sont des *pseudo-instructions*, c'est-à-dire des raccourcis pour de courtes séquences d'instructions machine (généralement des séquences d'une ou deux instructions seulement).

En l'occurrence :

— `li` est une pseudo-instruction qui s'adapte à la constante à charger : si cette constante s'exprime sur 16 bits alors elle sera traduite par une unique instruction machine, mais dans le cas contraire on aura deux instructions pour charger indépendamment les deux octets hauts et les deux octets bas. Ainsi, cette pseudo-instruction permet au programmeur de s'affranchir de la limite de 16 bits pour les valeurs immédiates.
— `move` est une pseudo-instruction, traduite par une unique instruction d'addition : plutôt que d'inclure dans les circuits du processeur une instruction spécifique pour une affectation de la forme `$k` ← `$k'`, on réutilise l'instruction d'addition qui est de toute façon présente, en fixant le deuxième opérande à zéro : `$k` ← (`$k'` + `$zero`).

Arithmétique Les opérations arithmétiques et logiques du langage assembleur MIPS suivent toutes le même format « trois adresses » : après la mnémonique identifiant l'opération viennent dans l'ordre le registre cible (où sera placé le résultat de l'opération) puis les deux opérandes.

```
<mnemo>  <dest>, <r1>, <r2>
```

On y trouve des opérations variées, dont la plupart vous seront familières.
— Opérations arithmétiques : `add`, `sub`, `mul`, `div`, `rem` (*remainder* : reste), ...
— Opérations logiques : `and`, `or`, `xor` (*exclusive or*), ...
— Comparaisons : `seq` (*equal* : égalité), `sne` (*not equal* : inégalité) `slt` (*less than* : $<$), `sle` (*less or equal* : $\leqslant$), `slt` (*greater than* : $>$), `sge` (*greater or equal* : $\geqslant$), ...

À ces opérations binaires s'ajoutent quelques opérations unaires classiques également, avec un registre de destination et un seul opérande : `abs` (valeur absolue), `neg` (opposé), **`not`** (négation logique).

Les opérations arithmétiques admettent parfois des variantes pour l'arithmétique non signée, c'est-à-dire pour manipuler des entiers entre 0 et $2^{32} - 1$ plutôt que des entiers entre -2^{31} et $2^{31} - 1$. Ces opérations sont repérables par la présence d'un `u`. Par exemple : `addu`.

Un certain nombre des opérations précédentes admettent également une variante dans laquelle le deuxième opérande est une valeur immédiate plutôt qu'un registre. Ces opérations sont repérables par la présence d'un `i`. Par exemple : `addi`, ou encore `addiu`.

Enfin, les opérations de manipulation des séquences de bits par décalage ou rotation suivent le même format, mais en prenant par défaut une valeur immédiate pour le deuxième opérande (l'amplitude du décalage).
— Décalages et rotations : `sll` (*shift left logical*), `sra` (*shift right arithmetic*), `srl` (*shif right logical*), `rol` (*rotation left*), `ror` (*rotation right*), ...

Des variantes avec une amplitude de décalage variable existent, repérées par la présence d'un `v`. Par exemple : `sllv`. Dans ce cas, le deuxième opérande est un registre.

Voici par exemple une séquence d'instructions pour évaluer la comparaison `3*4 + 5 < 2*9`, ainsi qu'un tableau montrant l'évolution des différents registres à mesure que l'exécution progresse.

	`$t0`	`$t1`	`$t2`	`$t3`	`$t4`	`$t5`	`$t6`	`$t7`	`$t8`
`li $t0, 3`	3								
`li $t1, 4`	3	4							
`li $t2, 5`	3	4	5						
`li $t3, 2`	3	4	5	2					
`li $t4, 9`	3	4	5	2	9				
`mul $t5, $t0, $t1`	3	4	5	2	9	12			
`add $t6, $t5, $t2`	3	4	5	2	9	12	17		
`mul $t7, $t3, $t4`	3	4	5	2	9	12	17	18	
`slt $t8, $t6, $t7`	3	4	5	2	9	12	17	18	1

L'exemple précédent n'est cependant guère judicieux : il utilise un nouveau registre pour chaque opération, et n'utilise pas toujours les opérations les plus efficaces. Voici une meilleure

version pour réaliser le même calcul.

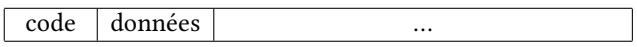
	$t0	$t1
`li $t0, 3`	3	
`li $t1, 4`	3	4
`mul $t0, $t0, $t1`	12	4
`addi $t0, $t0, 5`	17	4
`li $t1, 9`	17	9
`sll $t1, $t1, 1`	17	18
`slt $t0, $t0, $t1`	1	18

Données statiques À côté des instructions, un programme assembleur peut également déclarer et initialiser un certain nombre de données, correspondant par exemple à des variables globales du programme.

On introduit une liste de déclarations de telles données avec l'annotation

```
.data
```

En mémoire, les données ainsi déclarées sont placées dans une zone située après la zone réservée aux instructions du programme.



La définition d'une donnée ou d'un groupe de données combine :

— la déclaration d'une étiquette symbolique, c'est-à-dire d'un nom qui pourra être utilisé pour accéder à la donnée,

— la fourniture d'une ou plusieurs valeurs initiales.

Un mot-clé permet de préciser la nature des données fournies : `.word` pour une valeur 32 bits (4 octets), `.byte` pour une valeur d'un unique octet (8 bits), `.asciiz` pour une chaîne de caractères au format ASCII (un caractère par octet, la chaîne étant terminée par l'octet zéro (`0x00`).

On déclare ainsi une donnée désignée par le nom `reponse` et valant 42 :

```
reponse: .word 42
```

Ainsi, un mot mémoire va être réservé dans la zone des données et initialisé avec la valeur 42. L'identifiant `reponse` désigne l'adresse de ce mot mémoire, et peut être utilisée notamment pour récupérer ou modifier la valeur qui y est stockée.

On peut de même introduire une séquence de données.

```
prems: .word 2 3 5 7 11 13 17 19
```

La première donnée est placée à l'adresse correspondant à l'étiquette `prems`, puis les suivantes sont l'une après l'autre, de 4 octets en 4 octets.

L'adresse réservée pour chaque donnée sera calculée au moment de la traduction du programme assembleur en langage machine. À cette occasion, chaque mention d'une étiquette sera remplacée en dur par un accès à l'adresse correspondante.

Accès à la mémoire Le format standard des adresses mémoire en MIPS contient deux composantes :

— une adresse de base, donnée par la valeur d'un registre `$r`,

— un décalage, donné par une constante `d` (16 bits, signée).

On note `d($r)` une telle adresse, dont la valeur est donc `$r + d`.

Les instructions d'accès à la mémoire permettent alors de transférer une valeur depuis une adresse mémoire vers un registre (lecture, *load*) ou au contraire depuis un registre vers une adresse mémoire (écriture, *store*).

```
lw $t0, 8($a1)
sw $t0, 0($t1)
```

Les deux instructions précédentes enchaînent donc la lecture du mot mémoire à l'adresse obtenue en ajoutant 8 à la valeur du registre `$a1` (la valeur lue étant placée dans le registre

`$t0`), puis l'écriture de la valeur du registre `$t0` à l'adresse donnée directement par la valeur du registre `$t1`.

Les mnémoniques `lw` et `sw` signifient respectivement *Load Word* et *Store Word*. Des variantes existent pour lire ou écrire seulement un octet (`lb`, `lbu`, `sb`), ou un demi-mot (`lh`, `lhu`, `sh`), ou encore un double mot (`ld`, qui transfère les 8 octets lus vers 2 registres : celui donné dans l'instruction et le suivant, et `sd` qui fonctionne symétriquement).

L'assembleur permet également quelques notations simplifiées des adresses, qui sont converties vers le format standard au moment de la traduction vers le langage machine. On peut par exemple omettre le décalage lorsqu'il vaut zéro

```
sw $t0, ($t1)
```

ou encore accéder directement à l'adresse donnée par une étiquette, avec un éventuel décalage positif ou négatif.

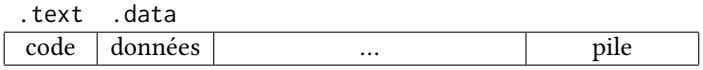
```
lw $t0, prems
lw $t1, prems+4
lw $t2, prems+12
```

Enfin, une instruction `la` (*Load Address*) permet de récupérer ou calculer une adresse, sans lire son contenu.

```
la $t0, prems
```

Pile Le mécanisme des données statiques n'est pas suffisant dans un programme ordinaire [11]. On a généralement besoin de pouvoir stocker des données en mémoire, ne serait-ce que temporairement, sans avoir prédit précisément dès l'écriture du programme la quantité d'espace occupée. Ainsi, dans un programme ordinaire la plus grande partie de la mémoire est *dynamique*, c'est-à-dire que l'allocation de l'espace aux différentes données n'y est décidée que *pendant l'exécution* du programme, et qu'une adresse occupée par une donnée à un moment de l'exécution a vocation à être libérée lorsque la donnée n'est plus utile, pour être ensuite réaffectée à une nouvelle donnée.

La partie supérieure de cette grande région de mémoire dynamique est appelée la ***pile***, et fonctionne de manière identique à la structure de données de pile. Il s'agit donc d'une structure linéaire, dont une extrémité est appelée le ***fond*** et l'autre le ***sommet***, et qui est modifiée uniquement du côté de son sommet : on peut ajouter de nouveaux éléments au sommet de la pile, ou retirer les éléments du sommet. Cette organisation implique qu'un élément retiré de la pile est toujours l'élément de la pile le plus récemment ajouté.



Petite particularité notable ici : la pile est placée à l'extrémité de la mémoire, du côté des adresses les plus hautes. Il n'est donc pas possible de l'étendre « vers le haut ». Le fond de cette pile est donc calé sur l'adresse la plus grande de la mémoire, et le sommet à une adresse inférieure : la pile croît ainsi en s'étendant vers les adresses inférieures.

Le sommet de la pile est donc l'adresse mémoire la plus basse utilisée par la pile. Cette adresse est stockée dans le registre `$sp`. On réaliser les opérations usuelles d'une structure de pile de la manière suivante.

— Pour consulter la valeur au sommet de la pile (`peek`), on lit un mot à l'adresse donnée par `$sp`.

```
    lw $t0, 0($sp)
```

Pour aller consulter des éléments plus profonds dans la pile, on ajoute un décalage de 4 octets pour chaque élément à sauter. On consulte donc le quatrième élément avec un décalage de 12 octets.

```
    lw $t0, 12($sp)
```

— Pour ajouter un élément au sommet de la pile (`push`), on décrémente `$sp` de 4 octets, puis on écrit la valeur souhaitée à la nouvelle adresse `$sp`.

11. On s'astreint parfois à s'en contenter, pour limiter les possibilités de bugs dans des catégories particulières de programmes, notamment dans des domaines critiques comme l'avionique.

```
    addi $sp, $sp, -4
    sw   $t0, 0($sp)
```

— Pour retirer l'élément au sommet de la pile, on incrémente `$sp` de 4 octets. Notez qu'il n'est pas besoin « d'effacer » la valeur présente à l'ancienne adresse `$sp` : le simple fait que cette adresse soit maintenant inférieure à `$sp` la désigne comme insignifiante.

```
    addi $sp, $sp, 4
```

L'opération usuelle pop, consistant à récupérer l'élément au sommet tout en le retirant de la pile, combine cet incrément avec une opération peek.

```
    lw $t0, 0($sp)
    addi $sp, $sp, 4
```

Le code assembleur obtenu est d'une certaine manière symétrique à celui de push.

Sauts et branchements Rappelons que les instructions d'un programme MIPS sont stockées en mémoire. Chaque instruction a donc une adresse. De même que les adresses des données, les adresses de certaines instructions peuvent être désignées par des étiquettes : une étiquette insérée dans une séquence d'instructions MIPS désigne l'adresse de l'instruction qui la suit immédiatement, et pourra être utilisée comme cible d'une instruction de saut.

MIPS propose une variété d'instructions (ou pseudo-instructions) de branchement conditionnel. En voici un exemple :

```
  beq $t0, $t1, lab
```

Cette instruction compare les valeurs des registres `$t0` et `$t1`. Si ces deux valeurs sont égales, alors l'exécution se poursuivra avec l'instruction d'étiquette `lab` (techniquement, l'instruction de branchement commande une modification du registre spécial pc). Sinon, l'exécution se poursuivra naturellement avec l'instruction suivante.

Des variantes existent pour d'autres modalités de comparaison

$=$	$\neq$	$<$	$\leqslant$	$>$	$\geqslant$
beq	bne	blt	ble	bgt	bge

et chacune admet à nouveau une variante pour comparer une unique valeur à zéro : beqz, bnez, bltz, ... Certaines admettent également pour deuxième paramètre une valeur immédiate plutôt qu'un registre. Enfin, une pseudo-instruction `b lab` provoque un branchement inconditionnel vers l'instruction d'étiquette `lab`.

Voici un exemple de fragment de code calculant la factorielle du nombre stocké dans le registre `$a0`, et plaçant le résultat dans le registre `$v0`.

```
1           move $t0, $a0
2           li $t1, 1
3           b test
      loop:
4           mul $t1, $t1, $t0
5           addi $t0, $t0, -1
      test:
6           bgtz $t0, loop
7           move $v0, $t1
```

Les deux premières instructions initialisent les deux registres `$t0` et `$t1` qui seront utilisés pour le calcul. Les instructions 4 à 6 forment une boucle, et la dernière instruction transfère le résultat dans `$v0` comme demandé. Le corps de la boucle est constitué des instructions 4 et 5, la première étant associée à l'étiquette `loop`. Le test de la boucle est à l'instruction 6, associée à l'étiquette `test`. Lorsque ce test est positif, on déclenche un nouveau tour de boucle à l'aide d'un branchement vers `loop`. À l'inverse, lorsque le test est négatif on poursuit avec l'instruction suivante, c'est-à-dire l'instruction 7. Enfin, l'instruction 3 démarre la boucle en branchant vers le test. *Notez que l'on aurait pu se passer de* `$t0`*, de* `$t1` *et des instructions 1 et 7 en travaillant exclusivement avec* `$a0` *et* `$v0`*.*

Les instructions de branchement sont utilisées pour des déplacements « locaux » dans la séquence d'instructions, c'est-à-dire en restant dans la même unité de code (par exemple :

la même fonction). Les instructions machine correspondantes font avancer ou reculer le pointeur de code d'un certain nombre d'instructions, ne dépassant pas 2^{15}.

À l'inverse, les instructions de sauts définissent la prochain instruction de manière absolue, en fournissant directement son adresse. Cela peut se faire à l'aide d'une étiquette, et dans ce cas la différence avec une instruction de branchement n'est pas flagrante,

```
j label
```

ou en fournissant directement une adresse calculée, stockée dans un registre.

```
jr $t0
```

Les instructions de saut servent notamment dans le cadre d'un saut non local, comme un appel de fonction. Pour cette situation, on dispose également de deux instructions qui, avant de sauter, sauvegardent une adresse de retour dans le registre `$ra`, qui permettra plus tard de revenir à l'exécution de la séquence en cours.

```
jal label
jalr $t0
```

On reviendra sur ce mécanisme d'appel de fonction à la section suivante.

Appels système Pour certaines fonctionnalités dépendant du système d'exploitation, un code assembleur doit faire appel aux bibliothèques système. Dans ce cours, on exécutera notre code MIPS dans un simulateur, qui prendra en charge ces différents services avec une instruction dédiée. On décrit donc ici des fonctionnalités du simulateur, qui miment des services dépendant normalement du système d'exploitation.

On déclenche un appel système à l'aide de l'instruction `syscall`, après avoir placé dans le registre `$v0` un code désignant le service demandé. Voici une petite sélection :

service	code	arg.	rés.
affichage	1 (entier) 4 (chaîne) 11 (ascii)	`$a0`	
lecture	5 (entier)		`$v0`
arrêt	10		
extension mémoire	9	`$a0`	`$v0`

On trouve encore des services pour la manipulation de fichiers, et d'autres variantes d'affichage ou de lecture.

Hello, world Le programme le plus célèbre de la terre, en MIPS.

```
        .text
main:   li $v0, 4
        la $a0, hw
        syscall
        li $v0, 10
        syscall

        .data
hw:     .asciiz "hello world\n"
```

Notes : 4 est le code de l'appel système d'affichage d'une chaîne de caractères, 10 celui de l'arrêt. L'annotation `.asciiz` permet de placer une chaîne de caractères dans les données statiques, dont l'adresse est ici associée à l'étiquette `hw`.

6.7 Fonctions et conventions d'appel

Un point clé du développement de programmes réalistes est la possibilité de définir et d'appeler des fonctions. Du point de vue de la compilation, cela demande un peu de technologie supplémentaire. Fixons d'abord le vocabulaire, en observant l'***appel de fonction*** `f(6)` dans le code suivant.

```
1    void g() {
2      int y;
3      y = f(6);
4      print(y);
5    }

6    int f(int x) {
7      return x*7;
8    }
```

La valeur `6` est le ***paramètre effectif*** (on dit aussi l'*argument*) de l'appel de fonction `f(6)`. Le bloc de code allant des lignes 2 à 4 est le ***contexte appelant***, et la fonction `f` définie aux lignes 6 à 8 est la ***fonction appelée***. Du côté de la fonction appelée, la variable `x` est le ***paramètre formel*** de la fonction, et l'expression `x*7` calcule le résultat ***renvoyé*** par la fonction.

Appel de fonction : mécanisme de base Un appel de fonction implique des transerts de différents natures entre le contexte appelant et la fonction appelée. On a d'une part un transfert de données, avec

- un ou plusieurs paramètres effectifs transmis par le contexte appelant à la fonction appelée,
- un résultat, renvoyé au contexte appelant par la fonction appelée.

En MIPS, on convient usuellement que les paramètres effectifs sont transmis par les registres `$a0` à `$a3` et que le résultat renvoyé est transmis par le registre `$v0`. *Notez que c'est cohérent avec les appels système vus ci-dessus.* Dans le cas où il y a plus de quatre paramètre effectifs, on utilise les registres `$a*` pour les quatre premiers paramètres, puis la pile pour les suivants.

Outre ce transfert de données, on a un transert *temporaire* de contrôle.

- Lors de l'appel, l'exécution *saute* au code de la fonction appelée.
- À la fin de l'appel, l'exécution *revient* au contexte appelant, en reprenant « juste après » le point où on a réalisé l'appel.

Ce transfert temporaire demande, au moment de l'appel, de mémoriser l'adresse de l'instruction à laquelle il faudra revenir après l'appel. En MIPS, on utilise pour cela le registre `$ra` (*Return Address*). Pour sauter au code de la fonction tout en mémorisant dans `$ra` l'adresse de l'instruction suivante du contexte appelant, on utilise

```
jal f
```

À la fin de l'appel de fonction, on rend alors la main à l'appelant avec au saut à l'adresse donnée par `$ra`.

```
jr $ra
```

Voici une traduction possible du fragment de code précédent, où les nombres à gauche donnent l'adresse de chaque instruction.

```
12  g:  li $a0, 6
16      jal f
20      move $a0, $v0
24      li $v0, 1
28      syscall

36  f:  li $v0, 7
40      mul $v0, $a0, $v0
44      jr $ra
```

Note : pour obtenir des adresse réalistes il faut ajouter à chacune la constante `0x400000`.

Voici une trace d'exécution de ce code assembleur.

instruction	$a0	$v0	$ra	pc	
				12	
`li $a0, 6`	6			16	
`jal f`	6		20	36	
`li $v0, 7`	6	7	20	40	
`mul $v0, $a0, $v0`	6	42	20	44	
`jr $ra`	6	42	20	20	
`move $a0, $v0`	42	42	20	24	
`li $v0, 1`	42	1	20	28	
`syscall`	42	1	20	32	affiche 42

Observons maintenant ce qui se passe si on ajoute un code principal réalisant un appel à notre fonction g. On complète le code comme suit.

```
00  main: jal g
04        li $v0, 10
08        syscall
12  g:    li $a0, 6
16        jal f
20        move $a0, $v0
24        li $v0, 1
28        syscall
32        jr $ra
36  f:    li $v0, 7
40        mul $v0, $a0, $v0
44        jr $ra
```

La trace d'exécution est maintenant la suivante.

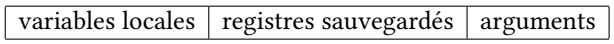
instruction	$a0	$v0	$ra	pc	
				0	
`jal g`			4	12	
`li $a0, 6`	6		4	16	
`jal f`	6		20	36	
`li $v0, 7`	6	7	20	40	
`mul $v0, $a0, $v0`	6	42	20	44	
`jr $ra`	6	42	20	20	
`move $a0, $v0`	42	42	20	24	
`li $v0, 1`	42	1	20	28	
`syscall`	42	1	20	32	affiche 42
`jr $ra`	42	1	20	20	
`move $a0, $v0`	1	1	20	24	
`li $v0, 1`	1	1	20	28	
`syscall`	1	1	20	32	affiche 1
`jr $ra`	1	1	20	20	
`move $a0, $v0`	1	1	20	24	
…	…	…	…	…	

Nous voilà coincés dans une boucle ! Problème : lorsque l'instruction `jal f` d'appel à `f` stocke son adresse de retour dans `$ra`, elle écrase la valeur qui était déjà présente dans `$ra`, à savoir l'adresse de retour de l'appel à g. Ainsi, lorsque l'on arrive à l'instruction `jr $ra` qui achève le code de g à l'adresse 32, le registre `$ra` ne contient plus la valeur qui avait été sauvegardée au moment de l'appel à g, et notre saut se fait vers la mauvaise cible.

Pile d'appels Lorsque des appels de fonction sont emboîtés, c'est-à-dire lorsque le code d'une fonction appelée réalise lui-même un appel de fonction et ainsi de suite, on se retrouve avec plusieurs appels actifs à un même moment. Plus précisément, l'appel le plus récent est en cours d'exécution et un certain nombre d'autres appels plus anciens sont en suspens.

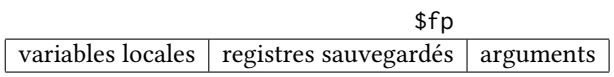
Chaque appel de fonction actif possède son propre contexte, contenant notamment les arguments qui lui ont été passés, les valeurs de ses variables locales, ou encore l'adresse de retour stockée dans `$ra`. Toutes ces informations doivent être stockées de manière à survivre à des appels de fonction internes. La structure utilisée pour stocker les informations

d'un appel donné est appelée ***tableau d'activation*** (en anglais *call frame*). Un tableau d'activation a la forme générale suivante :



où la partie « registres sauvegardés » contient en particulier la valeur sauvegardée du registre `$ra`.

Ces tableaux sont stockés dans la mémoire. On utilise un registre dédié `$fp` (*Frame Pointer*) pour localiser le tableau actif, c'est-à-dire le tableau d'activation de l'appel actuellement en cours d'exécution. On peut ensuite, à l'aide de `$fp`, accéder aux différents éléments stockés dans le tableau. Le pointeur de base `$fp` désigne l'adresse du dernier mot dédié aux registres sauvegardés.



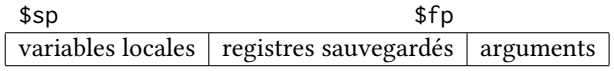
On accède donc aux arguments à partir de `$fp` avec un décalage positif, et aux variables locales avec un décalage négatif.

La vie des tableaux d'activation se déroule ainsi :

— création d'un tableau d'activation au début d'un appel,
— destruction du tableau à la fin de l'appel.

En outre, tout appel doit se terminer avant de rendre la main à son contexte appelant. Autrement dit, c'est toujours l'appel le plus récent qui terminera en premier, et toujours le tableau d'activation le plus récent qui sera détruit en premier. L'ensemble des tableaux d'activation a donc une structure de pile (*Last In First Out*), nommée ***pile d'appels*** (*call stack*). La pile d'appels est réalisée dans la pile MIPS elle-même, et en forme l'ossature principale.

Ainsi le premier mot du tableau d'activation correspond au sommet de la pile, désigné par le pointeur `$sp`.



En outre, pour faciliter l'identification du tableau d'activation précédent (qui est celui du contexte appelant), tout tableau d'activation a dans ses registres sauvegardés la valeur précédente du registre de base `$fp`, c'est-à-dire l'adresse qui permet d'accéder à ce tableau précédent.

Convention d'appel Pour assurer les bons transferts d'information et de contrôle, et la bonne gestion de la pile d'appels, le contexte appelant et la fonction appelée doivent respecter un protocole commun appelé ***convention d'appel***. Ce protocole se découpe en plusieurs étapes, à la charge de l'un ou l'autre des participants :

1. L'appelant, avant l'appel : prend en charge la passation des arguments via des registres et/ou la pile, puis déclenche l'appel avec `jal` pour stocker l'adresse de retour dans `$ra`.
2. L'appelé, au début de l'appel : réserve de l'espace sur la pile pour le tableau d'activation, y sauvegarde les registres qui doivent l'être, et donne à `$fp` sa nouvelle valeur.
3. Exécution du corps de la fonction appelée.
4. L'appelé, à la fin de l'appel : place le résultat dans un registre dédié, restaure les registres qui avaient été sauvegardés, désalloue le tableau d'activation, et redonne la main à l'appelant avec `jr $ra`.
5. L'appelant, après l'appel : retire les arguments placés sur la pile.

Le tableau d'activation est donc créé à l'étape 2 et détruit à l'étape 4 : il existe uniquement pendant l'exécution du corps de la fonction appelée.

Dans la convention standard de MIPS, les quatre premiers arguments sont passés par les registres `$a0` à `$a3` et les suivants éventuels par la pile, et le résultat est renvoyé par le registre `$v0`. Il est possible d'utiliser une convention différente, à condition de rester cohérent, et que tout appel respecte bien une même convention. *On utilisera effectivement une convention simplifiée pour compiler le langage LLIR, notamment en passant l'intégralité des arguments pas la pile (rappel : ce langage intermédiaire utilise uniquement une pile et un accumulateur).*

Dans la convention MIPS usuelle, on a également une subtilité au niveau de la sauvegarde des registres. Les registres sont découpés en deux familles :

— la sauvegarde des registres `$s*` est à la charge de l'appelé (*callee-saved*), de même que ce que nous avons déjà vu pour `$ra` et `$fp`,

— la sauvegarde des registres `$a*` et `$t*` est à la charge de l'appelant (*caller-saved*).

Autrement dit, dans l'étape 1 du protocole d'appel, l'appelant doit également sauvegarder les valeurs des registres `$a*` ou `$t*` dont il aurait encore besoin après l'appel (et il peut ensuite les restaurer à l'étape 5). Inversement, l'appelé n'a besoin à l'étape 2 de sauvegarder que les valeurs des registres `$ra`, `$fp` et `$s*`, et même plus précisément que les valeurs des registres qu'il modifiera. En conséquence, on utilise les registres `$t*` en priorité pour des valeurs intermédiaires à la durée de vie courte (pour ne pas avoir besoin de les sauvegarder avant de réaliser un appel). À l'inverse, les éléments que l'on souhaite préserver sur une durée plus longue seront avantageusement stockés dans les registres `$s*` : ils n'auront alors à être sauvegardés que lorsqu'une fonction appelée aura effectivement besoin d'utiliser ces mêmes registres. *À nouveau, cette subtilité sera invisible dans notre compilateur du langage LLIR, puisque ce dernier n'utilise presque pas de registres.*

Code MIPS pour une fonction récursive Imaginons une fonction factorielle, définie par un code de la forme suivante.

```
int fact(int n) {
  if (n > 0) {
    return fact(n-1) * n;
  } else {
    return 1;
  }
}
```

Après traduction en assembleur MIPS suivant les conventions précédentes, on obtient une fonction attendant un paramètre passé par le registre `$a0` et renvoyant un résultat via le registre `$v0`. Voici donc un fragment de code MIPS réalisant l'appel `fact(10)` et affichant le résultat.

```
        # Appel
00      li      $a0, 10
04      jal     fact
        # Affichage
08      move    $a0, $v0
12      li      $v0, 1
16      syscall
        # Fin du programme
20      li      $v0, 10
24      syscall
```

Le code de la fonction elle-même est construit comme suit. D'abord, on réserve de l'espace sur la pile pour le tableau d'activation, dans lequel on sauvegarde les registres `$ra` et `$fp`. Notez que l'on a prévu la place dans le tableau d'activation pour un troisième élément, qui servira plus tard.

```
    fact:
        # Tableau d'activation
28      sw      $fp, -4($sp)
32      sw      $ra, -8($sp)
36      addi    $fp, $sp, -4
40      addi    $sp, $sp, -12
```

Ensuite vient le corps de la fonction à proprement parler, commençant par un test. Si le test est positif, on saute au fragment de code correspondant à la branche « then ». Sinon on poursuit avec l'instruction suivante, correspondant à la branche « else ».

```
        # Test
44      bgtz    $a0, fact_rec
48      li      $v0, 1
52      b       fact_end
  fact_rec:
```

Dans la branche « then » il faut réaliser un appel récursif à `fact`, avec un argument décrémenté de 1. Or la valeur de l'argument servira encore après l'appel : il faudra la multiplier avec le résultat de l'appel. On sauvegarde donc cette valeur avant de préparer l'appel récursif.

```
        # Appel recursif
56      sw      $a0, -8($fp)
60      addi    $a0, $a0, -1
64      jal     fact
```

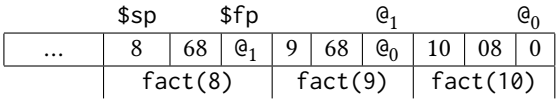
Après l'appel, on récupère donc la valeur sauvegardée pour réaliser la multiplication.

```
68      lw      $t0, -8($fp)
72      mul     $v0, $v0, $t0
```

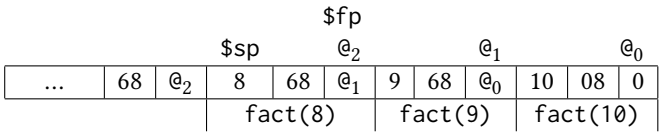
Enfin, on ajoute une section finale s'occupant de conclure l'appel de fonction. Notez que l'on arrive à cette section soit après la branche « then », soit après la branche « else », mais que dans un cas comme dans l'autre le résultat a déjà été placé dans `$v0`. Il ne reste donc ici qu'à restaurer les registres `$fp` et `$ra` et détruire le tableau d'activation.

```
  fact_end:
        # Nettoyage
76      addi    $sp, $fp, 4
80      lw      $fp, -4($sp)
84      lw      $ra, -8($sp)
88      jr      $ra
```

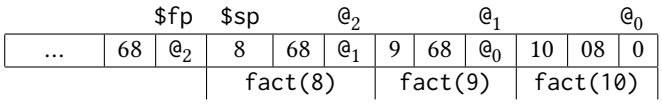
En abordant la ligne 28 au début de l'appel récursif `fact(7)`, la pile d'appels a la forme suivante :



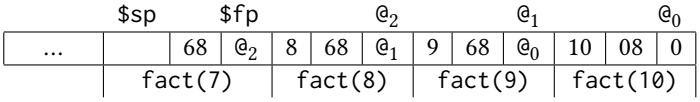
Les lignes 28 et 32 écrivent les valeurs courantes de `$fp` et `$ra` juste au-delà du sommet de la pile.



La ligne 36 déplace le pointeur `$fp` juste au-delà du sommet de la pile.



Enfin, la ligne 40 complète la création du tableau d'activation en déplaçant le sommet de pile. La case destinée à recevoir la valeur de `$a0` est présente, mais n'est pas encore initialisée.



Les lignes 76 à 84 effectuent l'opération inverse et nous ramènent à la configuration initiale.

Appels terminaux Regardons le cas où une fonction termine par un appel à une autre fonction (on parle d'un ***appel terminal***).

```
int f() {
  return g(3);
}
```

Cette fonction `f` n'a plus rien à faire après l'appel à `g`, à part nettoyer son propre tableau d'activation. Le tableau d'activation de l'appel à `f` n'est donc plus utile au moment de l'appel à `g`, et on pourrait imaginer le détruire *avant* l'appel. En outre, la valeur de `$ra` sauvegardée dans le tableau d'activation de l'appel à `f`, qui est restaurée lors de la destruction du tableau d'activation, est alors précisément l'adresse à laquelle il faudra retourner après l'appel à `g`. On peut donc réaliser l'appel avec un simple saut, c'est-à-dire une instruction `j` plutôt

que `jal`. Et pour limiter encore les manipulations on peut, au lieu de détruire le tableau d'activation de l'appel à `f`, le réutiliser pour l'appel à g (il faut affiner les détails si les deux fonctions utilisent des tableaux d'activation de tailles différentes).

Cette optimisation des appels terminaux a un effet particulièrement spectaculaire lorsqu'on l'applique à une fonction récursive.

```
int fact(int n) {
  return fact_aux(n, 1);
}

int fact_aux(int n, int acc) {
  if (n <= 0) {
    return acc;
  } else {
    return fact_aux(n-1, n*acc);
  }
}
```

On conserve les lignes 00 à 24 à l'identique pour appeler `fact` et afficher le résultat. Le reste va être simplifié drastiquement.

La fonction `fact` réalise un unique appel terminal à la fonction `fact_aux`. Il n'y a pas de tableau de tableau d'activation à créer, le premier argument est déjà dans `$a0`, il suffit de placer le deuxième argument dans `$a1` puis sauter à la fonction auxiliaire.

```
    fact:
28'     li      $a1, 1
32'     j       fact_aux
```

Notez que l'on utilise l'instruction de saut `j` et pas l'instruction d'appel `jal`, puisque `$ra` a déjà la bonne valeur.

Vient ensuite la fonction auxiliaire `fact_aux`. Celle-ci n'a pas non plus besoin de tableau d'activation, puisqu'elle n'a pas de variables locales et n'opère que des appels terminaux à elle-même. On réalise donc directement le test. S'il est positif on saute à la branche « then », qui sera le cas d'arrêt.

```
    fact_aux:
36'     blez    $a0, fact_aux_end
```

Sinon on poursuit avec l'appel récursif terminal, pour lequel il suffit de placer les bonnes valeurs dans `$a0` et `$a1`.

```
40'     mul     $a1, $a1, $a0
44'     addi    $a0, $a0, -1
48'     j       fact_aux
```

Enfin la fonction termine en plaçant le résultat dans `$v0` et en revenant, enfin, à l'adresse de retour enregistrée lors de l'appel d'origine à `fact`, ligne 4.

```
    fact_aux_end:
52'     move    $v0, $a1
56'     jr      $ra
```

Le code obtenu est beaucoup plus simple, et utilise exclusivement les registres. À noter : il est même identique à ce qu'on aurait pu obtenir en compilant un code impératif basé sur une boucle comme la suivante.

```
int res = 1;
while (n > 0) {
  res = n*res;
  n--;
}
```

6.8 Traduction LLIR vers MIPS

Reprenons maintenant notre langage LLIR, pour le traduire en assembleur MIPS. La combinaison de cette traduction avec la précédente fournira donc un compilateur de IMP vers MIPS.

On suit la stratégie générale suivante :

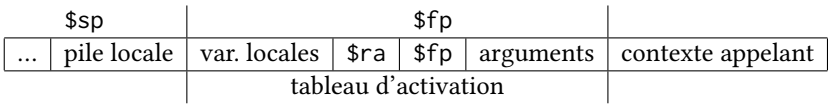
— L'unique registre de LLIR est matérialisé par le registre `$t0` en MIPS. On appelle ce registre l'accumulateur.
— La pile LLIR est réalisée avec la pile MIPS et le pointeur `$sp`.

On utilisera très peu d'autres registres MIPS : un registre auxiliaire `$t1` pour les opérations arithmétiques et certaines opérations mémoire, plus les registres spéciaux `$sp`, `$fp` et `$ra` (le registre `pc` est toujours utilisé également, mais on ne le manipule pas explicitement).

LLIR étant déjà bas niveau, la plupart des instructions LLIR se traduisent en seulement une ou deux instructions MIPS. Le plus gros travail concerne la réalisation des fonctions et des appels. Pour cela, on va se laisser guider par les conventions vues pour MIPS, avec de petites simplifications reflétant l'usage qui était fait dans LLIR de la pile et de l'accumulateur.

— Tous les arguments sont placés sur la pile, avec le premier argument au sommet (repris de LLIR).
— Le résultat d'un appel est placé dans l'accumulateur `$t0` (repris de LLIR).
— À charge pour l'appelé de sauvegarder et restaurer `$fp` et `$ra` (MIPS standard).
— Les autres registres n'étant pas utilisés, on n'a jamais besoin de les sauvegarder ou de les restaurer au moment des appels.

Lors d'un appel de fonction la pile aura donc la forme suivante.



On utilise toujours le registre `$fp` pour l'adresse de référence du tableau d'activation de l'appel en cours. Au-dessus de cette adresse de référence on trouve l'intégralité des paramètres effectifs (et pas juste les paramètres à partir du cinquième). À l'adresse de référence et à l'adresse immédiatement inférieure on stocke l'ancienne valeur du pointeur `$fp` et l'adresse de retour `$ra`. Sous ces deux mots, le tableau d'activation est complété par une zone dédiée aux variables locales. En dessous du tableau d'activation, on se sert de la pile MIPS et de son pointeur `$sp` pour réaliser la pile locale de l'appel de fonction LLIR.

Module auxiliaire MIPS Pour représenter le code MIPS, on se donne un type minimal permettant de représenter efficacement des concaténations de chaînes de caractères.

```
type asm =
  | Nop
  | S of string
  | C of asm * asm

let (@@) x y = C (x, y)

type program = { text: asm; data: asm; }
```

Ainsi, pour « concaténer » deux fragments a_1 et a_2 de code assembleur, il suffit de former la structure `C(`a_1`,` a_2`)`, ce que l'on peut abréger avec la notation a_1 `@@` a_2. C'est plus efficace qu'une vraie concaténation, qui aurait un coût proportionnel à la longueur des chaînes manipulées.

Pour faciliter la production de code MIPS sous cette représentation, on définit en plus de cela quelques constantes pour les registres utilisés

```
let t0 = "$t0"
let t1 = "$t1"
let ra = "$ra"
let sp = "$sp"
let fp = "$fp"
```

ainsi qu'une série de fonctions auxiliaires produisant des instructions MIPS. L'objectif est que le code caml produisant une représentation de code assembleur MIPS soit aussi proche que possible du code MIPS représenté. On pourra ainsi écrire en caml dans du code caml l'expression `add t0 t0 t1` pour inclure la chaîne de caractères `"  add $t0, $t0, $t1\n"` dans un élément de type `asm`.

Voici les premières de ces fonctions.

```
open Printf
let li    r1 i      = S(sprintf "  li   %s, %i"     r  i)
let move  r1 r2     = S(sprintf "  move %s, %s"     r1 r2)

let add   r1 r2 r3  = S(sprintf "  add  %s, %s, %s" r1 r2 r3)
let addi  r1 r2 i   = S(sprintf "  addi %s, %s, %d" r1 r2 i)
let mul   r1 r2 r3  = S(sprintf "  mul  %s, %s, %s" r1 r2 r3)
let slt   r1 r2 r3  = S(sprintf "  slt  %s, %s, %s" r1 r2 r3)

let j     l         = S(sprintf "  j    %s"         l)
let jal   l         = S(sprintf "  jal  %s"         l)
let jr    r1        = S(sprintf "  jr   %s"         r1)
let bnez  r1 l      = S(sprintf "  bnez %s, %s"     r1 l)
let bltz  r1 l      = S(sprintf "  bltz %s, %s"     r1 l)
let bge   r1 r2 l   = S(sprintf "  bge  %s, %s, %s" r1 r2 l)

let syscall         = S("  syscall")
let nop             = Nop
let label l         = S(sprintf "%s:" l)
let comment s       = S(sprintf "  # %s" s)
```

Pour refléter la possibilité des instructions MIPS de la famille de `lw` et `sw` de décoder des adresses sous différents formats, on introduit également une distinction entre deux formats utiles : une adresse donnée directement par une étiquette, et une adresse calculée par un décalage à partir de l'adresse donnée par un registre.

```
type address =
  | L of string
  | A of int * string

let (++) i r = A(i, r)

let address = function
  | L l     -> l
  | A(i, r) -> sprintf "%i(%s)" i r

let lw   r1 a     = S(sprintf "  lw   %s, %s" r1 (address a))
let sw   r1 a     = S(sprintf "  sw   %s, %s" r1 (address a))
let lbu  r1 a     = S(sprintf "  lbu  %s, %s" r1 (address a))
```

Pour traiter les déclarations de données statiques, on introduit encore une les fonctions auxiliaires suivantes.

```
let ilist = function
  | [] -> ""
  | [i] -> sprintf "%i" i
  | i :: l -> sprintf "%i, %s" i (ilist l)
let dword l  = S(sprintf "  .word %s" (ilist l))
let asciiz s = S(sprintf "  .asciiz %s" s)
```

Cette batterie de fonctions auxiliaires donne une manière fluide de générer en caml une représentation de code MIPS. Par exemple, le code MIPS ci-dessous à gauche peut être défini par l'expression caml à droite.

```
  # built-in atoi
atoi:
  li    $v0, 0
  li    $t1, 10

atoi_loop:
  lbu   $t0, 0($a0)
  beqz  $t0, atoi_end
  addi  $t0, $t0, -48
  bltz  $t0, atoi_error
  bge   $t0, $t1 atoi_error
  mul   $v0, $v0, $t1
  add   $v0, $v0, $t0
  addi  $a0, $a0, 1
  j     atoi_loop

atoi_error:
  li    $v0, 10
  syscall

atoi_end:
  jr    $ra
```

```
let built_ins =
  comment "built-in atoi"
  @@ label "atoi"
  @@ li    v0 0
  @@ li    t1 10

  @@ label "atoi_loop"
  @@ lbu   t0 (0++a0)
  @@ beqz  t0 "atoi_end"
  @@ addi  t0 t0 (-48)
  @@ bltz  t0 "atoi_error"
  @@ bge   t0 t1 "atoi_error"
  @@ mul   v0 v0 t1
  @@ add   v0 v0 t0
  @@ addi  a0 a0 1
  @@ j     "atoi_loop"

  @@ label "atoi_error"
  @@ li    v0 10
  @@ syscall

  @@ label "atoi_end"
  @@ jr    ra
```

Pour finir, on ajoute deux fonctions `push` et `pop` produisant des séquences de code MIPS facilitant la manipulation de la pile.

- `push` *r* génère du code MIPS qui empile le contenu du registre *r*, et
- `pop` *r* génère du code MIPS qui retire l'élément au sommet de la pile et le place dans le registre *r*.

Ces deux éléments n'existent pas directement dans les instructions ou pseudo-instructions MIPS. Via l'utilisation de notre module MIPS on pourra les voir comme des *pseudo*-pseudo-instructions.

```
let push r =
  addi sp sp (-4) @@ sw r (0++sp)
let pop r =
  lw r (0++sp) @@ addi sp sp 4
```

Cette représentation caml des programmes MIPS étant fixée, une simple fonction permet ensuite d'écrire l'ensemble d'un programme MIPS vers une destination cible, par exemple un fichier.

```
let rec print_asm fmt a =
  match a with
  | Nop -> ()
  | S s -> fprintf fmt "%s\n" s
  | C (a1, a2) ->
     let () = print_asm fmt a1 in
     print_asm fmt a2

let print_program fmt p =
  fprintf fmt ".text\n";
  print_asm fmt p.text;
  fprintf fmt ".data\n";
  print_asm fmt p.data
```

Traduction de LLIR vers MIPS La fonction de traduction principale `tr_prog` s'applique à un programme LLIR `prog`, et renvoie un programme MIPS. La fonction de traduction des instructions `tr_instr` est définie comme fonction interne (elle aura besoin d'accéder à la liste `prog.functions`).

```
open Llir
open Mips

let tr_prog prog =
  let tr_instr = function
    | Cst n -> li t0 n
```

Rappel : LLIR place les valeurs calculées dans un accumulateur (le registre `$t0`), et les résultats intermédiaires sur la pile. Pour tout ce qui concerne l'arithmétique on utilise donc principalement ce registre et les deux opérations auxiliaires `push` et `pop`. Juste avant de réaliser une opération binaire il faut récupérer la valeur du premier opérande, qui a été stockée sur la pile une fois calculée. On utilise pour cela l'opération `pop`, et un deuxième registre de travail `$t1`.

```
      | Push  -> push t0
      | Add   -> pop t1 @@ add t0 t1 t0
      | Mul   -> pop t1 @@ mul t0 t1 t0
      | Lt    -> pop t1 @@ slt t0 t1 t0
```

Un accès à une variable en lecture (`Get`) ou en écriture (`Set`) est directement traduit par une instruction `lw` ou `sw`. Il faut cependant calculer la localisation de cette variable, qui varie en fonction de sa nature :

— les variables globales sont allouées dans le segment des données statiques, avec une étiquette à leur nom,
— les variables locales sont stockées dans le tableau d'activation, avec un décalage négatif par rapport à l'adresse de base `$fp`,
— les paramètres d'un appel de fonction sont stockés dans le tableau d'activation, avec un décalage positif par rapport à l'adresse de base `$fp`.

Une fonction auxiliaire `loc` produit ces localisations

```
  let loc = function
    | Global x -> (L x)
    | Param i  -> (4*(i+1))++fp
    | Local i  -> (-4*(i+2))++fp
```

et peut être directement utilisée dans les cas correspondants de la fonction `tr_instr`.

```
      | Get(v) -> lw t0 (loc v)
      | Set(v) -> sw t0 (loc v)
```

Les sauts LLIR sont traduits par des sauts MIPS. Pour représenter un branchement conditionnel à deux branches tel que je `CJump` de LLIR, on utilise en MIPS un saut conditionnel vers la première branche, suivi d'un saut vers la deuxième branche (exécuté si le premier branchement n'a pas été sélectionné).

```
      | Jump lab -> j lab
      | CJump(lab1, lab2) -> bnez t0 lab1 @@ j lab2
```

L'instruction d'affichage `putchar` est traduite par un appel système. Pour respecter les règles de `syscall` il faut donc transférer la valeur à afficher de l'accumulateur `$t0` vers le registre `$a0`.

```
      | Putchar -> move a0 t0 @@ li v0 11 @@ syscall
```

Dans la traduction d'un appel de fonction `Call`, il faut inclure les étapes du protocole d'appel qui sont à la charge de l'appelant. En l'occurrence il n'y a plus rien à faire avant l'appel (on suppose que les arguments ont déjà été placés au sommet de la pile), mais il faut encore se charger de nettoyer la pile après l'appel. Pour cela on consulte la définition de la fonction appelée pour connaître le nombre d'arguments à retirer de la pile.

```
      | Call f ->
        let fdef = List.find (fun fdef -> fdef.name = f) prog.functions in
        let n = fdef.nb_params in
        jal f
        @@ addi sp sp (4*n)
```

L'instruction `return` appartient au domaine de l'appelé. C'est à ce niveau que l'on prend en charge la partie du protocole d'appel devant être réalisée par l'appelé, à la fin de l'appel. Notez qu'à ce stade le résultat est déjà dans l'accumulateur `$t0`, il ne reste donc qu'à désallouer le tableau d'activation et à restaurer les registres `$ra` et `$fp`.

```
      | Return ->
        addi sp fp (-4)
        @@ pop ra
```

```
      @@ pop fp
      @@ jr ra
  in
```

Pour traduire une séquence d'instructions, il suffit d'itérer la fonction précédente.

```
  let rec tr_seq s =
    List.fold_right (fun i asm -> tr_instr i @@ asm) s nop
  in
```

La traduction d'une fonction LLIR contient deux aspects :

— il faut traduire l'ensemble des blocs composant le code de cette fonction, et
— il faut ajouter un code d'introduction qui réalise la partie du protocole d'appel qui est à la charge de l'appelé, au début de l'appel, pour ensuite passer la main à la première « vraie » instruction de la fonction.

Du côté de la convention d'appel, on a besoin de sauvegarder `$fp` et `$ra`, définir la nouvelle valeur de `$fp` et allouer le tableau d'activation. Cette partie est précédée de l'étiquette de la fonction, qui sera utilisée comme cible de saut lors d'un appel, et suivie du saut vers la première instruction du corps de la fonction.

```
  let tr_fdef fdef =
    label fdef.name
    @@ push fp
    @@ push ra
    @@ addi fp sp 4
    @@ addi sp sp (-4*fdef.nb_locals)
    @@ j fdef.start
```

On peut ensuite placer dans un ordre arbitraire l'ensemble des blocs composant la fonction, chacun précédé de son étiquette.

```
    @@ Hashtbl.fold
      (fun name seq code -> code @@ label name @@ tr_seq seq)
      fdef.code nop
  in
```

On obtient le texte complet du programme en combinant le code de chacune des fonctions, y compris la fonction prédéfinie `atoi` vue ci-dessus, avec un bloc de code principal qui :

— regarde si le paramètre `arg` de la fonction `main` a reçu une valeur sur la ligne de commande, et dans ce cas le traduit en un entier à l'aide de notre fonction MIPS auxiliaire `atoi`,
— appelle la fonction `main`,
— termine l'exécution du programme une fois l'appel à `main` achevé.

```
  let text =
       beqz a0 "init_end"
    @@ lw   a0 0 a1
    @@ jal  "atoi"
    @@ sw   v0 (0++sp)
    @@ addi sp sp (-4)
    @@ label "init_end"
    @@ jal  "main"
    @@ li   v0 10
    @@ syscall

    @@ List.fold_right
        (fun fdef asm -> tr_fdef fdef @@ asm)
        prog.functions
        nop

    @@ built_ins
  in
```

Ne reste plus qu'à y ajouter les déclarations des variables globales, allouées dans les données statiques et initialisées avec la valeur `0`.

```
  let data =
    List.fold_right
      (fun x asm -> label x @@ dword [0] @@ asm)
      prog.globals
      nop
  in

  { text; data }
```

6.9 Tableaux

Jusqu'ici, nous n'avons compilé que de petits langages de programmation, dans lesquels les données manipulées étaient limitées à des nombres entiers. L'objectif de la fin de ce chapitre est d'y ajouter des structures de données plus riches.

Les données, selon leur nature, peuvent être représentées de différentes manières en mémoire :

— un nombre entier ou un booléen est représenté par un unique mot mémoire (par exemple de 4 octets),
— une donnée plus complexe comme un tableau, un *n*-uplet, une structure, un objet ou une fermeture sera en revanche représenté par plusieurs mots, formant généralement un *bloc*, c'est-à-dire une suite de mots consécutifs en mémoire,
— une donnée plus riche encore comme une liste chaînée, un arbre ou une table de hachage pourra même être représentée par plusieurs blocs, non consécutifs en mémoire.

Tableau alloué statiquement Considérons le cas d'un tableau global, dont la taille est explicitement donnée par le texte du programme source. Dans un langage comme C on pourrait par exemple introduire ainsi une variable globale `tab` représentant un tableau de cinq entiers.

```
int tab[5];
void main(int x) { ... }
```

Comme les autres variables globales vues jusqu'ici, ce tableau peut être placé en mémoire dans les données statiques. En revanche, au lieu de lui allouer un unique mot mémoire, on en donne cinq consécutifs.

```
        .data
tab:    .word  0 0 0 0 0
```

Le premier de ces mots est à l'adresse donnée par l'étiquette `tab`, le suivant à l'adresse `tab+4`, et le dernier à l'adresse `tab+16`. Notez que l'on ne peut faire une telle déclaration que si l'on connaît explicitement la taille du tableau.

L'accès à une case d'un tel tableau se fait donc en partant de l'étiquette `tab`, qui donne une adresse de base, et en ajoutant un décalage correspondant à l'indice de la case cherchée (le décalage étant un multiple de 4 octets).

Considérons par exemple le tableau introduit par

```
        .data
tab:    .word  1 1 2 5 14 42
```

On accède au troisième élément à l'aide d'un décalage de 8 par rapport à l'adresse de base donnée par `tab` :

```
        la   $t0, tab
        lw   $t1, 8($t0)
```

Dans cette première version l'indice est connu statiquement : on peut calculer le décalage en amont, et l'écrire explicitement dans le code. Si en revanche l'indice auquel chercher est donné par un registre, il va falloir ajouter un peu de code MIPS pour calculer à l'exécution le bon décalage. Pour accéder à une case dont l'indice est donné par le registre `$a0`, il faut donc multiplier la valeur de `$a0` par 4 pour obtenir le bon décalage (on le fait efficacement par un décalage de 2 bits vers la gauche, avec l'instruction `sll`), puis ajouter ce décalage à l'adresse de base.

```
        la   $t0, tab
        sll  $a0, 2
        add  $t0, $t0, $a0
        lw   $t1, 0($t0)
```

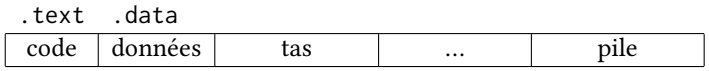
Lors de l'instruction `lw` on n'indique donc plus de décalage par rapport à l'adresse donnée par `$t0`, puisque que celui-ci contient déjà précisément l'adresse voulue. *Note : le décalage donné dans l'instruction* `lw` *ne peut être qu'une constante entière explicite, on ne peut pas y mettre* `$a0`.

Tableaux alloués dynamiquement Imaginons cette fois le cas d'un tableau créé dynamiquement, avec une taille qui n'est pas connue explicitement à l'avance.

```
int f(int n) {
  int tab[n];
  ...
}
```

C'est beaucoup plus souple pour le programmeur, mais demande une nouvelle stratégie pour le compilateur. On peut imaginer que ce tableau soit représenté par n mots consécutifs en mémoire, rangés sur la pile avec les autres variables locales. C'est effectivement ce qui se passe en C, par défaut. Attention cependant : comme tout ce qui se trouve sur la pile, ce tableau a une durée de vie limitée et sera détruit à la fin de l'appel.

Nous allons nous concentrer ici sur une troisième stratégie, permettant d'allouer dynamiquement des structures de données *persistantes*, qui vont survivre à l'appel de fonction qui les a créées. On utilise pour cela une nouvelle région de la mémoire : le *tas*, qui se trouve entre les données statiques et la pile.



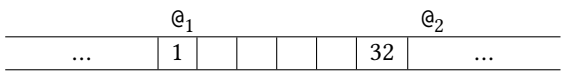
La plus petite adresse du tas est immédiatement au-dessus de la zone des données statiques. Un pointeur `brk` appelé *memory break* identifie la fin du tas, ou plus précisément la première adresse au-delà du tas. Ainsi le dernier mot du tas est à l'adresse `brk-4`. L'espace situé entre le tas et la pile, c'est-à-dire entre les pointeurs `brk` (inclus) et `$sp` (exclu), est vide. Lorsque l'on a besoin de plus de place sur le tas, on incrémente le pointeur `brk`, pour absorber une partie de l'espace libre entre le tas et la pile.

Le pointeur `brk` est géré par le système d'exploitation, aussi son déplacement est fait par un appel système `sbrk`, qui a le code 9 dans notre simulateur MIPS. Cet appel système prend comme argument, via `$a0`, le nombre d'octets dont `brk` doit être incrémenté. Il renvoie comme résultat, via `$v0`, l'ancienne valeur de `brk`, c'est-à-dire la première adresse de la zone qui vient d'être ajoutée au tas.

Voici un exemple d'utilisation de ce mécanisme, dans lequel on alloue sur le tas de l'espace pour un tableau de 6 cases (24 octets), avant d'écrire `1` dans la première case et `32` dans la sixième.

instruction	`$a0`	`$v0`	`$t0`	`brk`
				$0x10040000$
`li $a0, 24`	24			$0x10040000$
`li $v0, 9`	24	9		$0x10040000$
`syscall`	24	$0x10040000$		$0x10040018$
`li $t0, 1`	24	$0x10040000$	1	$0x10040018$
`sw $t0, 0($v0)`	24	$0x10040000$	1	$0x10040018$
`li $t0, 32`	24	$0x10040000$	32	$0x10040018$
`li $t0, 20($v0)`	24	$0x10040000$	32	$0x10040018$

La forme du tas associée à cet exemple est la suivante, où l'adresse $@_1$ est la position d'origine de `brk` (0x10040000) et $@_2$ est la nouvelle position après appel à `sbrk` (0x10040018).



Extension de LLIR Pour l'instant, ni notre langage source IMP ni notre représentation intermédiaire LLIR ne contiennent les ingrédients nécessaires pour la manipulation des tableaux. On peut cependant étendre légèrement ces deux langages et les schémas de compilation associés pour obtenir une version améliorée de IMP et de son compilateur.

On ajoute au langage LLIR trois instructions, pour la lecture et l'écriture en mémoire, ainsi que pour l'extension du tas.

— Lecture en mémoire : l'instruction

```
read
```

lit la valeur à l'adresse mémoire donnée par le registre, et place la valeur lue dans le registre. On peut la traduire en MIPS par l'unique instruction

```
lw    $t0, 0($t0)
```

Rappel : dans notre traduction de LLIR vers MIPS, l'unique registre LLIR est réalisé par le registre MIPS `$t0`.

— Écriture en mémoire : l'instruction

```
write
```

écrit la valeur du registre à l'adresse mémoire prélevée au sommet de la pile. On la traduit en MIPS par une séquence retirant l'élément au sommet de la pile puis écrivant la valeur du registre `$t0` à l'adresse ainsi obtenue.

```
lw    $t1, 0($sp)
addi  $sp, $sp, 4
sw    $t0, 0($t1)
```

Rappel : dans notre traduction de LLIR vers MIPS, on utilise le registre `$t1` pour travailler sur une valeur extraite de la pile.

— Extension du tas : l'instruction

```
sbrk
```

étend le tas d'un nombre d'octets donné par le registre. Le résultat de l'appel `sbrk`, qui est la première adresse de la zone ajoutée au tas, est placé dans le registre. On traduit cette instruction en MIPS par une séquence effectuant un appel système `sbrk` puis déplaçant son résultat du registre `$v0` où le résultat est placé vers le registre `$t0` où le résultat est attendu.

```
move $a0, $t0
li   $v0, 9
syscall
move $t0, $v0
```

Extension de IMP avec des pointeurs : PIMP On étend la syntaxe abstraite de IMP avec des constructions correspondant directement aux trois nouvelles instructions LLIR : deux nouvelles formes d'expressions pour la lecture en mémoire et l'extension du tas, et une nouvelle forme d'instruction pour l'écriture en mémoire.

```
type expr =
  ...
  | Read of expr
  | Sbrk of expr

type instr =
  ...
  | Write of expr * expr
```

La traduction de ces nouveaux éléments en LLIR est similaire à ce que l'on a déjà vu pour la traduction des opérations arithmétiques ou de l'instruction `putchar`.

— Une expression `Read` *e* est traduite en LLIR par :
  — le code évaluant *e*, qui place la valeur obtenue dans le registre, suivi de
  — l'instruction `read`.
— Une expression `Sbrk` *e* est traduite de la même façon, avec l'instruction `sbrk`.

— Une instruction `Write(`e_1`, `e_2`)` est traduite en LLIR par :
  — le code évaluant e_1, puis
  — l'instruction `push`, suivie par
  — le code évaluant e_2, et enfin
  — l'instruction `write`.

On étend enfin la syntaxe concrète du langage IMP et les analyseurs associés pour permettre l'utilisation en pratique de ces nouveaux éléments. On introduit les nouvelles formes concrètes suivantes

— `sbrk(1024)` pour l'extension du tas d'un certain nombre d'octets,
— `*p` pour l'accès à l'adresse donnée par un pointeur,
— `t[k]` pour l'accès à une case d'un tableau.

L'accès à une adresse peut être utilisé aussi bien en lecture qu'en écriture. L'analyse syntaxique des expressions va donc reconnaître la forme

```
*p
```

et la traduire sous une forme

```
Read p
```

et l'analyse syntaxique des instructions va reconnaître la nouvelle forme

```
*p = e;
```

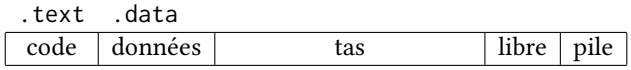
et la traduire sous une forme

```
Write(p, e)
```

La forme concrète `t[k]` n'est en revanche qu'un sucre syntaxique pour `*(t+4k)`. Ainsi une expression de la forme `t[k]` est traduite en la forme `Read(Add(t, Mul(k, Cst 4)))` et une instruction de la forme `t[k] = e` est traduite en la forme `Write(Add(t, Mul(k, Cst 4)), e)`.

6.10 Gestion dynamique de la mémoire

Nous avons évoqué avec les tableaux l'utilisation d'une nouvelle région de la mémoire, appelée le *tas*, que nous allons maintenant détailler.



Le tas est une région de la mémoire qui est adaptée au stockage des données persistantes. Cette région est gérée par un programme dédié qui permet en particulier de :

— demander de la mémoire pour stocker de nouvelles données,
— libérer la mémoire qui avait été affectée à des données qui ne sont plus utiles, pour qu'elle puisse être à nouveau utilisée pour d'autres données.

Ces deux opérations sont qualifiées de *dynamiques*, car elles ont lieu *pendant l'exécution* du programme principal.

Selon les cas, le programme dédié à la gestion de la mémoire peut prendre différentes formes.

— En C, on utilise la bibliothèque `malloc`, et notamment les fonctions `malloc` et `free` pour allouer ou libérer manuellement de la mémoire.
— En caml, java, python, un gestionnaire automatique appelé *GC* (*garbage collector*, ou *glaneur de cellules*) est intégré à l'environnement d'exécution. Il tourne en parallèle du programme principal et identifie puis libère les parties du tas qui ne sont plus utiles.

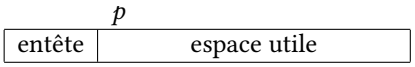
Nous allons détailler ici le fonctionnement des opérations manuelles `malloc` et `free`, qui peuvent ensuite servir de base à la réalisation d'un gestionnaire automatique.

On veut donc deux opérations `malloc` et `free` répondant aux spécifications suivantes.

— Un appel `malloc(`n`)`, avec n un entier positif, trouve et réserve au moins n octets consécutifs dans le tas, et renvoie un pointeur vers le premier octet réservé. Cette fonction ne peut réserver qu'une zone de mémoire qui était libre jusque là. Si la fonction ne trouve pas de bloc mémoire libre assez grand elle peut soit étendre le tas (en absorbant une partie de l'espace libre situé entre le tas et la pile) soit échouer.
— Un appel `free(`p`)` libère le bloc de mémoire à l'adresse donnée par le pointeur p. Un tel appel suppose que le pointeur p cible effectivement un bloc précédemment alloué par `malloc` (sinon, l'effet est indéterminé).

Nous allons d'abord détailler ce qu'est un *bloc mémoire*, puis comment réaliser `malloc` et `free`.

Blocs mémoire Les blocs sont l'unité de gestion de la mémoire dynamique. Un bloc est une zone de mémoire contiguë, qui peut être libre ou utilisée. Chaque bloc est constitué d'un entête et d'un espace utile.



L'espace utile est la partie effectivement utilisée pour stocker des données, tandis que l'entête contient des informations additionnelles utilisées exclusivement par le gestionnaire de mémoire. On accède au contenu d'un bloc à l'aide d'un pointeur p, qui donne l'adresse du premier octet utile. On a couramment des contraintes sur l'adresse désignée par le pointeur p, par exemple qu'il s'agisse d'un multiple de 8 (on le supposera effectivement dans la suite).

En supposant que l'entête soit représenté par un mot mémoire, et chaque donnée stockée dans l'espace utile soit également représentée par un mot mémoire, on peut donc accèder à l'entête avec `p[-1]` et au mot d'indice i avec `p[`i`]`.

L'entête d'un bloc doit contenir les informations utiles au gestionnaire de mémoire. Les deux principales sont :

— le statut *libre* ou *réservé* du bloc, correspondant à un bit d'information valant `1` pour un bloc réservé et `0` pour un bloc libre,

— la *taille* totale du bloc (espace utile + entête), exprimée par un nombre entier d'octets.

Si l'on suppose que la taille d'un bloc est un multiple de 8, cette dernière est représentée par un nombre dont les trois derniers bits ne donnent pas d'information (pour un multiple de 8 ces trois bits valent nécessairement `0`). On peut alors enregistrer à la fois la taille t et le statut s dans un unique mot mémoire e, en plaçant le bit de statut à la place du dernier bit de taille. On peut réaliser ceci à l'aide d'une opération de « ou » bit à bit : $e = t$ `|` s.

t	s	e
32	libre	$0b100000$
24	réservé	$0b011001$

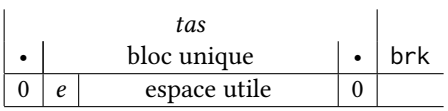
On peut à l'inverse récupérer le statut ou la taille à partir d'un mot d'entête e à l'aide d'opérations de masquage :

— le statut est donné par le dernier bit : $s = e$ `& 0x1`,

— la taille est obtenue en masquant les trois derniers bits : $t = e$ `& ~0x7`.

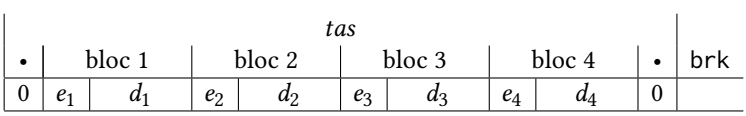
Initialisation On considère à partir de maintenant que l'ensemble du tas est formé de blocs consécutifs en mémoire, sans espace libre entre les blocs (éventuellement cela peut nécessiter de donner un peu plus d'espace utile que demandé à certains blocs, pour bien toujours tomber sur des multiples de 8 pour les adresses).

Pour initialiser un tel tas, on demande au système une zone de mémoire à l'aide d'un appel `sbrk`. Pour marquer les extrémités de cette zone, on laisse une valeur spéciale (en l'occurrence zéro) dans le premier mot et le dernier mot. Tout le reste forme un grand bloc.



Si on a alloué 1024 octets pour l'ensemble du tas, une fois sacrifiés les premier et dernier mots il reste 1016 octets pour le bloc : 4 octets d'entête et 1012 octets utiles. Ce bloc est vide, son bit de statut vaut donc 0. Ainsi l'entête e vaut 1016. Notez également que le pointeur p vers l'espace utile de cet unique bloc est un multiple de 8, puisque cette adresse se situe deux mots après le début du tas (lui-même à l'adresse `0x1004000` en MIPS).

Lorsque le tas est étendu par un nouvel appel à `sbrk`, on déplace le zéro final pour préserver la forme générale : une séquence de blocs consécutifs, donc les extrémités sont marquées par deux mots nuls.



Fonction `malloc` Considérons un appel `malloc(`n`)`. La fonction `malloc` doit rechercher dans le tas un bloc libre suffisamment grand. On peut imaginer une première version qui parcourt séquentiellement tous les blocs du tas jusqu'à en trouver un adapté. Ce parcourt nécessite deux opérations principales.

— Tester un bloc désigné par un pointeur p. Pour cela, il faut regarder l'entête (`p[-1]`), la décomposer en un bit de statut s et une taille t, et vérifier d'une part le statut libre (`!`s), et d'autre part que la taille est bien suffisante pour accueillir n mots dans l'espace utile (`n <= t - 4`).

— Passer au bloc suivant. En ajoutant au pointeur p désignant un bloc la taille t de ce bloc, on obtient l'adresse du bloc suivant : on a un chaînage implicite des blocs.

Ces opérations de base étant fixées, on peut varier les stratégies. Par exemple :

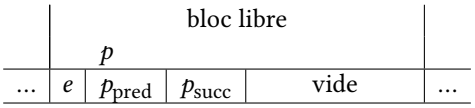
— commencer par le premier bloc et s'arrêter au premier bloc convenable trouvé (*first fit*),

— commencer au bloc touché le plus récemment et s'arrêter au premier bloc convenable trouvé (*next fit*), ou

— chercher le plus petit des blocs convenables (*best fit*).

Enfin, si on a parcouru l'ensemble des blocs sans en trouver de convenable on peut effectuer un nouvel appel à `sbrk` pour étendre le tas, du moins tant qu'il reste de l'espace disponible. On réserve alors un nouveau bloc taillé dans cette extension.

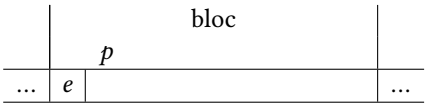
Cette première version de la recherche parcourt séquentiellement tous les blocs, libres ou réservés, jusqu'à avoir choisi le bloc à réserver. On pourrait faire nettement plus efficace en ne considérant que les blocs libres. Cela suppose en revanche d'avoir un chaînage entre les blocs libres, c'est-à-dire d'être capable, étant donné un bloc libre, de déterminer efficacement l'adresse d'un bloc libre « suivant ». Une astuce pour cela consiste à remarquer que tout bloc libre est... libre ! Autrement dit, l'espace utile d'un bloc libre n'est pas utilisé par le programme client, et l'allocateur peut l'utiliser pour stocker des méta-données supplémentaires. En l'occurrence, on prend les deux premiers champs de l'espace utile d'un bloc libre pour stocker les adresses p_{pred} et p_{succ} du bloc libre précédent et du bloc libre suivant.



À noter : pour bien avoir la place de stocker ces deux pointeurs, il faut que la taille de chaque bloc soit d'au moins 12 octets, et même 16 octets en tenant compte de la contrainte « multiple de 8 ».

On peut donc coder, à l'aide des ces zones temporairement inutilisées de la mémoire, une liste doublement chaînée des blocs libres (*free list*), qui sera aisément mise à jour à chaque changement de statut d'un bloc. Il faut simplement mémoriser quelque part, dans une variable globale, l'adresse d'un premier bloc libre. On améliore encore cette technique en n'utilisant pas une seule mais plusieurs listes doublement chaînées de blocs libres, en distinguant plusieurs catégories de tailles (*segregated free lists*).

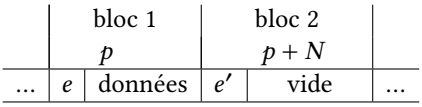
Une fois qu'on a trouvé un bloc convenable, désigné par un pointeur p vers son premier octet utile, on peut le réserver.



Il y a encore deux possibilités :

— on peut réserver intégralement le bloc, en modifiant simplement son bit de statut ($e = e$ `| 0x1`),

— si le bloc est trop grand, on peut aussi le découper en deux blocs, un qui sera réservé et l'autre laissé libre.



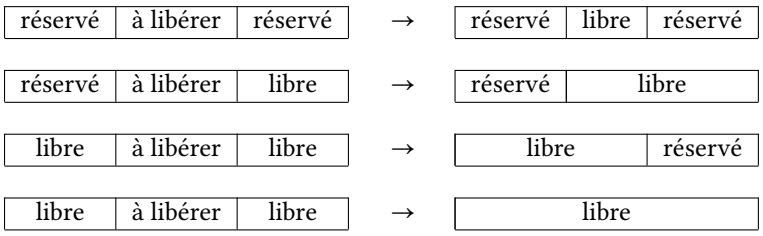
Dans ce schéma, on a réservé un premier bloc de taille N, pour une certaine valeur $N \geq n + 4$ multiple de 8, et laissé un deuxième bloc libre de taille $t - N$.

Le choix de découper ou non est à la discrétion de l'allocateur mémoire, et n'est pas si simple a priori : découper permet d'éviter de sous-employer un bloc, mais peut aussi impliquer à terme une fragmentation de la mémoire en de multiples petits blocs difficilement réutilisables. De même, le choix de la valeur N appartient à la stratégie de l'allocateur.

Fonction `free` Il y a une manière extrêmement simple de libérer un bloc désigné par un pointeur p : aller modifier dans son entête le bit de statut, pour le repasser à 0 ($e = e$ `& ~0x1`). Dans le cas d'une stratégie *free list* ou *segregated free lists* il faut ajouter à cela l'intégration du bloc libéré dans la liste de blocs libres. On peut le faire par exemple juste avant le premier bloc, en suivant un algorithme standard d'insertion dans une liste doublement chaînée.

Le défaut de cette manière simple est l'apparition progressive de *fragmentation* de la mémoire : si on a découpé un grand bloc pour éviter de perdre de la place (ce qui peut être une bonne chose), on aura après libération plusieurs petits blocs. Ces petits blocs ne pourront être utilisés que pour de petites allocations, et il sera à terme de plus en plus difficile de trouver un bloc de grand taille pour une une allocation d'un grand nombre d'octets.

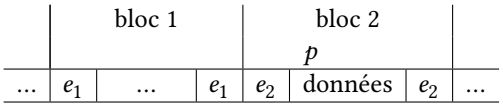
On peut améliorer la situation en fusionnant un groupe de blocs libres adjacents, c'est-à-dire consécutifs en mémoire. Cette opération de fusion peut même être intégrée à l'opération `free` elle-même : si on trouve un bloc libre juste avant ou juste après le bloc libéré par `free`, on les fusionne. Avec les différents statuts « libre » ou « réservé » des deux blocs entourants le bloc libéré par `free`, on a les quatre situations possibles suivantes.



À noter : si ces fusions sont faites systématiquement, on a la garantie de ne jamais avoir deux blocs libres consécutifs.

Pour réaliser ces fusions il faut pouvoir consulter les méta-données, c'est-à-dire les entêtes, des blocs situés juste avant et juste après le bloc libéré.

— Partant du pointeur p du bloc à libérer, on accède facilement au bloc suivant : il suffit d'ajouter la taille du bloc à libérer (et l'entête est le mot précédent l'adresse obtenue).
— En l'état, on n'a en revanche pas de manière directe de calculer l'adresse du bloc précédent. On peut régler ce problème en enrichissant encore un peu la structure de nos blocs, en répétant l'entête d'un bloc à la fin de celui-ci (*boundary tags*).



On accède alors aux méta-données du bloc précédent en regardant deux mots avant l'adresse p.

La technique des *boundary tags* permet de meilleures fusions. Elle a néanmoins un coût : il faut sacrifier un bloc supplémentaire pour cette deuxième copie de l'entête. On peut cependant ajouter une nouvelle optimisation : ne répéter l'entête que dans le cas d'un bloc libre, car c'est seulement dans cette situation que l'on a besoin de la taille, et ajouter dans l'entête de chaque bloc un bit donnant le statut du bloc précédent.

Bilan : un bon allocateur de mémoire est un programme assez subtil, qui repose sur des structures de données qui sont cachées dans les interstices du tas. Il y a beaucoup de choix de stratégie à faire, et une abondance d'études empiriques sur l'efficacité des différents stratégies. Dans Linux, la bibliothèque `malloc.c` *contient plus de 5000 lignes de code.*

6.11 Structures de données

Les langages de programmation permettent couramment la définition de structures de données composées de plusieurs champs, chaque champ étant identifié et accessible par un nom. On peut citer par exemple :

— les structures en C

```
struct point {
  int x;
  int y;
};
```

— les enregistrements en caml

```
type point = {
  x: int;
  y: int;
}
```

— les objets en java

```
class Point {
  public final int x;
  public final int y;
  public Point(int x, int y) { this.x = x; this.y = y; }
}
```

Ici, `point` désigne un nouveau type de données, `x` et `y` désignent des noms de champs de ce type de données, et les occurrences de `int` désignent le type du contenu associé aux champs `x` et `y`.

Manipulation de structures On peut représenter une structure en mémoire par un bloc de plusieurs mots mémoire consécutifs, où les valeurs des champs sont stockées l'une après l'autre.

x	y	z	b
1	2	42	true

Dans un type de structure donné, on place systématiquement les champs dans le même ordre. Ainsi, chaque nom de champ peut être associé à une position, et le compilateur traduit chaque accès à un champ identifié par son nom en un accès à la position correspondante dans le bloc mémoire.

$$\begin{aligned} x &\mapsto 0 \\ y &\mapsto 1 \\ z &\mapsto 2 \\ b &\mapsto 3 \end{aligned}$$

On crée une telle structure en allouant l'espace disponible pour le bloc mémoire complet, et en initialisant éventuellement les champs (le détail dépendant du langage). En particulier :

— En caml, on note

```
let p = { x=1; y=2; } in ...
```

la création d'une nouvelle structure `point` dont les champs `x` et `y` sont initialisés respectivement avec 1 et 2. Le bloc est créé dans le tas, ce qui implique une allocation de type `malloc`. La valeur de la variable `p` de type `point` est ensuite concrètement un pointeur vers le bloc alloué dans le tas. On accède au champ `x` de `p` avec la notation `p.x`.

— En java, la situation est similaire à celle de caml : on crée une structure (un objet) avec

```
Point p = new Point(1, 2);
```

Cette opération alloue un bloc sur le tas, et définit `p` comme un pointeur vers ce bloc. On accède au champ `x` de `p` de même avec `p.x`.

— En C, on alloue l'espace pour une structure sur le tas à l'aide de `malloc`.

```
point *p = malloc(sizeof(point));
```

L'adresse du bloc explicitement alloué par `malloc` est ensuite explicitement stockée dans une variable `p` de type `point*`, c'est-à-dire un pointeur vers une structure de type `point`. On accède au champ `x` de la structure pointée par `p` avec la notation `p->x`. On peut donc initialiser ensuite la structure avec les instructions supplémentaires

```
p->x = 1;
p->y = 2;
```

À noter, C admet également la notation utilisée en caml mais avec une signification légèrement différente : `point p = { x=1; y=2; };` place la structure sur la pile plutôt que sur le tas, et la variable `p` désigne directement cette structure et non un pointeur. Et dans ce cas, on accède au champ `x` avec la même notation `p.x`.

Un langage avec structures de données On se propose d'étendre le langage Mini-C du DM avec des structures de données définies par l'utilisateur. On introduit pour cela les nouveaux éléments de syntaxe concrète suivants :

- Une nouvelle déclaration
  `struct s { t1 x1; ...; tN xN; }`
  pour la définition du type `s` d'une structure dont les champs ont pour noms les `xi` et pour types les `ti`.
- Un nouveau type
  `s*`
  désignant un pointeur vers une structure de nom `s`.
- Deux nouvelles formes d'expressions
  `p->x`
  pour l'accès au champ `x` de la structure pointée par `p`, et
  `sizeof(s)`
  pour le calcul du nombre d'octets à allouer pour une structure de type `s`.
- Une nouvelle instruction
  `p->x = e`
  pour l'affectation d'une nouvelle valeur au champ `x` de la structure pointée par `p`.

La définition d'un programme ne se limite plus à une liste de variables globales et une liste de fonctions : on y ajoute une liste de types de structures.

On peut ensuite compiler ce langage Mini-C, vers le langage PIMP (l'extension de IMP avec des pointeurs décrite à la section 6.9). Ces deux langages sont très proches : ils possèdent la même base impérative, et diffèrent seulement par leur manière d'accéder aux données du tas :

- en Mini-C, on définit des structures de données allouées dans le tas, et on accède à leurs champs via les noms avec `e->x`,
- en PIMP, on manipule directement des pointeurs et des calculs d'adresses.

Pour compiler un accès `e->x` de Mini-C vers PIMP, il faut donc :

1. identifier le type `s` de la structure pointée par `e`,
2. consulter la définition de `s` pour obtenir la position du champ `x` dans la structure,
3. en déduire l'adresse de la donnée à laquelle on veut accéder.

On a ici un phénomène que nous n'avions pas observé dans les traductions de PIMP vers LLIR ou LLIR vers MIPS : on a besoin des types pour traduire un programme Mini-C vers PIMP ! Ici, les types ne servent donc pas seulement à vérifier la cohérence des programmes.

On peut donc suivre l'organisation suivante pour un compilateur de Mini-C :

1. L'analyse syntaxique produit un AST Mini-C dans lequel on trouve, en plus du code, la liste des définitions de types de structures et les types déclarés pour chaque variable ou fonction.
2. Le vérificateur de type prend en entrée un AST Mini-C. Son résultat n'est pas un booléen (bien typé/mal typé), mais un nouvel AST « Mini-C typé ». Ce nouvel AST typé a exactement la même structure que l'AST pris en entrée, et contient juste des informations supplémentaires : chaque expression (et chaque sous-expression) est annotée par le type qui a été calculé par le vérificateur.
3. Le traducteur de Mini-C vers PIMP à proprement parler prend en entrée l'AST Mini-C typé.

Autres formes de types définies par l'utilisateur On peut également intégrer à Mini-C d'autres formes courantes de types définis par l'utilisateur. Une *énumération* est un type défini par un ensemble fini de symboles. On trouve le concept aussi bien en C avec le mot-clé `enum`

```
enum tete {
  valet; dame; roi;
```

```
}
```

qu'en caml avec un type algébrique où tous les constructeurs sont constants.

```
type tete =
  Valet | Dame | Roi
```

Pour traduire un programme utilisant des énumérations vers un langage de plus bas niveau comme PIMP, on numérote les symboles de chaque énumération.

$$\begin{aligned} \texttt{valet} &\mapsto 0 \\ \texttt{dame} &\mapsto 1 \\ \texttt{roi} &\mapsto 2 \end{aligned}$$

Il suffit alors de traduire dans le programme chaque occurrence d'un tel symbole par son numéro.

Un peu plus riche, un type *union* décrit pour un type de donnée une alternative entre plusieurs formats. En C par exemple, on déclare avec

```
union s {
  point p;
  int   n;
};
```

un type `s` désignant une donnée qui peut être soit un point soit un entier. On trouve un concept similaire en caml avec la définition de type algébrique

```
type s =
  | P of point
  | N of int
```

Nuance entre ces deux versions : en caml on introduit impérativement des *constructeurs*, ici les symboles `P` et `N`, qui permettent de distinguer les différents formats. L'opération de filtrage permet alors, partant d'une donnée concrète de type `s` en caml, de tester sa forme. En C en revanche, la représentation d'une donnée de type `s` ne permet pas par défaut de distinguer si à l'intérieur se trouve un point ou un entier. Il faut ajouter explicitement ces informations dans la structure, par exemple à l'aide d'une énumération des différents formes possibles.

Compilateur Mini-C, bilan On peut obtenir un compilateur complet de Mini-C (langage impératif avec fonctions et structures de données) vers l'assembleur MIPS en combinant les éléments suivants du cours :

Élément	Référence	Langage utilisé
analyseur lexical	chapitre 3 et DM	ocamllex
analyseur grammatical	chapitre 4 et DM	menhir
vérificateur de types	chapitre 5 et DM	caml
traducteur Mini-C vers PIMP	section 6.11	caml
allocateur mémoire	section 6.10	*PIMP*
traducteur PIMP vers LLIR	sections 6.4 et 6.9	caml
traducteur LLIR vers MIPS	section 6.8	caml

On pourrait ensuite continuer dans plusieurs directions, et notamment enrichir le langage source et étendre le compilateur pour traiter de nouvelles fonctionnalités de plus haut niveau (objets ? exceptions ? programmation fonctionnelle ?), ou améliorer le compilateur lui-même pour qu'il produise du code assembleur optimisé dont l'exécution sera (beaucoup) plus efficace.

Et ensuite

Revenons sur ce que nous avons parcouru pendant ce semestre sur la compilation et la théorie des langages de programmation, et en particulier sur :

— ce qu'il faut en retenir, même pour ceux qui ne voudraient plus jamais toucher au domaine,
— ce qui reste, pour ceux qui voudraient aller plus loin.

Définition d'un langage de programmation Un langage de programmation est défini par *des* syntaxes et *des* sémantiques. Côté syntaxe :

— la syntaxe concrète définit ce que l'on écrit lorsque l'on programme, et attise parfois les passions et les trolls,
— la syntaxe abstraite définit les structures qui sont représentées par un texte concret, et est également ce sur quoi on peut raisonner [12].

Côté sémantique :

— la sémantique statique, en particulier avec les types, distingue les programmes qui ont du sens de ceux qui n'en ont pas,
— la sémantique dynamique décrit les comportements à attendre d'un programme, et spécifie notamment les résultats et les effets.

Exécution d'un programme Pour permettre l'exécution d'un programme donné dans un langage source, l'interprétation et la compilation sont deux voies de natures différentes. On peut observer cette différence par les entrées et sorties produites dans chaque cas.

— Interpréter un programme, c'est exécuter son comportement. Dans ce contexte on a :
  — en entrée : un programme *et* des entrées,
  — en sortie : le résultat.
— Compiler un programme, c'est le traduire vers un autre langage (qui pourra ensuite être exécuté à l'aide de ses mécanismes propres). Dans ce contexte on a :
  — en entrée : un programme,
  — en sortie : un programme *équivalent*.

Au-delà de leurs différences fondamentales, ces deux voies ont de nombreux aspects techniques communs. Elles mettent en œuvre l'analyse d'un texte source (analyse lexicale, analyse grammaticale), la manipulation d'une représentation symbolique du programme (AST), et imposent le respect d'une sémantique (correction).

Un domaine tentaculaire Étudier la compilation, c'est approfondir sa culture et sa compréhension des langages de programmation. On y décortique les langages et leurs mécanismes. On en ressort meilleur programmeur, mieux apte à écrire des programmes sûrs et efficaces, et on améliore sa capacité à comprendre et corriger les erreurs dans ses programmes.

C'est aussi une occasion d'explorer et de connecter des champs multiples de l'informatique :

— l'algorithmique, avec des algorithmes avancés d'analyse et d'optimisation,
— l'informatique théorique, avec la sémantique, les langages formels, la calculabilité,
— la technologie informatique, lors de l'interface avec l'architecture et le système,
— la programmation elle-même : un compilateur est un programme d'envergure, dont l'écriture regroupe tous les aspects précédents.

Pour aller plus loin Certains des chapitres de ce cours ouvrent sur des domaines que vous pourrez explorer dans la suite de votre cursus.

À la suite des chapitres 3 et 4 s'étendent les domaines de la *calculabilité* et de la *complexité*, qui donnent les *fondements théoriques de l'algorithmique*. On y cherche notamment à caractériser les dispositifs de calcul (automates, machines de Turing, machines RAM, etc) et les problèmes qu'ils peuvent résoudre. On y découvre avec la notion d'indécidabilité que certains problèmes ne peuvent pas être résolus par des algorithmes. On y caractérise aussi certains problèmes algorithmiques qui, bien que solubles, sont intrinséquement difficiles. On ne sait en revanche toujours pas si P est égal ou non à NP.

À la suite des chapitres 2 et 5, on arrive à la *science des langages de programmation*. Outre la modélisation du comportement des programmes, on y vise à créer de nouveaux outils

12. De là à dire que les trolls résonnent plus qu'ils ne raisonnent, il n'y a qu'un pas.

d'analyse des programmes et de leurs comportements, de nouveaux systèmes de types et de nouveaux langages de programmation permettant d'améliorer la sûreté des programmes futurs. C'est ici que l'on traite de preuve de programmes. C'est de là que viennent les assistants à la preuve, et les plus intéressants des nouveaux langages de programmation. À ce propos, connaissez-vous Rust ?

Le chapitre 6 est le début de l'*implémentation des langages de programmation*. On y veut faire le lien entre des mécanismes de programmation de haut niveau, agréables au programmeur, et les opérations plus élémentaires accessibles à la machine. On y cherche également à ce que l'exécution des programmes soit ensuite la plus efficace possible. Cela peut passer par des analyses fines de la structure des programmes et de leurs opérations pour détecter des endroits où des optimisations sont possibles, ou encore par la meilleure utilisation des capacités matérielles de la machine cible.

Table des matières