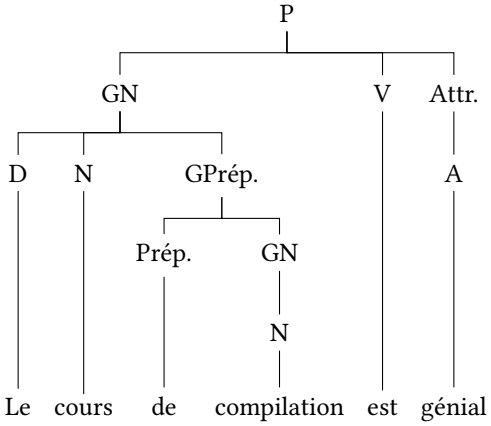
Langages, interprétation, compilation

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

Table des matières

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.

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

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

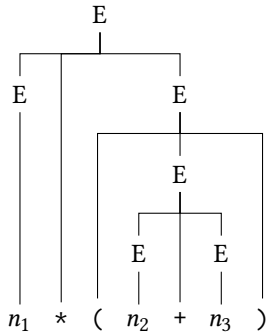
La ***grammaire*** d'un langage décrit les structures de phrases légales dans ce langage. On la définit à l'aide de deux éléments en plus du lexique :

- 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_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, n), $(E, E\mathsf{+}E)$, $(E, E\mathsf{*}E)$ et $(E, \mathsf{(}E\mathsf{)})$, encore écrites

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

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*(+n_3) 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 :

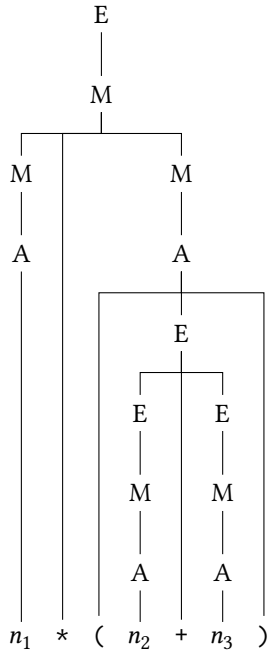
- le regroupement d'un fragment E, du mot `+` et d'un deuxième fragment E forme un fragment E,
- un fragment M seul forme également un fragment E,
- le regroupement d'un fragment M, du mot `*` et d'un deuxième fragment M forme un fragment M,
- un fragment A seul forme également un fragment M,
- un mot 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

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

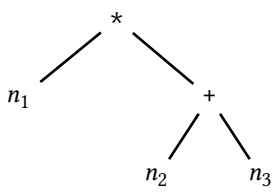
Nous avons alors pour $n_1\texttt{*(}n_2\texttt{+}n_3\texttt{)}$ un nouvel arbre de dérivation à partir du symbole de départ E.



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))
```

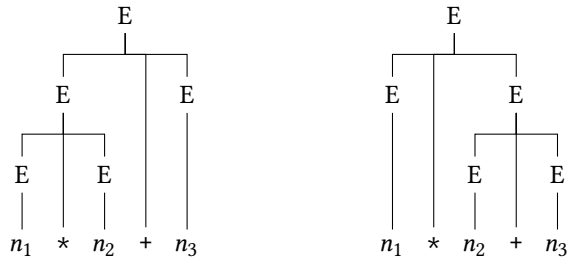
et serait dessiné ainsi, indépendamment de la grammaire utilisée pour caractériser la structure des expressions bien formées.



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.

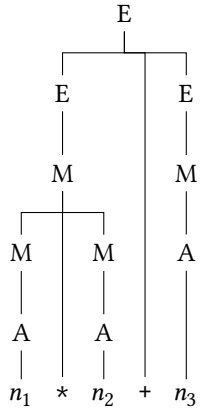
Ambiguïtés

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.

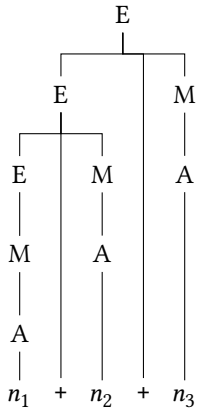
La deuxième grammaire ne présente pas d'ambiguïté sur cette expression, qui ne peut être dérivée que par l'arbre suivant.



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.

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

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

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

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

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

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 :

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

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

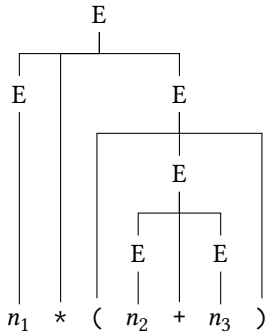
Les deux notions de dérivabilité sont équivalentes. Que l'on considère des arbres de dérivation ou des suites de transformations, les mêmes phrases sont dérivables à partir des mêmes symboles.

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.

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



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 $(X, a_1 \dots a_k)$ 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.

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

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

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

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.

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

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 Premiers(α) l'ensemble des symboles terminaux qui peuvent se trouver en tête d'une séquence dérivée à partir de α.

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

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.

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

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 Premiers(α), pour α une séquence de $(T \cup N)^*$. On a les équations suivantes :

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

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 $(\emptyset, \ldots, \emptyset)$, 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 :

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

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 $a \in \mathsf{Premiers}(\beta)$ 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 $a \in \mathsf{Suivants}(X)$ 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.

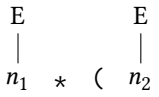
— Lecture de l'entrée un mot à la fois, de gauche à droite.

— 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.

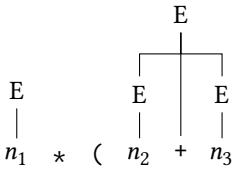
Sur la phrase n_1 * (n_2 + n_3) et avec la grammaire naïve des expressions arithmétiques, voici les regroupements qui peuvent être faits en fonction des différents niveaux d'avancement de la lecture.

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

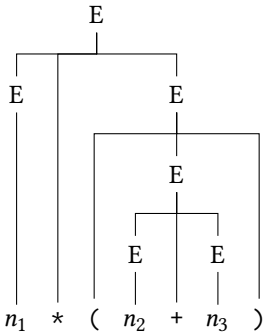
— Après lecture jusqu'à n_2 on a identifié une séquence comportant une expression, les symboles * et (et une deuxième expression.



— 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.

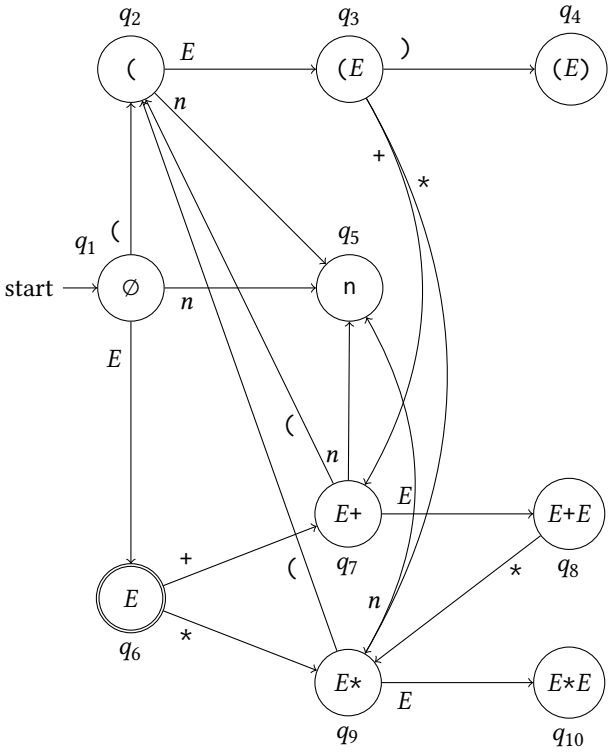
Sommet de pile	**Prochain mot**	**Action**
$\emptyset$	n ou `(`	progression
	autres	échec
n	quelconque	réduction $E \prec n$
`(`	n ou `(`	progression
	autres	échec
`(` E	`)` ou `+` ou `*`	progression
	autres	échec
`(` E `)`	quelconque	réduction $E \prec$ `(`E`)`
E	`+` ou `*`	progression
	fin de l'entrée	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 _ | ParO 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.

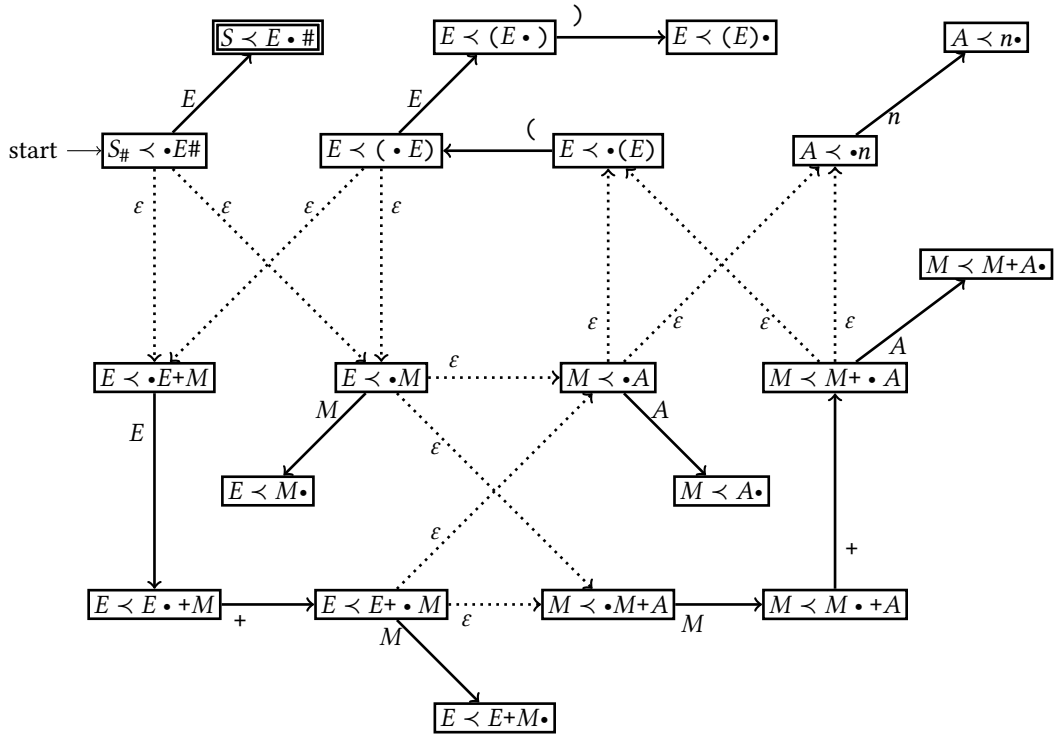
En prenant la grammaire suivante des expressions arithmétiques

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

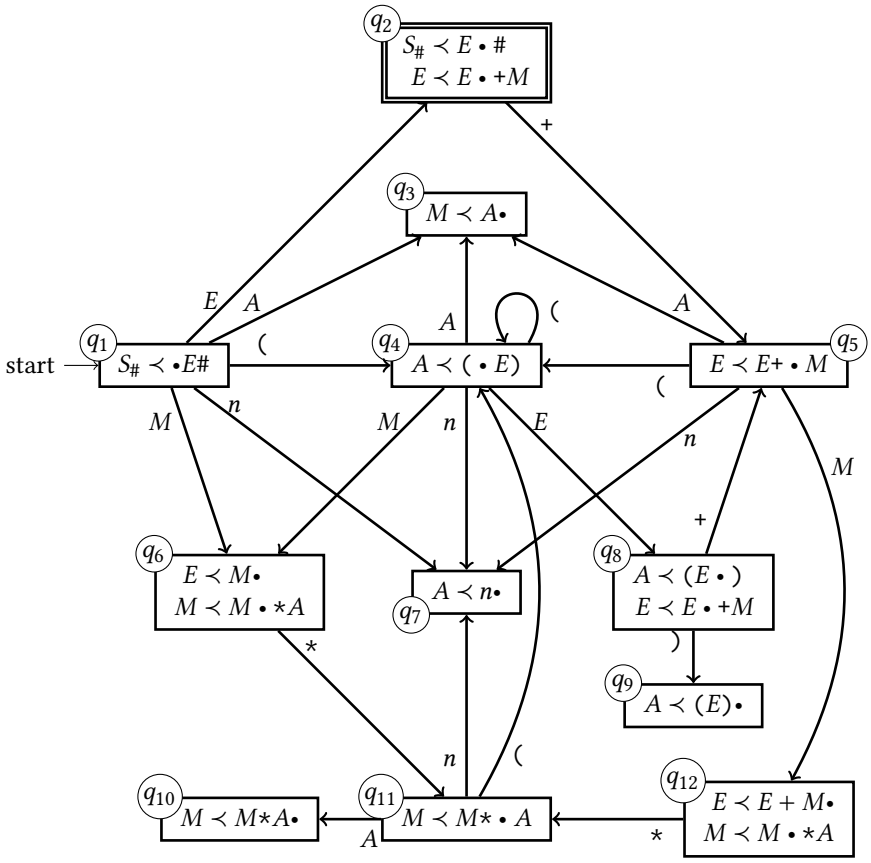
et en y ajoutant la règle de départ

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

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

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

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 $a \in \mathsf{Suivants}(X)$. Un tel état s'interprète comme « nous sommes en train de chercher à reconnaître la séquence $\alpha\beta$ pour la regrouper en un fragment X qui devra être suivi du symbole a, nous avons déjà reconnu la partie α et il reste à reconnaître la partie β (et vérifier la présence de a ensuite) ».
— L'unique état initial est $[S_\# \prec \bullet S, \#]$.
— L'unique état acceptant est $[S_\# \prec S \bullet, \#]$.
— Les transitions sont toutes celles qui peuvent être formées de l'une des trois manières suivants :
  — $[Y \prec \alpha \bullet a\beta, b] \xrightarrow{a} [Y \prec \alpha a \bullet \beta, b]$ avec a symbole terminal,
  — $[Y \prec \alpha \bullet X\beta, b] \xrightarrow{a} [Y \prec \alpha X \bullet \beta, b]$ 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 $c \in \mathsf{Premiers}(\beta b)$.

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

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

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

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).

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

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

— Une grammaire naïve et naturelle.

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

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.

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

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:
```

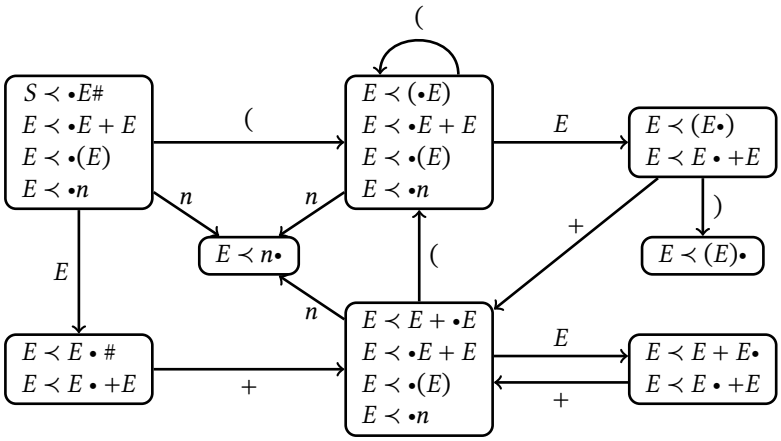
On y voit notamment un numéro pour l'état, un ensemble d'éléments nécessairement présents au sommet de la pile et une règle de réduction $E \prec E{+}E$ en cours de reconnaissance. Le point entre le symbole `PLUS` et la deuxième occurrence de `expr` correspond au niveau de progression dans la règle (c'est 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.

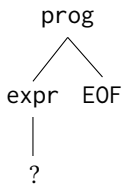
Détaillons le contenu de ce fichier pour notre exemple. On a d'abord une identification de l'état, du prochain lexème de l'entrée pour lequel le conflit existe, et d'un état de la pile correspondant à cet état (ce dernier point peut être vu comme un chemin dans l'automate).

```
** 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

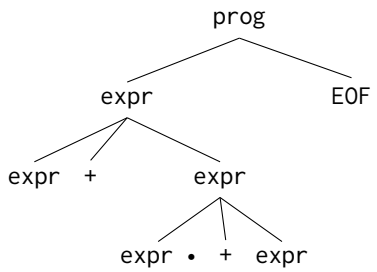


Les deux actions possibles (progression ou réduction), correspondent alors à deux formes différentes du sous-arbre noté par un point d'interrogation. Le rapport détaille d'abord le cas de la progression.

```
** 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'en-

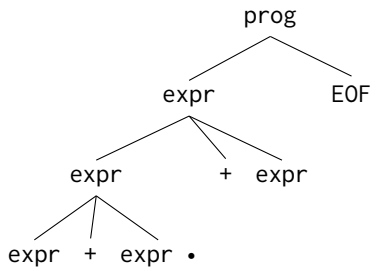
trée.



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

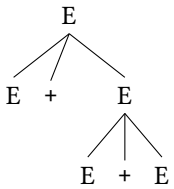
```
** In state 5, looking ahead at PLUS, reducing production
** expr -> expr PLUS expr
** is permitted because of the following sub-derivation:

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.

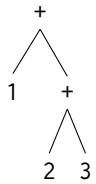


Bilan de ce rapport : à un tel stade de l'analyse de notre entrée, nous avons un choix entre progresser avec le symbole +, ou réduire avec la règle $E \prec E{+}E$.

— Progresser mène à un arbre de dérivation

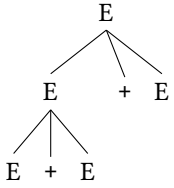


Sur l'entrée concrète `1+2+3` nous aurions l'arbre de syntaxe

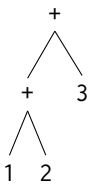


correspondant au parenthésage `1+(2+3)`.

— 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.

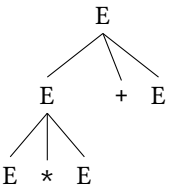
```
%left PLUS
```

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.

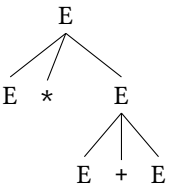
On obtient trois nouveaux conflits :
— réduire `expr STAR expr` ou progresser avec `STAR`,
— réduire `expr STAR expr` ou progresser avec `PLUS`,
— réduire `expr PLUS expr` ou progresser avec `STAR`.

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

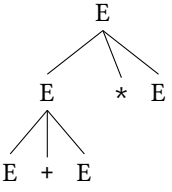


c'est-à-dire interpréter `2*3+4` comme `(2*3)+4`,
— soit progresser et s'orienter vers un arbre de dérivation de la forme



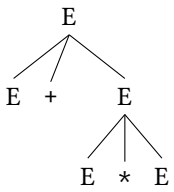
c'est-à-dire interpréter `2*3+4` comme `2*(3+4)`.
La solution cohérente avec les conventions mathématiques usuelles est la première : il faut donc dans ce cas favoriser la réduction.

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

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

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 $\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2$.

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. $\Gamma \vdash x \texttt{ + } f \texttt{ 1 : int}$ 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.

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

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 :

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

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

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 $\Gamma \vdash \texttt{5 : int}$).

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 $\Gamma \vdash$ `fun x -> x x` $: \tau$ aurait nécessairement la forme

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

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 $\text{fv}(e_1) \subseteq \text{dom}(\Gamma)$ et $\text{fv}(e_2) \subseteq \text{dom}(\Gamma)$. Or $\text{fv}(e_1 + e_2) = \text{fv}(e_1) \cup \text{fv}(e_2)$. Des deux hypothèses de récurrence on déduit $\text{fv}(e_1) \cup \text{fv}(e_2) \subseteq \text{dom}(\Gamma)$, et donc $\text{fv}(e_1 + e_2) \subseteq \text{dom}(\Gamma)$.
— Cas $\Gamma \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \tau_2$ 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 $\text{fv}(e_1) \subseteq \text{dom}(\Gamma)$ et $\text{fv}(e_2) \subseteq \text{dom}(\Gamma) \cup \{x\}$ (remarquez en effet que la prémisse relative à e_2 est dans un environnement étendu avec la variable x). Or $\text{fv}(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2) = \text{fv}(e_1) \cup (\text{fv}(e_2) \setminus \{x\})$. Par la première hypothèse de récurrence nous avons $\text{fv}(e_1) \subseteq \text{dom}(\Gamma)$. Par la deuxième hypothèse de récurrence nous avons $\text{fv}(e_2) \subseteq \text{dom}(\Gamma) \cup \{x\}$, dont nous déduisons $\text{fv}(e_2) \setminus \{x\} \subseteq \text{dom}(\Gamma)$. Ainsi on a bien $\text{fv}(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2) \subseteq \text{dom}(\Gamma)$.
— 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.

$$\begin{array}{rcl} \mathrm{fv}(\texttt{int}) & = & \emptyset \\ \mathrm{fv}(\alpha) & = & \{\alpha\} \\ \mathrm{fv}(\tau_1 \to \tau_2) & = & \mathrm{fv}(\tau_1) \cup \mathrm{fv}(\tau_2) \\ \mathrm{fv}(\forall\alpha.\tau) & = & \mathrm{fv}(\tau) \setminus \{\alpha\} \end{array}$$

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 $\forall\alpha.\tau$.

$$\frac{\Gamma \vdash e : \tau \qquad \alpha \notin \text{fv}(\Gamma)}{\Gamma \vdash e : \forall\alpha.\tau}$$

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.

$$\text{fv}(x_1 : \tau_1, \dots, x_n : \tau_n) \quad = \quad \bigcup_{1 \leq i \leq n} \text{fv}(\tau_i)$$

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*.

$$\frac{\dfrac{\overline{x : \alpha \vdash x : \alpha}}{\vdash \texttt{fun x -> x} : \alpha \to \alpha} \quad \alpha \notin \text{fv}(\emptyset)}{\vdash \texttt{fun x -> x} : \forall\alpha.\alpha \to \alpha}$$

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.

$$\frac{\dfrac{\overline{\Gamma \vdash f : \forall\alpha.\alpha \to \alpha}}{\Gamma \vdash \texttt{f} : (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})} \qquad \dfrac{\overline{\Gamma \vdash f : \forall\alpha.\alpha \to \alpha}}{\Gamma \vdash \texttt{f} : \texttt{int} \to \texttt{int}}}{\Gamma \vdash \texttt{f f} : \texttt{int} \to \texttt{int}}$$

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.

$$\frac{\dfrac{\dfrac{\overline{\Gamma \vdash f : \forall\alpha.\alpha \to \alpha}}{\Gamma \vdash \texttt{f} : (\beta \to \beta) \to (\beta \to \beta)} \qquad \dfrac{\overline{\Gamma \vdash f : \forall\alpha.\alpha \to \alpha}}{\Gamma \vdash \texttt{f} : \beta \to \beta}}{\Gamma \vdash \texttt{f f} : \beta \to \beta} \quad \beta \notin \text{fv}(\Gamma)}{\Gamma \vdash \texttt{f f} : \forall\beta.\beta \to \beta}$$

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).

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash g : \beta \to \gamma} \quad \dfrac{\dfrac{}{\Gamma \vdash f : \alpha \to \beta} \quad \dfrac{}{\Gamma \vdash x : \alpha}}{\Gamma \vdash \texttt{f x} : \beta}}{\Gamma \vdash \texttt{g (f x)} : \gamma}}{\vdash \texttt{fun f -> fun g -> fun x -> g (f x)} : (\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma)}\ \alpha, \beta, \gamma \notin \mathsf{fv}(\emptyset)}{\vdash \texttt{fun f -> fun g -> fun x -> g (f x)} : \forall\alpha\beta\gamma.(\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma)}$$

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 $\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2$ 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 $\Gamma_2 = \{ f : \forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha) \}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash x : \alpha \to \alpha} \quad \dfrac{\dfrac{}{\Gamma_1 \vdash x : \alpha \to \alpha} \quad \dfrac{}{\Gamma_1 \vdash y : \alpha}}{\Gamma_1 \vdash x\ y : \alpha}}{x : \alpha \to \alpha, y : \alpha \vdash x\ (x\ y) : \alpha}}{x : \alpha \to \alpha \vdash \texttt{fun}\ y \texttt{->}\ x\ (x\ y) : \alpha \to \alpha}}{\vdash \texttt{fun}\ x \texttt{->}\ \texttt{fun}\ y \texttt{->}\ x\ (x\ y) : (\alpha \to \alpha) \to (\alpha \to \alpha)} \quad \dfrac{\dfrac{}{\Gamma_2 \vdash f : (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})} \quad \dfrac{\dfrac{\dfrac{}{\Gamma_2, z : \texttt{int} \vdash z : \texttt{int}} \quad \dfrac{}{\Gamma_2, z : \texttt{int} \vdash 1 : \texttt{int}}}{\Gamma_2, z : \texttt{int} \vdash z\texttt{+1} : \texttt{int}}}{\Gamma_2 \vdash \texttt{fun}\ z \texttt{->}\ z\texttt{+1} : \texttt{int} \to \texttt{int}}}{f : \forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha) \vdash f\ (\texttt{fun}\ z \texttt{->}\ z\texttt{+1}) : \texttt{int} \to \texttt{int}}$$

$$\vdash \texttt{let}\ f = \texttt{fun}\ x \texttt{->}\ \texttt{fun}\ y \texttt{->}\ x\ (x\ y)\ \texttt{in}\ f\ (\texttt{fun}\ z \texttt{->}\ z\texttt{+1}) : \texttt{int} \to \texttt{int}$$

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, `fun x -> fun y -> 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 η = `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 ζ = `int` et θ, c'est-à-dire `int` $\to$ `int`.

Finalement, l'expression f `(fun z -> z+1)` a le type θ = `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

$$e \implies v$$

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.

$$\begin{array}{rcl} v & ::= & n \\ & | & \texttt{fun}\ x\ \texttt{->}\ e \end{array}$$

Les règles d'inférence vont ensuite traduire les équations définissant notre précédente fonction eval.

- $\text{eval}(n) = n$. Une constante entière est sa propre valeur. La règle associée est un axiome.

$$\frac{}{n \implies n}$$

- $\text{eval}(e_1\ \texttt{+}\ e_2) = \text{eval}(e_1) + \text{eval}(e_2)$. 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.
- $\text{eval}(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2) = \text{eval}(e_2[x := \text{eval}(e_1)])$. 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.

$$\frac{e_1 \implies v_1 \qquad e_2[x := v_1] \implies v}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \implies v}$$

- $\text{eval}(\texttt{fun}\ x\ \texttt{->}\ e) = \texttt{fun}\ x\ \texttt{->}\ e$. Une fonction est sa propre valeur. Comme pour les constantes, la règle associée est un axiome.

$$\frac{}{\texttt{fun}\ x\ \texttt{->}\ e \implies \texttt{fun}\ x\ \texttt{->}\ e}$$

- $\text{eval}(e_1\ e_2) = \text{eval}(e[x = \text{eval}(e_2)])$ si $\text{eval}(e_1) = \texttt{fun}\ x\ \texttt{->}\ e$. 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.

$$\dfrac{\dfrac{}{\texttt{fun}\,x\texttt{->}x\texttt{+}x \implies \texttt{fun}\,x\texttt{->}x\texttt{+}x} \qquad \dfrac{\dfrac{}{20 \implies 20} \qquad \dfrac{}{1 \implies 1}}{20\texttt{+}1 \implies 21} \qquad \dfrac{\dfrac{}{21 \implies 21} \qquad \dfrac{}{21 \implies 21}}{21\texttt{+}21 \implies 42}}{(\texttt{fun}\ x\ \texttt{->}\ x\ \texttt{+}\ x)\,(20\ \texttt{+}\ 1) \implies 42}$$

$$\texttt{let}\ f\ \texttt{=}\ \texttt{fun}\ x\ \texttt{->}\ x\ \texttt{+}\ x\ \texttt{in}\ f\,(20\ \texttt{+}\ 1) \implies 42$$

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 $e \implies v$ (la forme de la dérivation peut changer, mais le seul fait qui compte est qu'une dérivation *existe*). En l'occurrence, on peut dériver

$$\texttt{let}\ f\ \texttt{=}\ \texttt{fun}\ x\ \texttt{->}\ x\ \texttt{+}\ x\ \texttt{in}\ f\,(20\ \texttt{+}\ 1) \implies 42$$

en appel par nom de la manière suivante.

$$\dfrac{\dfrac{}{\texttt{fun}\,x\texttt{->}x\texttt{+}x \implies \texttt{fun}\,x\texttt{->}x\texttt{+}x} \qquad \dfrac{\dfrac{\dfrac{}{20 \implies 20} \quad \dfrac{}{1 \implies 1}}{20\texttt{+}1 \implies 21} \qquad \dfrac{\dfrac{}{20 \implies 20} \quad \dfrac{}{1 \implies 1}}{20\texttt{+}1 \implies 21}}{(20\texttt{+}1)\texttt{+}(20\texttt{+}1) \implies 42}}{(\texttt{fun}\ x\ \texttt{->}\ x\ \texttt{+}\ x)\,(20\ \texttt{+}\ 1) \implies 42}}{\texttt{let}\ f\ \texttt{=}\ \texttt{fun}\ x\ \texttt{->}\ x\ \texttt{+}\ x\ \texttt{in}\ f\,(20\ \texttt{+}\ 1) \implies 42}$$

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 $e \implies v$ 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 $e \implies v$. 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,

$$\frac{}{n \implies n} \qquad \frac{e_1 \implies n_1 \qquad e_2 \implies n_2}{e_1\ \texttt{+}\ e_2 \implies n_1 + n_2} \qquad \frac{e_2[x := e_1] \implies v}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \implies v}$$

$$\frac{}{\texttt{fun}\ x\ \texttt{->}\ e \implies \texttt{fun}\ x\ \texttt{->}\ e} \qquad \frac{e_1 \implies \texttt{fun}\ x\ \texttt{->}\ e \qquad e[x := e_2] \implies v}{e_1\ e_2 \implies v}$$

et démontrons que si $e \implies v$ alors v est une valeur et $\mathrm{fv}(v) \subseteq \mathrm{fv}(e)$, par récurrence sur la dérivation de $e \implies v$.

- Cas $n \implies n$: immédiat, car n est bien une valeur, et $\mathrm{fv}(n) \subseteq \mathrm{fv}(n)$.
- Cas $\texttt{fun}\ x\ \texttt{->}\ e \implies \texttt{fun}\ x\ \texttt{->}\ e$ immédiat de même.
- Cas $e_1\ \texttt{+}\ e_2 \implies n_1 + n_2$ avec $e_1 \implies n_1$ et $e_2 \implies n_2$. 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 $\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \implies v$ avec $e_2[x := e_1] \implies v$. 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 $e_1\ e_2 \implies v$ avec $e_1 \implies \texttt{fun } x \texttt{ -> } e$ et $e[x := e_2] \implies v$. 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.

$$\text{Si } \Gamma \vdash e : \tau \text{ et } e \implies v \text{ alors } \Gamma \vdash v : \tau.$$

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 $e \implies v$ 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 $n_1\ \texttt{+}\ n_2$, 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 $e_1\ \texttt{+}\ e_2$, 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)`.

$$\frac{\dfrac{\dfrac{1 + 2 = 3}{1\ \texttt{+}\ 2 \to 3}}{(1\ \texttt{+}\ 2)\ \texttt{+}\ 3 \to 3\ \texttt{+}\ 3}}{((1\ \texttt{+}\ 2)\ \texttt{+}\ 3)\ \texttt{+}\ (4\ \texttt{+}\ 5) \to (3\ \texttt{+}\ 3)\ \texttt{+}\ (4\ \texttt{+}\ 5)}$$

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.

$$\texttt{((1+2)+3)+(4+5)} \to \texttt{(3+3)+(4+5)} \to \texttt{(3+3)+9} \to \texttt{6+9} \to \texttt{15}$$

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 `fun` x `->` 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 $e' \Longrightarrow v$ alors $e \Longrightarrow v$.

Lemme : si $e \to e'$ et $e' \Longrightarrow v$ alors $e \Longrightarrow v$. 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 $n \Longrightarrow v$, qui n'est possible qu'avec $v = n$. On conclut donc avec la dérivation

$$\frac{\dfrac{}{n_1 \Longrightarrow n_1} \qquad \dfrac{}{n_2 \Longrightarrow n_2}}{n_1 + n_2 \Longrightarrow n}$$

— Cas `let` x `=` w `in` $e \to e[x := w]$, avec w une valeur et où l'hypothèse $e' \Longrightarrow v$ s'écrit $e' = e[x := w] \Longrightarrow v$. On conclut donc avec la dérivation

$$\frac{w \Longrightarrow w \qquad e[x := w] \Longrightarrow v}{\texttt{let}\ x\ \texttt{=}\ w\ \texttt{in}\ e \Longrightarrow v}$$

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 $e' \Longrightarrow v$ s'écrit $e_1' + e_2 \Longrightarrow v$. La prémisse $e_1 + e_1'$ nous donne comme hypothèse de récurrence « pour toute valeur v_1 telle que $e_1' \Longrightarrow v_1$ on a $e_1 \Longrightarrow v_1$ ».
Par inversion du jugement $e_1' + e_2 \Longrightarrow v$ on déduit que v s'exprime $n_1 + n_2$ avec n_1 et n_2 tels que $e_1' \Longrightarrow n_1$ et $e_2 \Longrightarrow n_2$. Avec l'hypothèse de récurrence on déduit donc $e_1 \Longrightarrow n_1$. Grâce à ce jugement, on peut construire la dérivation suivante.

$$\frac{e_1 \Longrightarrow n_1 \qquad e_2 \Longrightarrow n_2}{e_1 + e_2 \Longrightarrow n}$$

— 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

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

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

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

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 $n_1 + n_2 \to n$ avec $n = n_1 + n_2$. De l'hypothèse $\Gamma \vdash n_1 + n_2 : \texttt{int}$ on a nécessairement $\tau = \texttt{int}$ (lemme d'***inversion*** détaillé plus bas), et en outre $\Gamma \vdash n : \texttt{int}$.
- Cas $e_1 + e_2 \to e_1' + e_2$ 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 $\Gamma \vdash e_1 + e_2 : \tau$ 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

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

- 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 $\Gamma \vdash e_1 + e_2 : \tau$ alors $\tau = \texttt{int}$, $\Gamma \vdash e_1 : \texttt{int}$ et $\Gamma \vdash e_2 : \texttt{int}$.
- Si $\Gamma \vdash e_1\, e_2 : \tau$ alors il existe τ' tel que $\Gamma \vdash e_1 : \tau' \to \tau$ et $\Gamma \vdash e_2 : \tau'$.
- Si $\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau$ alors il existe τ_1 et τ_2 tels que $\tau = \tau_1 \to \tau_2$ et $\Gamma, x : \tau_1 \vdash e : \tau_2$.

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.

$$\text{Si } \Gamma, x : \tau' \vdash e : \tau \text{ et } \Gamma \vdash e' : \tau' \text{ alors } \Gamma \vdash e[x := e'] : \tau.$$

Preuve par récurrence sur la dérivation de typage $\Gamma, x : \tau' \vdash e : \tau$.

Théorème de sûreté du typage

Des lemmes de progression et de préservation du typage, on déduit l'énoncé suivant.

$$\textit{Si } \Gamma \vdash e : \tau \textit{ et } e \to^* e' \textit{ avec } e' \textit{ irréductible, alors } e' \textit{ est une valeur.}$$

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).