Introduction à la compilation (Langages et Grammaires)

Polytech'Paris-Saclay– 4ème année –

Département Informatique

Frédéric VOISIN

GRAMMAIRES ET LANGAGES

Les grammaires sont un formalisme pour décrire les règles d'un langage, informatique ou non, à partir desquelles on pourra déduire si un « mot » appartient au langage ou non, en précisant dans le premier cas la combinaison de règles qui permet de l'accepter. La notion a pour origine des travaux de linguistes, notamment N. Chomsky à la fin des années 1950, qui recherchaient un moyen systématique de décrire la grammaire pour des langues " naturelles " comme l'anglais ou le français. Ce fût un échec partiel, les langues naturelles n'étant pas assez régulières et il y est difficile de séparer nettement entre les aspects lexicaux, syntaxiques et sémantiques, mais les outils théoriques développés se sont révélés très bien adaptés aux langages informatiques, beaucoup moins complexes structurellement.

Dans la suite on se donne un alphabet fini V_t de « lettres », aussi appelées « symboles terminaux », qui peuvent apparaître dans les « mots » du langage (pour nous, une « lettre » correspondra à une unité lexicale et un « mot » au programme entier). Un second ensemble fini V_n de symboles permettra de désigner des " catégories syntaxiques ". Ses symboles sont appelés " non-terminaux " puisqu'ils n'apparaissent pas dans les mots du langage mais servent à en décrire la structure. Une règle de grammaire définit les sous-mots produits par un non-terminal. Ces catégories syntaxiques peuvent se faire mutuellement référence. On distingue un non-terminal particulier, appelé " axiome ", à partir duquel débutera la description de tout mot du langage de la grammaire :

Définition : une **grammaire** est un quadruplet $G = (V_t , V_n , P, S)$, où $V_t \cap V_n = \varnothing$

- V_t est un ensemble fini de symboles dits terminaux,
- V_n est un ensemble fini de symboles dits non-terminaux,
- P est un ensemble fini de règles, notées $\alpha ::= \beta$ avec $\alpha \in A^+$ et $\beta \in A^*$, où $A = V_t \cup V_n$ est l'ensemble des symboles de la grammaire. On impose que α contienne au moins un élément de V_n,
- $S \in V_n$ est appelé « axiome » de la grammaire.

Dans une règle (aussi appelée « production ») on distingue la partie gauche α et la partie droite β**.** Si la partie β est vide, on la note ε, symbole qui représente le mot vide. La notation $\alpha ::= \beta_1 \mid ... \mid \beta_n$ est utile quand plusieurs règles ont la même partie gauche.

Dans les exemples qui suivent les parties gauches de règles sont réduites à un non-terminal. Dans la suite on adopte la convention que les non-terminaux débutent par une majuscule, les terminaux étant les autres symboles. L'axiome est le non-terminal qui est partie gauche de la première production. Ces conventions permettent de définir une grammaire en ne listant explicitement que l'ensemble des productions. Nous notons par des lettres de la fin de l'alphabet (*u*, *v*, *w*, ...) des mots composés uniquement de terminaux, et par des lettres grecques des combinaisons arbitraires de terminaux et de non-terminaux (« formes sentencielles »).

- `G1 = { S ::= 0, S ::= 1 }`
- `G2 = { S ::= a S, S ::= T, T ::= b T, T ::= ε }`
- `G2' = { S: := a S, S ::= T, T ::= T b, T ::= ε }`
- `G3 = { S ::= ( S ) S, S::= [ S ] S, S ::= ε }`
- `G4 = { S ::= a S b, S ::= ε }`

Définition : Soit G une grammaire, deux formes sentencielles γ et δ sont reliées par un **pas de dérivation simple**, ce qu'on note $\gamma \rightarrow \delta$, s'il existe deux chaînes τ_1 et τ_2 et une règle α `::=` β de G qui permet de décomposer γ et δ sous la forme $\gamma = \tau_1 \alpha \tau_2$ et $\delta = \tau_1 \beta \tau_2$.

On appelle **dérivation**, notée $\rightarrow^*$, la fermeture réflexive et transitive de la relation $\rightarrow$

Un **pas de dérivation simple** entre γ et δ identifie dans γ une occurrence de partie gauche d'une règle et la remplace dans δ par la partie droite correspondante, sans modifier les contextes gauche et droite. Une **dérivation** est une séquence, éventuellement vide, de pas de dérivation simples.

On définit le langage engendré par une grammaire comme l'ensemble des mots (i .e. composés uniquement de **terminaux**) dérivables à partir de l'**axiome** :

Définition : Soit $G = (V_t , V_n , P, S)$ une grammaire, on définit le **langage engendré** par G, noté L(G), par : $L(G) = \{ w \in V_t^* / S \rightarrow^* w \}$.

Dans les exemples précédents, on a L(`G1`) = {`0, 1`}, L(`G2`) = L(`G2'`) = $\{ a^n b^m / n, m \geq 0 \}$, L(`G4`) = $\{ a^n b^n / n \geq 0 \}$. La grammaire `G3` engendre les expressions « bien parenthésées » à deux sortes de parenthèses : `(` et `)` d'une part, `[` et `]` d'autre part. Notez que les langages engendrés par `G3` et `G4` ne sont pas reconnaissables par automates à états finis : ceux-ci ne permettent pas de « compter » qu'il y a autant de `a` que de `b`, sauf si on borne arbitrairement ce nombre.

La classification de Chomsky

Le linguiste N. Chomsky a classifié les grammaires en 4 grandes catégories selon la puissance d'expression des règles utilisées, c'est-à-dire en se donnant des restrictions particulières sur la forme des productions. Plus on ajoutera de contraintes, plus il sera difficile de donner une grammaire décrivant la structure d'un langage donné mais en contrepartie on pourra obtenir des algorithmes plus efficaces pour tester l'appartenance d'un mot au langage décrit par cette grammaire.

- " Grammaires de type 0 " : aucune restriction sur la forme des productions. Cela correspond à la classe des langages dits " récursivement énumérables ", pour lesquels on dispose d'un algorithme pour engendrer les mots du langage. Ceci ne veut pas dire qu'on dispose d'un algorithme pour décider si un mot appartient au langage : si le langage a un nombre infini de mots, on ne peut pas se servir de cet énumérateur pour vérifier si le mot cible est ou non dans le langage !
- " Grammaires de type 1" : il existe deux caractérisations dont il a été montré qu'elles engendraient la même classe de langages :

  « Grammaires monotones » : toutes les règles sont " croissantes ", de la forme α `::=` β avec $|\alpha| \leq |\beta|$, en notant $|\ |$ l'opérateur qui renvoie la taille d'une chaîne.

  « Grammaires avec contexte » : les règles sont de la forme $\gamma X \delta$ `::=` $\gamma\beta\delta$ avec $\gamma \in (V_t \cup V_n)^*$, $\beta \in (V_t \cup V_n)^+$ et X un non-terminal : on ne remplace X par β que dans un contexte où l'occurrence de X est encadrée par les chaînes γ et δ qui sont laissées inchangées par la règle. La partie β ne peut être vide donc les grammaires avec contexte sont des cas particuliers de grammaires monotones.

  Avec ces définitions, ε ne peut appartenir au langage défini par la grammaire. On peut étendre les définitions pour autoriser la règle `S ::=` ε, pour autoriser l'appartenance de ε au langage de la grammaire, en imposant alors que `S` n'apparaisse pas en partie droite de production[1].

[1] La non décroissance des formes sentencielles produites garantit qu'on peut décider simplement (quoique très inefficacement) de l'appartenance d'un mot quelconque au langage engendré par la grammaire. Il suffit d'engendrer par tailles croissantes toutes les formes sentencielles possibles jusqu'à soit obtenir le mot considéré, soit dépasser sa taille.

- " Grammaires de type 2 " (ou " hors contexte " ou " algébriques ") : la partie gauche de chaque production est réduite à un **unique** non-terminal[2]. Pour ces grammaires, il existe deux algorithmes classiques (dus à Coke-Younger-Kasami et Earley) pour décider de l'appartenance d'un mot au langage engendré par la grammaire, de complexité en temps cubique en la taille du mot à reconnaître. Pour des langages de programmation cette complexité est trop élevée et on utilisera des méthodes qui travaillent en temps linéaire, sur une sous-classe de ces grammaires.
- " Grammaires de type 3 " : c'est une grammaire algébrique dans laquelle chaque partie droite de règle comporte **au plus un non-terminal** qui est alors forcément le symbole **le plus à droite** de la règle (on parle de « **grammaires linéaires droites** "). On peut être encore plus restrictif et demander qu'il y ait au plus un symbole terminal, quitte à découper une règles en plusieurs règles à l'aide de nouveaux non-terminaux.

Les grammaires `G1` et `G2` sont " linéaires droites ", tandis que les trois autres grammaires en exemple sont algébriques et non « linéaires droites ». Un langage peut être engendré par des grammaires de types différents, par exemple L(`G2`) = L(`G2'`). On définit le **type d'un langage** comme le type le plus élevé d'une grammaire qui l'engendre.

Dans le cas des langages informatiques, la classe des grammaires algébriques est celle qui donne le meilleur compromis *pouvoir d'expression* / *performances des algorithmes de reconnaissance*. Les restrictions sur la forme des règles ne sont pas trop pénalisantes, quitte à reporter à la phase de vérification contextuelle certaines contraintes du langage. Nous serons aussi amenés à ajouter des contraintes " techniques " sur les grammaires, de façon à obtenir des algorithmes d'analyse syntaxique fonctionnant en temps et espace linéaires par rapport à la taille du mot à reconnaître.

Remarques :

- Les grammaires linéaires droites engendrent exactement les langages qui peuvent être reconnus par automates à états finis (et donc décrits par des expressions régulières).
- Au lieu des grammaires linéaires droites, on peut définir les grammaires linéaires gauches, dans lesquelles au plus un non-terminal peut apparaître dans une partie droite de règle, et forcément à la position la plus à gauche. On obtient la même classe de langages. Si on autorise dans une grammaire à la fois des productions linéaires gauches et des productions linéaires droites (grammaires « linéaires »), ou si on autorise un terminal de part et d'autre du non-terminal, on sort de la classe des grammaires de type 3 puisqu'on obtient par exemple `G4` dont le langage ne peut être reconnu par un automate à états finis.

Ces classes de grammaires induisent des inclusions strictes sur les classes de langages engendrables. Pour chaque type *i* de grammaires, il existe un langage qui peut être décrit par une grammaire de ce type mais pas par une grammaire de type *i+1*. C'est le cas *i = 2* avec $\{ a^n b^n / n > 0 \}$; pour les langages de type 1, citons $\{ a^n b^n c^n / n > 0 \}$, $\{ a^n b^m c^n d^m / n,m > 0 \}$ ainsi que $\{ ww / w \in V_t^* \}$.

On rappelle qu'un problème (à réponse vrai/faux) est « *décidable* » s'il existe un algorithme qui répond en un temps fini **pour toute instance** du problème. Les propriétés décidables sur une grammaire dépendent en général de la classe de cette grammaire. Nous en citons quelques unes à titre d'exemples :

- Deux grammaires engendrent-elles le même langage : soient G1 et G2 deux grammaires, peut-on décider si L(G1) = L(G2) ?

  Oui si G1 et G2 sont de type 3, non sinon.

- L'appartenance d'un mot au langage décrit par une grammaire : soit G une grammaire et $w \in V_t^*$ a-t-on $w \in L(G)$?

  Oui si G est de type 1, 2 ou 3 (en dehors de toute contrainte de performance !)

[2] On pourra aussi imposer, comme dans le cas précédent, que seul S a le droit de dériver ε et dans ce cas il n'apparaît pas en partie droite de production.

- Soit G une grammaire, existe-t-il une grammaire G' **de même type** qui engendre le complémentaire de L(G) ?

  Oui si G est de type 3, non sinon.

- Soient G1 et G2 deux grammaires de même type, existe-t-il une grammaire **de même type** pour engendrer L(G1) ∪ L(G2) ? Idem pour (G1) ∩ L(G2) ?

  union : oui pour tous les types de grammaires.
  intersection : oui pour les grammaires de types 3, non sinon

  Par exemple $\{ a^n b^n c^m / n, m > 0 \}$ et $\{ a^m b^n c^n / n, m > 0 \}$ sont deux langages de type 2 dont l'intersection est un langage de type 1.

Dans la suite, on ne s'intéresse plus qu'aux grammaires de type 2. Les grammaires de type 3 sont supposées traitées par des automates à états finis, celles de type 1 posent des problèmes de performances que ne compense pas, pour un langage de programmation, le gain d'expressivité.

Nous donnons deux propriétés désirables pour un symbole non-terminal :

Définitions :
- Un symbole non-terminal est **productif** s'il est capable d'engendrer au moins un mot constitué uniquement de terminaux.
- Un symbole non-terminal est **accessible** s'il apparaît dans au moins une forme sentencielle dérivable à partir de l'axiome.

Seuls les non-terminaux productifs et accessibles contribuent à la définition du langage engendré par une grammaire. Les autres symboles sont inutiles et peuvent être éliminés (avec les règles dans lesquelles ils apparaissent) sans changer le langage reconnu. Dans la suite, les grammaires sont supposées **sans** symboles improductifs ou inaccessibles. Les symboles productifs et accessibles peuvent être calculés à l'aide des algorithmes en pseudo-code ci-dessous.

Détection des symboles accessibles:
-- A_i est l'ensemble des symboles qu'on peut prouver accessibles en au plus *i* étapes à partir de S

$A_0 := \{ S \}$; i := 0 ;
répéter
  i := i + 1 ;
  $A_i := A_{i-1} \cup \{ X \in V_n / \exists Y ::= \alpha X \beta \in G : Y \in A_{i-1} \}$;
tant que $A_i \neq A_{i-1}$

Détection des symboles productifs:
-- A_i est l'ensemble des symboles qu'on peut montrer productifs en au plus *i* étapes de dérivation en parallèle :

$A_0 := \{ Y \in V_n / \exists Y ::= w \in G : w \in V_t^* \}$; i := 0 ;
répéter
  i := i + 1 ;
  $A_i := A_{i-1} \cup \{ X \in V_n / \exists X ::= x_1 \ldots x_n \in G : \forall 1 \leq j \leq n, x_j \in V_t \vee x_j \in A_{i-1}) \}$;
tant que $A_i \neq A_{i-1}$

Les deux algorithmes construisent des ensembles de cardinalités croissantes mais bornées par la taille de V_n. Ils terminent donc toujours. Des algorithmes peuvent être bâtis selon le même schéma pour calculer d'autres propriétés des grammaires (*Exercice* : donnez un algorithme pour calculer l'ensemble des symboles qui peuvent dériver, directement ou non, le mot vide ε).

Ambiguïté, arbres syntaxiques et arbres abstraits

Dès que plus d'un non-terminal apparaît dans une forme sentencielle, l'ordre d'application des productions est arbitraire, et on peut poursuivre la dérivation à partir de n'importe lequel d'entre eux. La notion d'arbre syntaxique permet de décrire la combinaison des règle utilisées pour prouver l'appartenance d'un mot au langage engendré par une grammaire, indépendamment de l'ordre dans lequel les productions ont été utilisées :

Définition : Soit G une grammaire d'axiome S et *w* un mot de L(G), on appelle **arbre syntaxique** de *w*, un arbre étiqueté tel que :

- la racine est étiquetée par S,
- chaque nœud étiqueté par un non-terminal X a pour successeurs des nœuds dont la suite ordonnée des étiquettes est $X_1X_2 \ldots X_p$, où X `::=` $X_1X_2 \ldots X_p$ est une règle de G.
- le mot formé par la concaténation des étiquettes des feuilles est égal à *w* (ε est élément neutre pour la concaténation et donc " invisible " dans cette concaténation).

Une grammaire est dite **ambiguë** s'il existe au moins un mot de son langage qui admet plus d'un arbre syntaxique.

L'ambiguïté est une propriété de la grammaire : un langage peut être décrit par une grammaire ambiguë, mais aussi peut-être par une grammaire non ambiguë. Un langage est **ambigu de façon inhérente** s'il ne peut être engendré une grammaire non-ambiguë. Citons deux propriétés :

- Il est indécidable de savoir si une grammaire est **ambiguë**.[3]
- Il existe des langages ambigus de façon inhérente. C'est par exemple le cas du langage $L = \{ a^n b^n c^m / n, m \geq 0 \} \cup \{ a^n b^m c^m / n, m \geq 0 \}$. Toute grammaire pour L a forcément deux arbres syntaxiques pour les mots de la forme $a^n b^n c^n$.

Ci-dessous, nous décrivons le langage des expressions arithmétiques réduites aux opérations `+` et `*` et aux identificateurs (`ident`) par trois grammaires différentes :

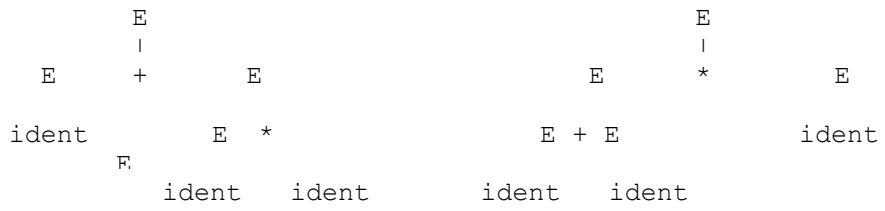
```
      G1 :                 G2 :                  G3 :
E ::= E + E          E  ::= T E'           E ::= E + T | T
    | E * E          E' ::= + T E' | ε     T ::= T * F | F
    | ident          T  ::= F T'           F ::= ( E ) | ident
    | ( E )          T' ::= * F T' | ε
                     F  ::= ( E )  | ident
```

La grammaire `G1` est la plus concise mais est ambiguë, comme on peut le voir pour le mot `ident+ident*ident` qui admet deux arbres syntaxiques, la grammaire ne fixant pas les **précédences** (priorités) des opérateurs `+` et `*`. Elle ne fixe pas non plus **l'associativité** des opérateurs binaires : le mot `ident+ident+ident` admet aussi deux arbres syntaxiques. Ce n'est pas forcément gênant, les deux arbres étant sémantiquement équivalents puisque `+` est associatif (quoique les deux arbres peuvent

[3] Rappelons que ceci veut dire qu'il n'existe pas d'algorithme universel capable, quelle que soit la grammaire, de déterminer si cette grammaire est ambiguë ou non. Pour une grammaire donnée on peut parfois prouver son éventuelle ambiguïté par une technique appropriée. C'est l'existence d'une technique universelle qui est impossible.

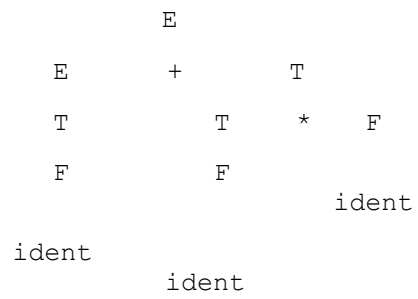
différer dans leurs comportements vis-à-vis d'éventuels dépassements de capacité numérique), mais il n'en serait pas de même si on avait considéré la soustraction au lieu de l'addition.

Les deux arbres syntaxiques pour `ident + ident * ident` avec `G1` :



La grammaire G3 résout les problèmes d'ambiguïté de G1 en reflétant les niveaux de précédence et l'associativité des opérateurs dans ses productions : on utilise un non-terminal pour chaque niveau de précédence d'opérateur, les non-terminaux associés aux niveaux les moins prioritaires étant les plus proches de l'axiome afin que dans un arbre syntaxique les opérateurs prioritaires soient les plus proches des feuilles de l'arbre, c'est-à-dire des opérandes. Une expression dérivée à partir de E peut contenir des sous-expressions contenant des + et des *, alors qu'une expression issue de T ne peut contenir directement que des symboles *, l'opérateur + ne pouvant ré-apparaître que dans des sous-expressions parenthésées. L'associativité est aussi reflétée dans les règles : les opérateurs étant associatifs à gauche, les productions sont récursives gauche[4], de façon que les opérandes les plus à gauche d'une expression soient associés aux occurrences les plus profondes de l'opérateur concerné.

Nous donnons ci-dessous l'arbre syntaxique pour `ident + ident * ident` avec G3 :

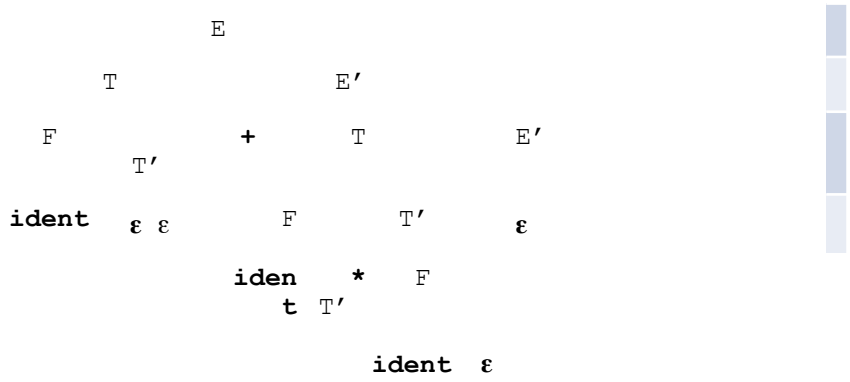


Le principe se généralise à de nouveaux opérateurs : ajoutons un opérateur `'-'` de même priorité et associativité que `'+'`, de même que `'/'` relativement à `'*'`, et un opérateur `'^'` d'exponentiation, muni d'une priorité maximale et associatif **à droite**. Nous obtenons alors la grammaire suivante :

```
G3':
E::= E + T | E - T | T
T::= T * F | T / F | F
F::= G ^ F | G
G::= ( E ) | ident
```

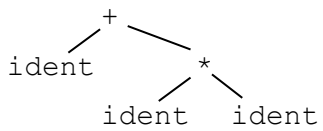
Enfin, G2 est obtenue à partir de G3 en éliminant la récursivité gauche des productions. Illustrons la transformation sur les productions `E ::= E + T | T`, en considérant temporairement T comme un symbole termina : le langage engendré à partir de E est { T, T + T, T + T + T, ... }, c'est-à-dire `T (+ T)*`. Ce même langage peut être engendré par la grammaire non récursive gauche: `E::= T E'`, `E'::= + T E' | ε`. Nous donnons ci-dessous l'arbre syntaxique pour le mot `ident+ident*ident` avec G2. L'arbre syntaxique de l'expression est moins naturel qu'avec G1 ou G3, ce qui pourra gêner les étapes suivantes de la compilation (vérifications contextuelles et génération de code). De même la représentation obtenue reflète mal l'associativité des opérateurs !

[4] Une grammaire est **récursive gauche** s'il existe un non terminal X et une forme sentencielle α telle que $X \rightarrow^{+} X\,\alpha$.



Arbre abstrait (ou « arbre de syntaxe abstraite » ou encore « AST »)

Un arbre abstrait est une autre représentation, proche de l'arbre syntaxique mais épurée de ce qui est inutile notamment les symboles terminaux tels que les mots-clefs du langage ou les symboles divers. Dans l'arbre abstrait, on ne retient que la structure générale du mot, souvent sous la forme d'un arbre dit *opérateur/opérandes*, dans lequel l'étiquette de la racine indique la construction utilisée et les sous-arbres sont les opérandes (au sens large) de la construction. Dans l'exemple du mot `ident + ident * ident` on voudrait obtenir un AST qui ressemble à l'arbre ci-dessous, que ce soit pour la grammaire G1 ou G3 : les niveaux de priorité reflétés dans G3 par les non-terminaux ont servi à lever les ambiguïtés de G1, mais les dérivations E → T → F , et les branches filiformes associées dans l'arbre syntaxique, n'apportent plus d'informations pour la suite : ce qui nous intéresse c'est le calcul représenté. Un AST sera construit à partir des règles de la grammaire mais sans pour autant refléter exactement la grammaire.



Avec la grammaire G2, obtenir le même AST demanderait plus de travail car dans cette grammaire les sous-arbres qui représentent les opérandes d'un opérateur binaire ne sont pas introduits dans une même règle. Par exemple avec `E ::= T E′` : le sous-arbre associé à `T` représente une expression dont l'éventuel opérateur associé et son second opérande sont introduits sous `E′` avec la règle `E′::= + T E′` (ou, s'il n'y a pas d'opérateur, on utilise `E′::= ε`).

Plus généralement, on s'arrange pour représenter par un même arbre abstrait des constructions dont la syntaxe diffère mais qui seront traitées sémantiquement de façon similaire. Par exemple, on peut représenter de la même façon dans un AST les constructions `if-then` et un `if-then-else` en ajoutant une partie `else` vide au premier cas ; ou encore transformer en un format unique les différentes formes de boucles d'un langage. On simplifie ainsi la suite des traitements ultérieurs.

Représentation d'un AST

Dans notre futur compilateur, le rôle d'un analyseur syntaxique, en parallèle avec la reconnaissance du mot, sera de construire un AST qui représente le programme source

Un langage comme OCAML permet de représenter facilement des AST et de les construire :

```
type expType =
  Id of string
| Plus of expType*expType
| Minus of expType*expType
| Times of expType*expType
| Div of expType*expType
```

Au mot `ident+ident*ident` dans lequel on suppose que les identificateurs en question sont `x`, `y` et `z`, on associe la valeur `Plus(Id "x", Times (Id "y", Id "z"))`.

L'AST peut être construit en associant des «calculs» aux productions de la grammaire, soit par le bais d'une fonction qui parcourra récursivement l'arbre syntaxique soit par un calcul à l'analyse syntaxique et qui construira l'arbre des feuilles vers la racine. À chaque production on associe un calcul qui définit l'AST pour le sous-arbre dont la racine est la partie gauche de règle à partir des AST supposés construits pour les non-terminaux de la partie droite de la règle.

Pour la grammaire G1, à la règle `E::= ident` on associe la valeur `Id ident.lexval`[5]. À la règle `E::= E + E`, en supposant qu'on dispose de l'AST de l'opérande gauche dans une variable `g` et celle de l'opérande droit dans `d`, la valeur pour le sous-arbre est `Plus(g, d)`.

Pour la grammaire G3, on associe le même calcul pour `F::= ident`. À la règle `E::= E + T`, avec les conventions précédentes, on associe la valeur `Plus(g, d)` tandis que la règle `E::= E + T` se contente de remonter la valeur produite pour `T`. De même pour les règles de partie gauche `T`. Comme déjà indiqué, le problème serait plus compliqué pour la grammaire G2. Partir d'une grammaire « naturelle » facilite aussi bien sa compréhension (donc sa correction) que les traitements ultérieurs ;

Dans un langage objet, on pourra aussi le faire avec une hiérarchie de classes adaptée !

[5] On rappelle que pour une unité lexicographique, le champ `lexval` permet de retrouver sa valeur, ici le texte de l'identificateur en question à cet endroit du programme.