Livret - 2

Programmer avec un langage

Historique, relations binaires, théorie des langages.

RM di scala

Cours informatique programmation

Rm di Scala - http://www.discala.net

SOMMAIRE

2.1 : Les langages

Plan du chapitre:

1.Historique des langages

1. Historique des langages de programmation

La communication entre l'homme et la machine s'effectue à l'aide de plusieurs moyens physiques externes. Les ordres que l'on donne à l'ordinateur pour agir sont fondés sur la notion d'instruction comme nous l'avons déjà vu. Ces instructions constituent un langage de programmation. Depuis leur création, les langages de programmation ont évolué et se sont diversifiés.

Schématiquement il est possible de les classer en cinq catégories :

1° Les langages procéduraux ou impératifs.
2° Les langages fonctionnels.
3° Les langages logiques.
4° Les langages objets.
5° Les langages de spécification.

L'un des principaux objectifs d'un langage de programmation est de permettre la construction de logiciels ayant un minimum de qualités comme la fiabilité, la convivialité, l'efficacité.
Il faut connaître l'histoire des langages et se rendre compte qu'à ce jour, malgré les nouveaux langages du marché et leur efficacité, c'est **Cobol** qui est le plus utilisé (numériquement 200 milliards de lignes Cobol seraient intégrées à des applications existantes [programmez, n°63 Avril 2004] dont 5 milliards de lignes nouvelles chaque année) dans le monde.

L'investissement intellectuel et matériel prédomine sur la nouveauté. Cette remarque est la clef de la compréhension de l'évolution actuelle et future des langages.

Les langages ont fait leurs premiers pas directement sur des instructions machines écrites en binaire, donc rudimentaires sur le plan sémantique. Les améliorations sur cette catégorie de langages se sont limitées à construire des langages symboliques (langage avec mnémonique) et des macro-assembleurs. J.Backus d'IBM avec son équipe a mis au point dès 1956-1958 le premier langage évolué de l'histoire, uniquement conçu pour le calcul scientifique (à l'époque l'ordinateur n'était qu'une calculatrice géante).

Les années 70 ont vu s'éloigner un rêve d'informaticien : parler et communiquer en langage naturel avec l'ordinateur.
Actuellement les langages évolués se diversifient et augmentent en qualité d'abstraction et de convivialité.

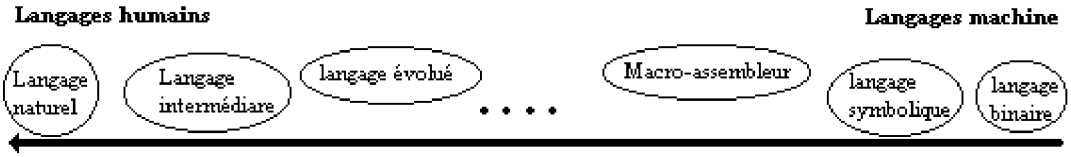


fig : classification sur un axe d'abstraction : de la machine à l'homme

Les langages majoritairement les plus utilisés actuellement sont ceux qui font partie de la catégorie des langages procéduraux ou Hybrides. Les ordinateurs étant des machines de Turing (améliorées par von Neumann), la notion de mémoire machine est représentée par la

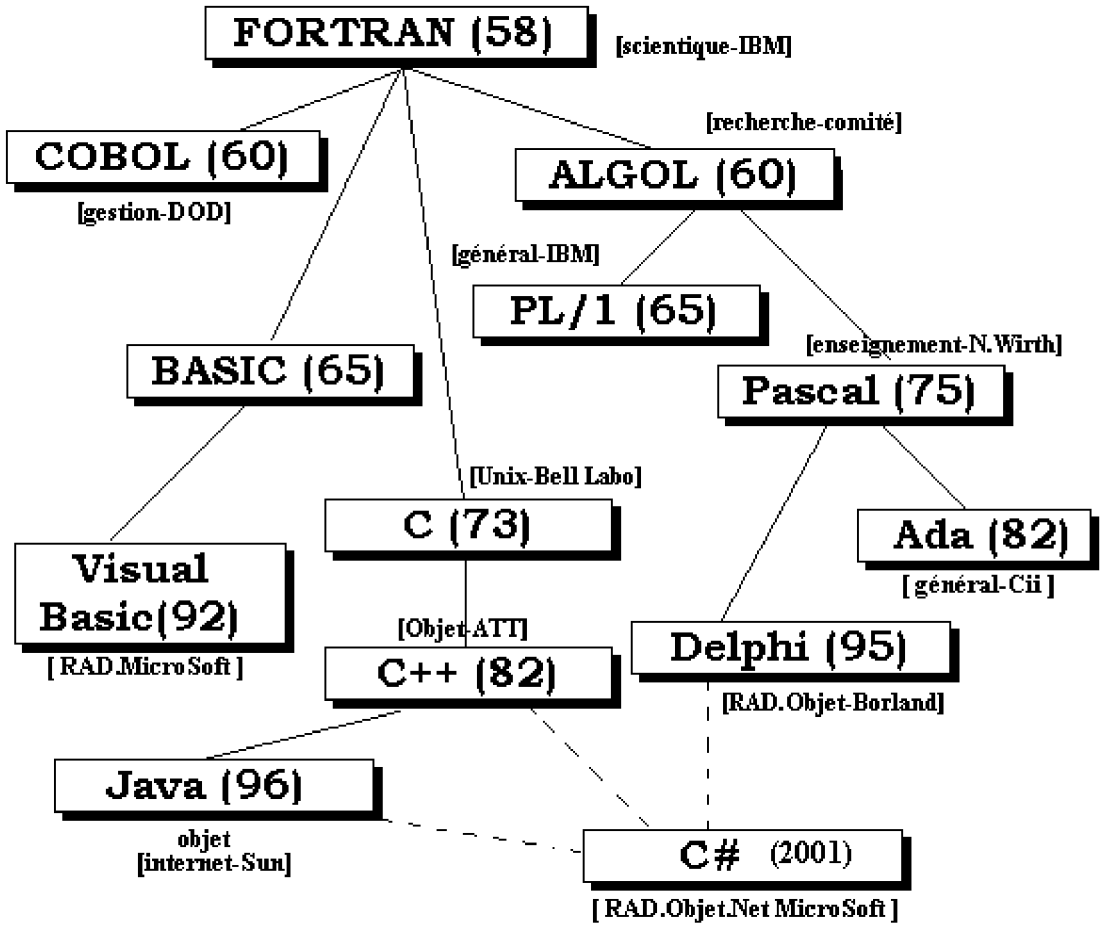
donnée abstraite qu'est une variable, dans un langage procédural. D'autre part, les machines de Türing sont séquentielles et les langages impératifs traitent les instructions séquentiellement. Ceci indique que les langages procéduraux sont parfaitement bien adaptés à l'architecture de l'ordinateur ; ils sont donc plus " facilement " adaptables à la machine.

1.1 Les langages procéduraux ou impératifs

Tous les langages procéduraux ont un ancêtre commun : le langage **FORTRAN**.

Voici un arbre généalogique (non exhaustif) de certains langages connus. Pour chaque langage nous avons indiqué quelques éléments de référence.

Par exemple : **FORTRAN (58) [scientifique - IBM]** signifie que le premier compilateur commercial a été diffusé environ en 1958, que le domaine d'activité pour lequel le langage a été élaboré est le domaine du calcul scientifique, enfin qu'il s'agit d'un acte commercial puisque c'est la compagnie IBM qui l'a fait réaliser.



Dans cette courte liste, seuls **Algol**, **Basic** et **Pascal** sont des langages qui ont été conçus par des équipes dans des buts de recherche ou d'enseignement. Les autres langages sont élaborés par des firmes et des compagnies dans des buts de commercialisation, de rationalisation des coûts de gestion (DOD) etc... Les langages de programmation, comme le reste des outils de la science informatique, sont fortement soumis aux règles du marché, ce qui provoque pour cette discipline le pire et le meilleur.

Pour donner les propriétés des autres catégories de langages, nous nous servirons de la catégorie des langages procéduraux comme référence.

Dans un langage procédural, l'affectation (transfert d'une valeur dans une mémoire) est la base des actions sur les données. La catégorie la plus utilisée après les langages procéduraux est celle des langages fonctionnels.

1.2 Les langages fonctionnels

Dans un langage fonctionnel, les actions reposent sur des fonctions mathématiques ou non qui renvoient des résultats.

Un langage fonctionnel est essentiellement composé d'un dictionnaire de fonctions prédéfinies et d'un mécanisme de construction de nouvelles fonctions par l'utilisateur.

Citons quelques représentants des langages fonctionnels :
LISP :(LISt Processing - 1962) en fait c'est essentiellement un langage de traitement de listes.
SCHEME : c'est un dialecte pédagogique épuré(1975) de LISP.
ML : langage fonctionnel moderne(1990) classé dans la catégorie des langages fonctionnels fortement typés (l'INRIA diffuse gratuitement sur micro-ordinateur une version CAML-Light pour l'enseignement). CAML est utilisé actuellement pour l'enseignement de l'informatique dans les classes préparatoires aux grandes écoles scientifiques françaises.

1.3 Les langages logiques

Citons la catégorie des langages de programmation en logique et son principal représentant :

PROLOG (**PRO**grammation en **LOG**ique - 1982).
Dérivé de l'intelligence artificielle, il oblige le programmeur à penser ses actions en termes de buts et à en faire une description relationnelle (vision déclarative).

Le langage Prolog est fondé sur un moteur d'inférence d'ordre 1 (logique des prédicats), et permet l'exploration exhaustive automatique de différents chemins amenant à des solutions. Il possède une qualité intéressante : il est possible d'interpréter un programme prolog d'une manière *déclarative* ou d'une manière *procédurale*.

Le Groupe d'Intelligence Artificielle de Marseille-Luminy fournit des prologs sur micro-ordinateurs à travers la société PrologIA.

1.4 Les langages orientés objets (L.O.O)

Les langages à objets : ils sont fondés sur une seule catégorie d'éléments : " les objets " qui communiquent entre eux grâce à l'envoi de messages (grâce à des opérateurs appelés méthodes). Par rapport à un langage impératif typé, un objet est l'équivalent (mutatis mutandis) d'une variable (simple ou structurée) et la classe dont il est l'instance correspond au type de la variable.

SIMULA-67 (1967) est le premier langage objet, **SMALLTALK-80**(1980) est un environnement de développement purement objet, **Eiffel**(1990) est un langage objet tourné vers le génie logiciel et la réutilisabilité.

1.5 Les langages de spécification

Les langages de spécification sont encore du domaine de la recherche. Leurs objectifs sont de décrire le plus rigoureusement possible (les modèles principaux sont mathématiques) un logiciel afin de pouvoir le valider et le vérifier.

Nous ne mentionnerons ici que le langage **LPG** de D.Bert(Grenoble) pour les spécifications de types abstraits algébriques, **Z** de J.R. Abrial, le langage dont la notation est fondée sur la théorie des ensembles (puis d'une amélioration de **Z** dénotée **B** par Abrial) et **VDM** langage formel de spécification par pré-condition et post-condition. Ces langages ne peuvent être utilisés d'une manière pratique que sous forme de notation, bien qu'ils soient implantés sur des systèmes informatiques. Ils ne sont pas encore à la disposition du grand public comme les langages des catégories précédentes, bien que certains soient utilisés dans des sites industriels. Par la suite, nous utiliserons un langage de spécification pédagogique fondé sur les types abstraits algébriques.

1.6 Les langages hybrides

Une mention spéciale ici pour des concepts hybrides qui peuvent être de bons compromis entre des catégories différentes. Les concepteurs de tels langages essaient d'importer dans leur langage les qualités inhérentes à au moins deux catégories. La catégorie la plus utilisée est celle des langages impératifs.

Par exemple, la plupart des langages impératifs purs cités plus haut bénéficient d'une " extension " objet, comme **C++** qui est une extension orientée objet du langage **C** conçu à l'origine pour écrire le système d'exploitation Unix.

Plus récemment est apparu un langage comme **Delphi** de Borland qui allie l'approche pédagogique et typée du Pascal, l'approche objet du C++ et les approches visuelles et événementielles de Visual Basic de Microsoft (la sortie fin 2001 de la version entièrement orientée objet de VB, dénommée **VB .Net**, procure à Visual Basic un statut de langage hybride).

Enfin, mentionnons l'important langage **Java** de Sun Microsystems qui permet le développement multi-plateforme en particulier pour l'intranet et qui est grandement utilisé malgré son léger manque de rapidité dû à sa machine virtuelle.

Un mot enfin sur le tout récent langage **C#** support de développement de la plateforme Microsoft .Net, qui a été inventé par le père du langage Delphi (C# s'approprie des avantages de Java et de **Delphi**, il suit de très près la syntaxe de **Java** et celle de **C++**)et qui est le fer de lance de la plateforme .Net de microsoft.

Object Pascal, **C++**, **Ada95**, **Java, C#** sont des langages procéduraux qui ont été fortement étendus ou remaniés pour se conformer aux standards objets.

Remarque de vocabulaire:

L'ordinateur ne " comprenant " que le langage binaire, il lui faut donc un "traducteur" qui lui traduise en binaire exécutable, les instructions que l'humain lui fournit en langage évolué.

Cette traduction est assurée par un programme appelé **compilateur**.

Un compilateur du langage L est donc un programme chargé de traduire un programme "source" écrit en L par un humain, en un programme " cible " écrit en binaire exécutable par l'ordinateur.

2.2 : Relations binaires

Plan du chapitre:

1. Rappel et convention

1. Relation binaire *sur un ensemble*
2. Produit de relations *binaires*
3. Représentation matricielle *d'une relation binaire*
4. *Relation* binaire transposée
5. Matrice du produit *de deux relations*
6. Fermeture transitive *d'une relation binaire*
7. Fermeture réflexo-transitive *d'une relation binaire*
8. Algorithmes *de calcul de matrices*
9. Exemple de calcul *sur une généalogie*

1. Rappels et conventions

Un peu de mathématiques utiles, mais pas trop !

En informatique, la notion de relation est importante. Nous indiquons ici sans rentrer dans les détails que le lecteur trouvera dans des livres spécialisés, en particulier sur la recherche opérationnelle, comment on implante une relation binaire à travers sa matrice de représentation. Ceci peut donc être considéré comme un bon exemple d'application des matrices booléennes en informatique.

Convention

Lorsque nous écrivons " x ← a " ceci se lit: **"x vaut la valeur de a"**.

1. Relation binaire sur un ensemble

Nous appelons relation binaire sur un ensemble E non vide, tout sous-ensemble **R** du produit cartésien E x E.

$$\mathbf{R} \subset E \times E$$

Il est donc possible de définir l'union et l'intersection de deux relations binaires.

2. Produit de relations binaires

Soient ρ et σ deux relations binaires sur un ensemble non vide E. On définit le produit des deux relations $\pi = \rho.\sigma$ ainsi :

$$\forall a , a \in E \quad \forall b , b \in E \qquad a \ \rho.\sigma \ b \ \textbf{ssi} \ \exists c , c \in E / (a \ \rho \ c) \text{ et } (c \ \sigma \ b)$$

Nous énonçons brièvement quelques propriétés de ce produit :

- Le produit est associatif.
- Le produit n'est pas commutatif.

Notations

$\rho^n = \rho.\rho....\rho$ (n fois)
ρ^0 , est la relation telle que : $\forall a , a \in E$ on a toujours $a \ \rho^0 \ a$
$\rho^{n+m} = \rho^n.\rho^m$

3. Représentation matricielle d'une relation binaire

Cas où E est un ensemble fini, c'est d'ailleurs le seul cas qui nous intéresse en informatique où nous ne pouvons pas traiter du non fini.

Soit E l'ensemble : E = { a_1, a_2, ..., a_n }

- Soit ρ une relation binaire sur E.
- Soit M une matrice carrée d'ordre n sur {0,1}. Nous notons $((m_{i,j}))$ l'élément générique de la matrice M.

Nous dirons que M est la matrice de représentation de la relation binaire ρ et nous la noterons Mρ, **ssi** par définition :

si a_i ρ a_j **alors** $m_{i,j} \leftarrow 1$ **sinon** $m_{i,j} \leftarrow 0$ **fsi**

Exemple :
E = { 7,8,3 } ; ρ = { (7,8),(7,3),(3,8),(8,7) }
$a_1 = 7$; $a_2 = 8$; $a_3 = 3$

Voici la matrice Mρ de la relation ρ définie ci-haut :

$$M\rho = \begin{array}{c} \begin{array}{ccc} 7 & 8 & 3 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \end{array} \begin{array}{l} 7 \\ 8 \\ 3 \end{array}$$

4. Relation binaire transposée

- Soit E l'ensemble : E = {a_1,a_2,...,a_n}
- Soit ρ une relation binaire sur E.

Nous notons ρ^t la relation binaire telle que :

$\forall a , a \in E$
$\forall b , b \in E$ $\quad a \ \rho^t \ b$ **ssi** $b \ \rho \ a$

Par construction la matrice de ρ^t est la transposée de la matrice de ρ.

$M\rho^t = {}^t M\rho$

5. Matrice du produit de deux relations

En munissant l'ensemble {0,1} d'une structure d'algèbre de boole avec les opérateurs $\wedge$, $\vee$,$\neg$, il nous est possible d'effectuer des calculs sur les matrices de représentation de relations binaires.

- Soient ρ et σ deux relations binaires sur un ensemble non vide E. Soit le produit des deux relations , $\pi = \rho.\sigma$, $M\rho$, $M\sigma$ et $M\pi$ les matrices de ρ, σ et π.
- Soit $((a_{i,j}))$ l'élément générique de $M\rho$.
- Soit $((b_{i,j}))$ l'élément générique de $M\sigma$.

La matrice $M\pi = M\rho . \sigma$ est très exactement par définition le produit booléen en croix des matrices $M\rho$ et $M\sigma$.

$$M\rho \times M\sigma = M\pi = M\rho.\sigma = [\bigvee_{k=1}^{n} (a_{i,k} \wedge b_{k,j})]$$

6. Fermeture transitive d'une relation binaire

- Soit E l'ensemble : $E = \{ a_1, a_2, \ldots, a_n \}$
- Soit ρ une relation binaire sur E.

Nous posons par définition sa fermeture transitive qui est la relation binaire ρ^+ :

$\rho^+ = \bigcup_{n=1}^{\infty} \rho^n$, en fait dans le cas où E est fini l'union se limite à un nombre fini **k** de ρ^n distincts donc :

$$\rho^+ = \bigcup_{n=1}^{k} \rho^n$$

En informatique, les ensembles sont toujours finis donc nous considérons que la fermeture transitive de ρ s'écrit :

$$\rho^+ = \rho^1 \cup \rho^2 \cup \ldots \cup \rho^{k-1} \cup \rho^k$$

7. Fermeture réflexo-transitive d'une relationinaire

- Soit E l'ensemble : E = $\{a_1,a_2,...,a_n\}$
- Soit ρ une relation binaire sur E, sa fermeture transitive $\rho+$.

On note par définition $\rho*$ sa fermeture réflexo-transitive :

$$\rho^* = \rho^+ \cup \rho^0$$

Remarque

Soit un couple (a,b) de E x E tel que a ρ^* b :
a ρ^* b $\Leftrightarrow \exists n , n \in E / a \rho^n b$

Nous dirons dans ce cas qu'il existe " un chemin de longueur n, allant de a vers b ". En effet d'après la définition du produit :

$$a \rho^* b \Leftrightarrow \exists (c_1,\ldots,c_n), \forall k, k \in [1,n], c_k \in E$$
tels que : $(a \rho c_1)$ et $(c_1 \rho c_2)$ ...et $(c_n \rho b)$

8. Algorithmes de calcul de matrices

Calcul de la matrice produit à partir de la formule :

$$M\rho \times M\sigma = M\pi = M\rho.\sigma = [\bigvee_{k=1}^{n} (a_{i,k} \wedge b_{k,j})]$$

Notons $((m_{i,j}))$ l'élément générique de la matrice produit, voici le corps d'un algorithme de calcul de la matrice produit :

```
pour i ← 1 jusquà n faire
 pour j ← 1 jusquà n faire
  S ← 0 ;
  pour k ← 1 jusquà n faire
   S ← S ∨ (a_{i,k} ∧ b_{k,j})
  Fpour ;
  m_{i,j} ← S
 Fpour
Fpour
```

Algorithme de Warshall pour le calcul de la fermeture transitive :

Avec les mêmes notations de l'algorithme précédent,soit $((a_{i,j}))$ l'élément générique de la matrice Mρ, l'algorithme de Warshall calcule $M\rho^+$:

```
pour k ← 1 jusquà n faire
 pour i ← 1 jusquà n faire
  pour j ← 1 jusquà n faire
   ai,j ←ai,j∨ (ai,k ∧ ak,j)
  Fpour ;
 Fpour
Fpour
```

9. Exemple de calcul sur une généalogie

E = l'ensemble des individus d'une même famille depuis plusieurs générations.

Soient **r**, **s** et **t** les relations binaires :

- x **r** y **ssi** x **est le père de** y
- x **s** y **ssi** x **est la mère de** y
- x **t** y **ssi** x **est un enfant de** y

On peut définir les liens familiaux à l'aide des opérations sur les relations binaires :

r^2 = **est grand père paternel de**
s^2 = **est grand mère maternelle de**
r.s = **est grand père maternel de**
s.r = **est grand mère paternelle de** (*non commutativité évidente !)*
$r \cup s$ = **est parent de**
r^n = **est arrière arrière...arrière grand père paternel de**
$(r \cup s)^+$ = **est un ancêtre de** (on voit ici la signification pratique de la fermeture transitive qui relie deux individus par un chemin d'ascendants dans son arbre généalogique)
$u.u^t$ = **est frère ou sœur de** etc ....

2.3 : Théorie des langages

Plan du chapitre:

1. Notations et définitions générales

Un langage est fait pour communiquer. Les humains doivent communiquer avec les ordinateurs : ils ont donc élaboré les bases d'une théorie des langages. Dans ce chapitre nous donnons les fondements formalisés d'une telle théorie autour de la notion de grammaire formelle.

Remarque et convention :

- Certains éléments d'un langage s'appellent les symboles.
- Soit S un ensemble de symboles ($S \neq \emptyset$). Ce sont les éléments indécomposables dans ce langage (c'est-à-dire non exprimables en autres symboles du langage).

Définition *expression sur S*

On appelle expression sur S, toute suite finie de symboles de S.

e : [1, **n**]→ S , **e** est une expression sur S, **n** est un entier naturel, **n** ≥ 1.

(*e est alors un métasymbole décrivant l'expression S*).

Notation:

On désigne **e** par : **e** = $s_1 s_2 s_3 ... s_n$, **n** ≥ 1 où : k, $1 \leq k \leq n$, $s_k \in S$
et par définition e (k) = s_k ($k \in [1, n]$).

On note S+ = { e / ∀e, e expression sur S }
S+ est l'ensemble de toutes les expressions formées sur S.

Définissons deux opérations sur S+ :

L'égalité d'expressions

Soient **e1** et **e2** deux expressions sur S, on définit leur égalité ainsi :

e1 = e2 **ssi**

$$\begin{cases} \exists k , k \geq 1 \\ \mathbf{e1} : [1, k] \rightarrow S \\ \mathbf{e2} : [1, k] \rightarrow S \\ \forall i, 1 \leq i \leq k \quad \mathbf{e1}(i) = \mathbf{e2}(i) \end{cases}$$

la concaténation d'expressions

soient **e**∈ S+ et **f** ∈ S+, on construit le "produit" des deux expressions e.f :

e : [1, n] → S
f : [1, p] → S
e.f: [1, n+p] → S

avec :
e.f(i) = e(i) **ssi** i ∈ [1, n]
e.f(i) = f(i) **ssi** i ∈ [n+1, n+p]

Notation : (la concaténation de 2 expressions sur S)

Soient **e** et **f** deux expressions :
e = $s_1 s_2 s_3 ... s_n$
f = $t_1 t_2 t_3 ... t_p$

e.f est notée : $s_1 s_2 s_3 ... s_n\ t_1 t_2 t_3 ... t_p$

2. Grammaire formelle ou algébrique

Comme dans les langages naturels, les informaticiens ont, grâce aux travaux de N.Chomsky, formalisé la notion de grammaire d'un langage informatique.

2.1 Monoïde

A) Soit A un ensemble fini appelé alphabet ainsi défini :

$A = \{ a_1, ..., a_n \}$ ($A \neq \varnothing$)

Notations :

$A^1 = A$
$A^2 = \{ x_1 x_2 / (x_1 \in A) \text{ et } (x_2 \in A) \}$
$A^3 = \{ x_1 x_2 x_3 / (x_1 \in A) \text{ et } (x_2 \in A) \text{ et } (x_3 \in A) \}$
.....
$A^n = \{ x_1 x_2 .... x_n / \ \forall i, 1 \leq i \leq n , (x_i \in A) \}$

convention

$A^0 = \{ \varepsilon \}$ (appelé séquence vide)

B) On note A^* et A^+ les ensembles suivants :

$A^* = \bigcup_{n=0}^{\infty} A^n$ $A^+ = A^* - \{ \varepsilon \} = A^* - A^0$	$A^+ = \bigcup_{n=1}^{\infty} A^n$

On définit sur A^* une loi de composition interne appelée concaténation, notée $\bullet$:

(x, y) $\rightarrow$ x $\bullet$ y = xy *(noms des symboles accolés)*

La concaténation possède les propriétés suivantes :

- La loi $\bullet$ est associative :
  $(x \bullet y) \bullet z = x \bullet (y \bullet z)$
- l'élément ε est un élément neutre pour la loi $\bullet$:
  $\forall x \in A^*, x \bullet \varepsilon = \varepsilon \bullet x = x$

Définition :

(A^*, $\bullet$) est un monoïde libre

2.2 Grammaire formelle

Notations :

Un alphabet est aussi appelé **vocabulaire** ; une **chaîne** ou un **mot** est un élément d'un monoïde ; la **longueur** d'un mot **x** (ou chaîne) est le nombre d'éléments du vocabulaire qui le compose et elle est notée habituellement |**x**|.

Exemple : Vocabulaire V = { a , b }
x = aaabbaab , x $\in$ V^* et |**x**| = 8

Remarque :

On note $|\boldsymbol{x}|_a$ le nombre de symboles " a " du vocabulaire V composant le mot **x**.
x = aaabbaab $\Rightarrow$ $|\mathbf{x}|_a = 5$ et $|\mathbf{x}|_b = 3$

Définition : C-Grammaire

On appelle **C-Grammaire** (ou, grammaire algébrique de type 2) tout quadruplet :

$G = (V_N, V_T, S, R)$ où :
V_N est un vocabulaire non terminal ou auxiliaire ($V_N \neq \varnothing$)
V_T est un vocabulaire terminal ($V_T \neq \varnothing$)
$S \in V_N$, un élément particulier appelé axiome de G
$V_N \cap V_T = \varnothing$
$R \subset (V_N \cup V_T)^*$ x $(V_N \cup V_T)^*$, R *est un sous-ensemble fini*

Notations :

- R est appelé l'ensemble des règles de la grammaire G ;
- Une règle $r_i \in R$ est de la forme $(A , \alpha) / [A \in V_N$ et $\alpha \in (V_N \cup V_T)^*]$,
  Elle est notée : $\mathbf{r_i : A \rightarrow \alpha}$
- Lorsque $\alpha \in V_T^*$, la règle $r_i : A \rightarrow \alpha$, est dite **règle terminale**.

Nous ne considérerons par la suite que les grammaires dites de type 2 encore appelées grammaires indépendante du contexte (Context Free), dans lesquelles les règles ont la forme suivante :

$$\mathbf{R \subset V_N \times (V_N \cup V_T)^*}$$

2.3 Opérations sur les mots

Soit G une C-Grammaire, $G = (V_N, V_T, S, R)$. On définit sur $(V_N \cup V_T)^*$ une relation binaire appelée " **dérivation directe** " notée $\Rightarrow$ définie comme suit :

Définition : dérivation directe

Soient $a \in (V_N \cup V_T)^*$ et $b \in (V_N \cup V_T)^*$

On note $a \Rightarrow b$ et l'on dit que **b** dérive directement de **a**, ou que **a** se dérive directement en **b**, si et seulement si

1°) $\exists \alpha \in (V_N \cup V_T)^*$
2°) $\exists \beta \in (V_N \cup V_T)^*$
3°) $\exists$ $r_i \in R$, telle que : $r_i : A_i \rightarrow \gamma$

4°) a et b s'écrivent :
$a = \alpha A_i \beta$
$b = \alpha \gamma \beta$

Notation :

> On emploie aussi les termes de " **règle de réécriture** "
> ou de " **règle de dérivation** ".

Nous obtenons un processus de construction des mots de la grammaire G par application de la dérivation directe. Si l'on réitère plusieurs fois ce processus de dérivation directe, on obtient à partir d'un mot, une suite de mots de G. En fait il s'agit de construire la fermeture transitive de la relation binaire $\Rightarrow$. Cette nouvelle relation est appelée la **dérivation dans G** (la dérivation directe en devenant un cas particulier)

Définition : dérivation

On dit que **x** se dérive en **y**, s'il existe une suite finie de dérivations directes permettant de passer de **x** à **y** :
(**x**, x_0, x_1,...,x_n et **y**) étant des mots de $(V_N \cup V_T)^*$on a le chemin suivant :

$$\mathbf{x} \Rightarrow x_0 \Rightarrow x_1 \Rightarrow \ldots x_n \Rightarrow \mathbf{y}$$

on écrit : $x \Rightarrow^* y$, que l'on lit : **x** se dérive en **y.**

$\Rightarrow^*$ est la fermeture transitive de la relation binaire $\Rightarrow$

2.4 Langage engendré par une grammaire

Nous nous intéressons maintenant à toutes les dérivations possibles construites dans G, par application des règles de G, en privilégiant un point de départ unique pour chacune des dérivations.

Nous avons vu que chaque règle de G commençait en partie gauche par un élément de V_N. Nous construisons alors toutes les productions ayant comme point de départ S l'axiome de G. L'ensemble **L** de tous les mots construits s'appelle le " **langage engendré par la grammaire G " :** $L \subset V_T^*$.

Définition : langage engendré

Soit la C-grammaire G, G = (V_N , V_T , S , R)

L'ensemble $L(G) = \{ \mathbf{u} \in V_T^* / S \Rightarrow^* \mathbf{u} \}$
s'appelle le **langage engendré** par G.

Exemple grammaire G_0 :

G_0 : V_N = { S } ,V_T = {a,b}
Axiome : S
Règles
1 : S $\rightarrow$ aSb
2 : S $\rightarrow$ ab

Le langage engendré par G_0 est :

$L(G_0) = \{ a^n b^n / n \geq 1 \}$

2.5 Grammaire d'états finis

Ce sont des C-Grammaires dans lesquelles les parties droites de règles ont une forme particulièrement simple (on classifie d'ailleurs les grammaires algébriques en général en 4 types en fonction de la forme de leurs règles.

Les C-grammaires sont dites de type-2 et les K-grammaires ou grammaires de Kleene sont dites de type-3).

Pour une grammaire de type-3 ou K-grammaire les règles sont de 2 formes :

$A \rightarrow a$ ($a \in V_T$)
ou bien
$A \rightarrow aB$ ($B \in V_N$ et **B** pouvant être égal à **A)**

Exemple :

G_1 : V_N = { S,A}
V_T = {a,b}
Axiome : S
Règles
1 : S $\rightarrow$ a S
2 : S $\rightarrow$ aA
3 : A $\rightarrow$ bA
4 : A $\rightarrow$ b

Le langage engendré par G_1 est :

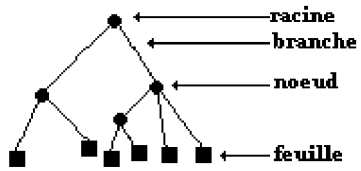
$L(G_1) = \{ a^n b^p / (n \geq 1)$ et $(p \geq 1) \}$

2.6 Arbre de dérivation d'un mot

On appelle **arbre** ***A*** toute structure sur un ensemble E qui est :

- soit une structure vide notée ***A***,
- soit une élément noeud **r** associé à un nombre fini d'arbres disjoints vides ou non : A_1, A_2,, ... , A_n .
- notation : ***A*** = < **r**, A_1, A_2 , ..., A_n >

Représentation graphique d'un arbre :



Un arbre est dit " **étiqueté** " si l'on nomme (attribution d'un symbole de nom) sa racine et ses noeuds.

Définition : arbre de dérivation

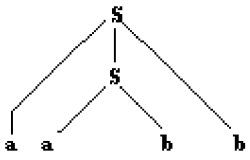
Soit la C-grammaire G, G = (V_N, V_T, S, R).

Un arbre étiqueté est un " **arbre de dérivation** " dans G **ssi** :

- L'alphabet des étiquettes est inclus dans $V_N \cup V_T$.
- Les noeuds sont étiquetés par des éléments de V_N.
- Les feuilles sont étiquetées par des éléments de V_T.
- L'étiquette de tout noeud est un élément de V_N.

Pour tout noeud < **A**, f_1, f_2, ..., f_n> on associe une règle R de la forme : **A** $\rightarrow f_1 f_2 ... f_n$ (règle de dérivation dans G).

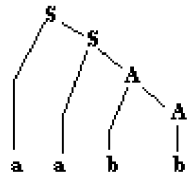
Exemples :



mot a^2b^2 dans G_0

Règles de G_0 appliquées :

$$S \rightarrow^1 aSb \rightarrow^2 aabb$$



mot a^2b^2 dans G_1

Règles de G_1 appliquées :

$$S \rightarrow^1 aS \rightarrow^2 aaA \rightarrow^3 aabA \rightarrow^4 aabb$$

Définition : grammaire ambiguë

Une grammaire est dite **ambiguë** si une chaîne a au moins deux arbres de dérivation différents dans G.

Exemple de grammaire ambiguë :

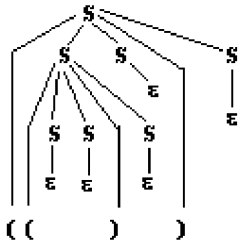
G2 : V_N = {S}
V_T = {(,)}
Axiome : S
Règles
1 : S → (SS)S
2 : S → ε

Le langage engendré par G_2 L(G_2) se dénomme langage des parenthèses bien formées.

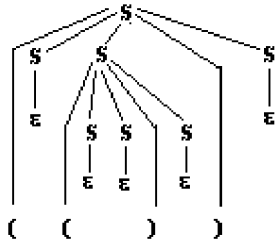
$L(G_2)$ = { ( ) , (( ))(( ))( )), ( )( ), ….. }

Soit le mot (( )) de G_2 , voici 2 arbres de dérivation de (( )) dans G_2 :

Arbre 1 :



Arbre 2 :

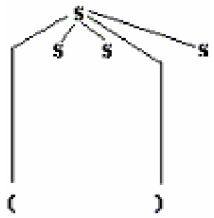


Arbre 1 correspond dans G_2 à la suite des dérivations suivantes :

(afin d'y voir plus clair entre les S choisis nous soulignons les symboles S dérivés)

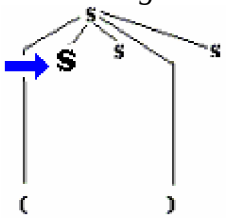
on part de l'axiome **S** *et l'on applique la règle 1:*

$S \rightarrow^1 (SS) S$



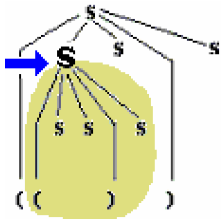
on applique la règle 1 au premier ***S*** *de gauche :*

$S \rightarrow^1 (SS) S$



Ce qui donne :

$S \rightarrow^1 (\underline{S}S) S \rightarrow^1 ((S S) S S) S$



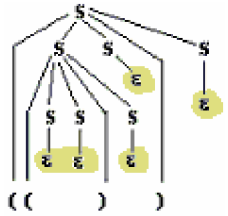
puis on dérive tous les symboles S à partir de la règle 2 :

Règle 2

$S \rightarrow \varepsilon$

appliquée 5 fois à :

((S S) S S) S

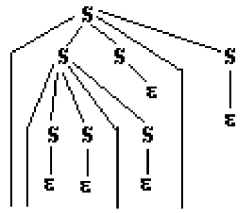


$S \rightarrow \ldots \rightarrow^{2} ((\varepsilon \varepsilon)\varepsilon \varepsilon) \varepsilon$

Le symbole ε est un élément neutre ($x\varepsilon = \varepsilon x = x$) , nous avons donc comme production finale de cette suite de dérivation le mot : $((\varepsilon \varepsilon) \varepsilon \varepsilon) \varepsilon = (( ))$.

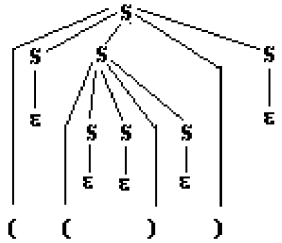
En conclusion, le mot **(( ))** dérive de l'axiome S :

$S \Rightarrow^{*} (( ))$



Arbre1 : (( )) est un arbre de dérivation de mot dans la grammaire G_2 .

Arbre 2 correspond dans G_2 à la suite des dérivations suivante :

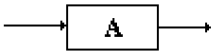
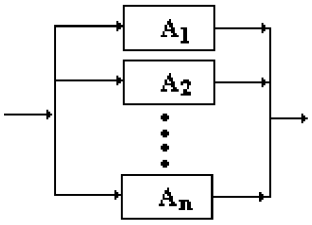
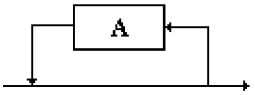
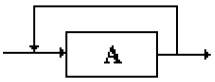


$S \rightarrow^{1} (S S) \rightarrow^{1} (S (S S) S) \; S \rightarrow^{2} \ldots \rightarrow^{2} (\varepsilon (\varepsilon \varepsilon) \varepsilon) \varepsilon = (( ))$.

Le mot **(( ))** dérive de l'axiome S une seconde fois avec un autre arbre de dérivation distinct du précédent, donc la grammaire G_2 est effectivement ambiguë.

2.7 Diagrammes syntaxiques

Il est possible de représenter graphiquement les règles de dérivation d'une grammaire formelle par des diagrammes dénotés " diagrammes syntaxiques ". Cette représentation graphique a pour effet de condenser l'écriture des règles et d'autoriser une meilleure lisibilité.

REGLES	DIAGRAMMES
$\mathbf{A} \in \mathbf{V_N}$	
$\mathbf{a} \in \mathbf{V_T}$	
$\mathbf{B} \rightarrow \varepsilon$	
B → A1 **B → A2** ou encore : **.....** **B → A1|...|An** **B → An**	
B → AB | ε ou : **B → {A}***	
B → AB | A ou: **B → {A}+**	

3. Classification de Chomsky des grammaires

Traditionnellement les grammaires algébriques sont classables en quatre catégories qui se différentient par la forme de leurs règles.

Elles sont notées par leur type (type 0, type 1, type 2, type 3). Il existe une relation d'inclusion provenant de leurs définitions :

type 3 ⊂ type 2 ⊂ type 1 ⊂ type 0

3.1 Les grammaires syntaxiques - type 0

Les règles ont la forme générale suivante : $\alpha \rightarrow \beta$

pour une règle $\alpha \rightarrow \beta$, les symboles (α, β) doivent être de la forme :

$$\alpha \in (V_N \cup V_T)^+$$
$$\beta \in (V_N \cup V_T)^*$$

3.2 Les grammaires sensibles au contexte - type 1

Les règles ont la forme suivante : $\alpha A \beta \rightarrow \alpha \gamma \beta$

pour une règle $\alpha A \beta \rightarrow \alpha \gamma \beta$, les symboles ($\alpha$, β, γ **,** A**)** doivent être de la forme :

$$A \in V_N$$
$$\alpha \in (V_N \cup V_T)^*$$
$$\beta \in (V_N \cup V_T)^*$$
$$\gamma \in (V_N \cup V_T)^+$$

3.3 Les grammaires indépendantes du contexte - type 2

Les règles ont la forme suivante : $A \rightarrow \alpha$

Pour une règle $A \rightarrow \alpha$, les symboles (α, A) doivent être de la forme :

$$\alpha \in (V_N \cup V_T)^*$$
$$A \in V_N$$

3.4 Les grammaires d'états finis ou de Kleene - type 3

Les règles n'ont que deux formes possibles :

$A \rightarrow a$ *ou bien* $A \rightarrow aB$

Pour ces règles, les symboles (a, A, B) doivent être de la forme :

$A \in V_N$
$B \in V_N$
$a \in V_T$

4. Applications et exemples

4.1 Expressions arithmétiques : une grammaire ambiguë

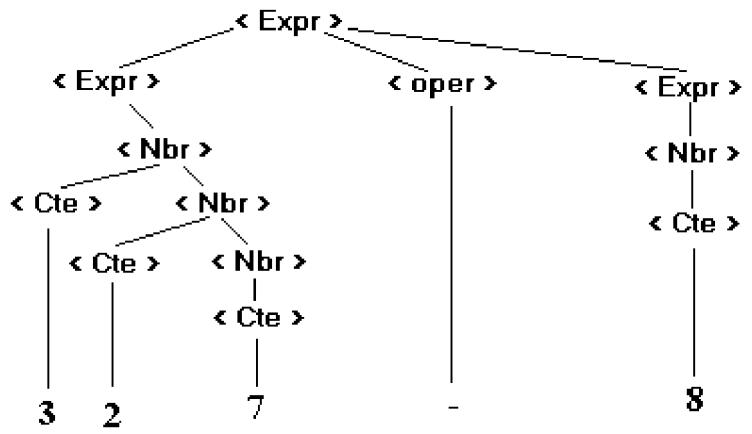
Soit la grammaire $G_{exp} = (V_N, V_T, \text{Axiome, Règles})$

$\mathbf{V_T} = \{$ **0**, ..., **9**, **+**,**-**, **/**, *****, **)**, **(** $\}$
$\mathbf{V_N} = \{$< Expr >, < Nbr > , < Cte > , < Oper > $\}$
Axiome : < Expr >

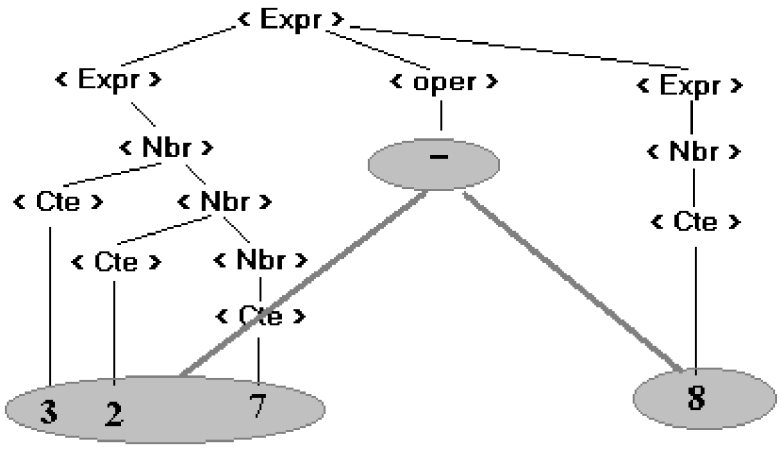
Règles :
1 : < Expr > → < Nbr > | **(**< Expr >**)** | < Expr >< Oper >< Expr >
2 : < Nbr > → < Cte > | < Cte >< Nbr >
3 : < Cte > → **0** | **1** |...| **9**
4 : < Oper > → **+** | **-** | ***** | **/**

Les mots de $L(G_{exp})$ sont des expressions de la forme (x+y-z)*x etc... où x, y, z sont des entiers.

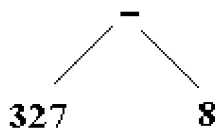
Exemple : **327 - 8** est un mot de $L(G_{exp})$
Ce mot n'a qu'un seul arbre de dérivation dans G_{exp}, dessinons son arbre de dérivation :



Nous pouvons faire ressortir les liens abstraits qui relient les trois éléments terminaux 327, - et 8 :



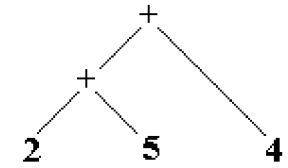
L'arbre obtenu en grisé à partir de l'arbre de dérivation s'appelle l'arbre abstrait du mot " 327-8 ".



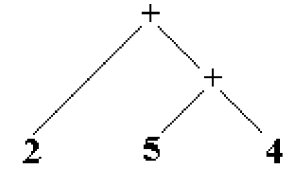
Cet arbre abstrait permet de manipuler la structure générale du mot facilement tout en résumant la structure générale de l'arbre de dérivation.

Soient trois autres mots de $L(G_{exp})$ **2+5+4**, **2-5+4** et **2+5*4**, ils ont chacun deux arbres de dérivation. Nous donnons ci-après deux arbres abstraits de chaque mot.

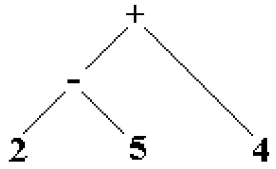
Arbre-1 : 2+5+4



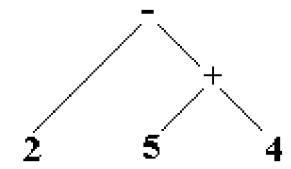
Arbre-2 : 2+5+4



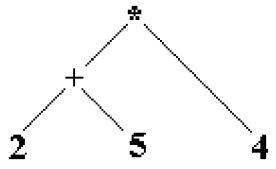
Arbre-3 : 2-5+4



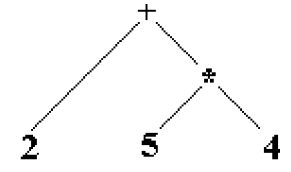
Arbre-4 : 2-5+4



Arbre-5 : 2+5*4



Arbre-6 : 2+5*4



Nous remarquons donc que G_{exp} est une grammaire ambiguë puisqu'il existe un mot possédant au moins deux arbres de dérivation.

Pour l'instant les mots **2+5+4**, **2-5+4** et **2+5*4** ne sont que des concaténations de symboles sans aucun sens particulier

Si nous voulions aller plus loin en donnant un sens (de la **sémantique**) à ces mots de telle façon qu'ils représentent des calculs sur les entiers avec les propriétés classiques des opérations sur les entiers, nous pourrions nous trouver un " *bon choix* " parmi les arbres abstraits précédents.

Nous appellerons ces choix " interpréter " l'expression.

Examen de la situation pour le mot 2+5+4 :

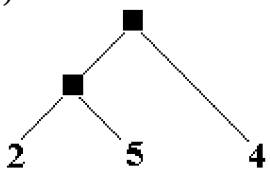
- Arbre-1 s'interprète : (2+5)+4
- Arbre-2 s'interprète : 2+(5+4)

L'opérateur " + " est associatif donc pour notre interprétation les deux arbres 1 et 2 peuvent convenir.

Examen de la situation pour le mot 2-5+4 :

- Arbre-3 s'interprète : (2-5)+4
- Arbre-4 s'interprète : 2-(5+4)

Les opérateurs + et - sont de même priorité et nous obtenons deux expressions différentes selon le choix de l'arbre.
Traditionnellement lorsque deux opérateurs ont la même priorité, l'évaluation se fait à partir de la gauche de l'expression. Donc l'arbre 3 conviendrait.
Nous pourrions penser lever l'ambiguïté en choisissant systématiquement l'arbre abstrait d'évaluation à gauche correspondant à un parenthésage implicite à gauche (comme arbre-1 et arbre-3) :



Nous allons voir ci-dessous que ce n'est pas possible.

Examen de la situation pour le mot 2+5*4 :

- Arbre-5 s'interprète :(2+5)*4
- Arbre-6 s'interprète : 2+(5*4)

Les opérateurs + et * n'ont pas la même priorité. Nous obtenons deux expressions différentes selon le choix de l'arbre. Mais ici c'est le choix de l'arbre 6 qui s'impose à cause de la priorité du * sur le +.

Nous avons fait ressortir le fait qu'il était impossible de privilégier systématiquement pour " l'interprétation " des expressions une catégorie d'arbre plutôt qu'une autre, il faut donc changer de grammaire et éviter l'ambiguïté.

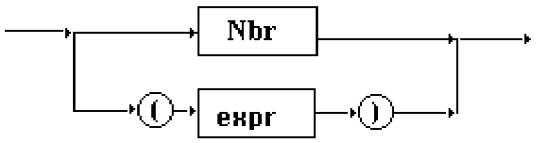
4.2 Expressions arithmétiques : une grammaire non ambiguë

Nous donnons ci-dessous une grammaire non ambiguë basée sur la précédente et tenant compte de la précédence (priorité d'opérateur). Nous séparons les opérateurs en deux catégories ; les opérateurs de priorité zéro (Oper_0) et ceux de priorité un (Oper_1).

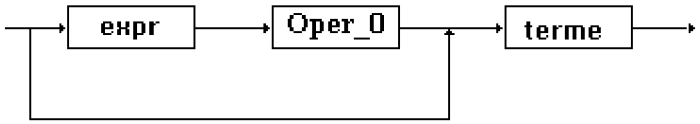
$\mathbf{V_T}$ = { **0**, ..., **9**, **+**,**-**, **/**, *****, **)**, **(** }
$\mathbf{V_N}$ = { < Expr >, < Nbr > , < Cte > , < Oper_0 > , < Oper_1 >, < facteur >, < terme > }
Axiome : < Expr >

Règles :

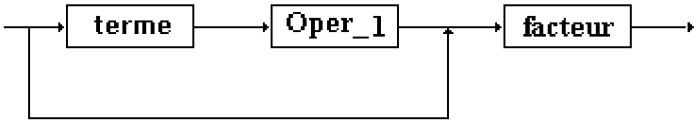
facteur



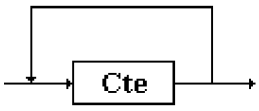
expr



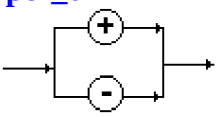
terme



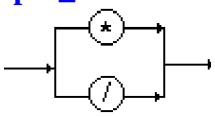
Nbr



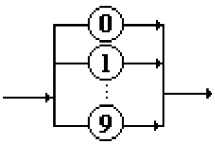
Oper_0



Oper_1



Cte



En pratique ce ne sera pas une telle de grammaire qui sera retenue pour le calcul des expressions arithmétiques car elle contient une règle récursive gauche, ce qui la rend difficilement analysable par des procédés simples.