

 Ceci est une copie d'une conversation entre Claude et Dominique. Le contenu peut inclure des éléments non vérifiés ou dangereux qui ne représentent pas les positions d'Anthropic. La capture partagée peut contenir des pièces jointes et des données non affichées ici.

[Signaler](#)

On définit un langage algébrique L en présentant ses éléments générateurs et ses opérateurs générateurs entre crochet $\langle \dots \rangle$. Chaque opérateur possède une arité indiquée occasionnellement par le suffixe $"(.)"$ pour unaire, $"(.,.)"$ pour binaire, $"(.,.,.)"$ pour ternaire. Exemple : $L = \langle a, b, f(.), g(.,.), h(.,.,.) \rangle$

Très bien, la notation est claire. Récapitulons pour vérifier que nous nous comprenons bien.

Le langage algébrique $L = \langle a, b, f(.), g(.,.), h(.,.,.) \rangle$ est l'ensemble des termes (ou expressions bien formées) construits inductivement à partir de :

- **Éléments générateurs** (constantes, arité 0) : a et b , qui appartiennent à L de base ;
- **Opérateurs générateurs** : f d'arité 1, g d'arité 2, h d'arité 3, qui permettent de construire de nouveaux termes à partir de termes existants.

Formellement, L est le plus petit ensemble tel que :

1. $a \in L$ et $b \in L$;
2. si $t \in L$, alors $f(t) \in L$;
3. si $t_1, t_2 \in L$, alors $g(t_1, t_2) \in L$;
4. si $t_1, t_2, t_3 \in L$, alors $h(t_1, t_2, t_3) \in L$.

Quelques exemples d'éléments de L : $a, f(b), g(a, f(a)), h(b, g(a, b), f(f(a)))$, etc. Chaque terme peut aussi se voir comme un arbre dont les feuilles sont étiquetées par a ou b et les nœuds internes par f, g ou h selon leur nombre de fils.

J'attends la suite : quelle est la question ou l'exercice portant sur ce type de langage (preuve par induction structurelle, comptage de termes, définition d'une fonction sur $L \dots$) ?

On ajoute au langage des variables muettes $V_1, V_2, V_3, \dots$. Elles sont dites muettes car elles ne correspondent pas à de nouveaux éléments générateurs, ce sont juste des inconnues appartenant au langage L , et qui donc jouent le rôle de terme quelconque. Exemple de terme avec variable : $h(a, V_2, g(V_1, V_2))$

Compris. On étend donc la syntaxe avec un ensemble dénombrable de variables $V = \{V_1, V_2, V_3, \dots\}$, ce qui donne l'ensemble des **termes à variables** (souvent noté $L(V)$ ou $T(\Sigma, V)$ dans la littérature), défini inductivement par :

1. $a \in L(V)$ et $b \in L(V)$;
2. $V_i \in L(V)$ pour tout i ;
3. si $t \in L(V)$, alors $f(t) \in L(V)$;
4. si $t_1, t_2 \in L(V)$, alors $g(t_1, t_2) \in L(V)$;
5. si $t_1, t_2, t_3 \in L(V)$, alors $h(t_1, t_2, t_3) \in L(V)$.

Le point important est sémantique : les variables ne sont **pas** de nouvelles constantes. Une variable V_i est un « trou » qui désigne un terme quelconque de L . Un terme avec variables, comme $h(a, V_2, g(V_1, V_2))$, est donc un **motif** (pattern) ou **schéma de terme** : il représente la famille de tous les termes clos de L qu'on obtient en remplaçant chaque variable par un terme de L .

Ce remplacement s'appelle une **substitution** : une application $\sigma : V \rightarrow L$, qu'on étend naturellement aux termes. Par exemple avec $\sigma = \{V_1 \mapsto b, V_2 \mapsto f(a)\}$:

$$\sigma(h(a, V_2, g(V_1, V_2))) = h(a, f(a), g(b, f(a)))$$

Deux remarques qui découlent de la définition :

- Une même variable qui apparaît plusieurs fois dans un terme (comme V_2 ci-dessus) doit être remplacée par le **même** terme à chaque occurrence — c'est ce qui distingue V_2 d'un simple « joker » indépendant.
- Un terme sans variable est dit **clos** (ground) : ce sont exactement les éléments de L au sens initial.

Ces termes à variables servent typiquement à écrire des équations ou des règles valables pour tous les termes (axiomes équationnels, règles de réécriture, filtrage/unification...). Quelle est la suite ?

On définit un automate qui reconnaît les chaînes de caractères correspondants à des termes du langage L . L'automate est représenté par un ensemble de règles de production : $\{ |--a, |--b, |--f(V_1), |--g(V_1, V_2), |--h(V_1, V_2, V_3) \}$.

Chaque élément de cet ensemble est une règle de production possible. Les variables V_1, V_2, V_3 , sont des termes quelconques, et constituent autant d'appels récursifs de cet automate.

Très clair. Cet automate est en fait un **reconnaisseur récursif** (analogue à une grammaire hors-contexte, ou à un analyseur par descente récursive), dont les règles de production sont :

- $\vdash a$
- $\vdash b$
- $\vdash f(V_1)$
- $\vdash g(V_1, V_2)$
- $\vdash h(V_1, V_2, V_3)$

Le fonctionnement se lit ainsi : pour reconnaître une chaîne comme terme de L , l'automate tente de la faire correspondre au motif d'une des cinq règles. Les parties fixes du motif (symboles a, b, f, g, h , parenthèses, virgules) doivent correspondre littéralement aux caractères de la chaîne ; chaque variable V_i du motif déclenche un **appel récursif** de l'automate sur la sous-chaîne correspondante, qui doit elle-même être reconnue comme un terme.

La chaîne est acceptée si et seulement s'il existe une dérivation finie, c'est-à-dire un arbre d'appels récursifs qui termine sur les règles de base $\vdash a$ et $\vdash b$ (les seules sans variable, donc sans récursion).

Exemple : reconnaissance de la chaîne $g(a, f(b))$.

$\vdash g(V_1, V_2)$	la chaîne matche le motif, avec $V_1 \leftarrow "a"$ et $V_2 \leftarrow "f(b)"$
$\vdash a$	appel récursif sur "a" : règle de base, accepté
$\vdash f(V_1)$	appel récursif sur "f(b)", avec $V_1 \leftarrow "b"$
$\vdash b$	appel récursif sur "b" : règle de base, accepté

Tous les appels aboutissent, donc $g(a, f(b))$ est reconnue ; l'arbre de dérivation obtenu est exactement l'arbre syntaxique du terme.

À l'inverse, une chaîne comme $f(a, b)$ est rejetée : elle ne matche aucun motif (f est unaire dans la règle $\vdash f(V_1)$, la virgule au niveau supérieur fait échouer le filtrage), et $g(a)$ échoue de même.

Deux observations en passant :

- Cet ensemble de règles correspond mot pour mot à la définition inductive de L donnée au départ : une règle par générateur, les variables jouant le rôle des prémisses « $t \in L$ ». En notation de grammaire : $T \rightarrow a \mid b \mid f(T) \mid g(T, T) \mid h(T, T, T)$.
- Le langage reconnu n'est pas régulier (le parenthésage imbriqué exige une pile / la récursion), mais il est hors-contexte et même déterministe : le premier caractère de la chaîne suffit à sélectionner l'unique règle applicable, donc la reconnaissance se fait en un seul passage, sans retour arrière.

Je suis prêt pour la suite.

L'usage des chaînes de caractères comme stockage de données constitue un compromis pertinent entre densité des données et lisibilité par un humain. C'est pourquoi on s'intéresse à différentes syntaxe. On modifie la syntaxe comme suit, par la déclaration à 4 composantes suivante :

g infixe 15 +

Cela change la syntaxe classique de l'opérateur binaire g en syntaxe infixe, c'est à dire que l'expression $g(V_1, V_2)$ n'est plus autorisé mais l'expression $V_1 g V_2$ est autorisé pour désigner le même terme. Puis la priorité syntaxique vaut 15. Puis le + signifie que à priorité égale, l'évaluation se fait de gauche à droite tel que par exemple $V_1 g V_2 g V_3$ désigne le terme $(V_1 g V_2) g V_3$.

Parfaitement compris. La déclaration à 4 composantes se lit :

g	infixe	15	+
nom	fixité	priorité	associativité (+ = gauche)

Elle transforme la règle de production de l'automate : $\vdash g(V_1, V_2)$ est remplacée par $\vdash V_1 g V_2$. L'ensemble des règles devient donc $\{\vdash a, \vdash b, \vdash f(V_1), \vdash V_1 g V_2, \vdash h(V_1, V_2, V_3)\}$, et une chaîne comme agb désigne désormais le terme qui s'écrivait $g(a, b)$ en syntaxe préfixe (laquelle n'est plus autorisée pour g).

Les deux dernières composantes servent à **désambiguïser** la lecture, car la syntaxe infixe, contrairement à la syntaxe préfixe pleinement parenthésée, introduit des ambiguïtés :

La priorité (15) détermine la force de liaison de g par rapport aux autres opérateurs infixes qui pourraient être déclarés. Si on déclarait par exemple un autre opérateur k avec la priorité $20 > 15$, alors $a k b g c$ se lirait $(akb) g c$ — l'opérateur le plus prioritaire lie plus fort et se retrouve plus profond dans l'arbre. Tant que g est le seul opérateur infixe, cette composante ne sert à rien face à f et h qui restent préfixes et délimités par leurs parenthèses ; elle prépare le terrain pour de futures déclarations.

L'associativité (+) résout l'ambiguïté entre occurrences de **même priorité** (notamment g avec lui-même) : $V_1 g V_2 g V_3$ pourrait a priori désigner deux arbres distincts, $(V_1 g V_2) g V_3$ ou $V_1 g (V_2 g V_3)$. Le + impose le regroupement à gauche : $agbga$ désigne le terme $(agb)ga$, c'est-à-dire, en revenant à l'arbre abstrait, $g(g(a, b), a)$. Un - (associativité droite, je suppose) donnerait $g(a, g(b, a))$.

Deux conséquences pratiques :

1. **Les parenthèses deviennent nécessaires comme délimiteurs de groupement** pour exprimer l'arbre non conforme à l'associativité par défaut : pour désigner $g(a, g(b, a))$, il faut écrire $ag(bga)$. Les parenthèses changent donc de rôle : elles ne sont plus seulement la ponctuation obligatoire des opérateurs préfixes, mais un mécanisme optionnel de regroupement.
2. **L'automate n'est plus un simple descente récursive pilotée par le premier caractère** : la règle $\vdash V_1 g V_2$ commence par une variable (récursion à gauche), donc la reconnaissance exige une technique adaptée — typiquement un analyseur par précedence d'opérateurs (precedence climbing / Pratt parsing), qui utilise justement les composantes priorité et associativité pour construire l'arbre en un passage.

Exemple complet : la chaîne $f(agb)gh(a, b, bga)$ se décompose en $(f(agb)) g (h(a, b, (bga)))$, soit l'arbre abstrait $g(f(g(a, b)), h(a, b, g(b, a)))$.

J'attends la suite — d'autres déclarations de syntaxe, je présume ?

la déclaration à 3 composantes suivante :

f préfixe 20

Cela change la syntaxe classique de l'opérateur unaire f en syntaxe préfixe (sans parenthèse), c'est à dire que l'expression $f(V_1)$ n'est plus autorisée mais l'expression fV_1 est autorisée pour désigner le même terme. Puis la priorité syntaxique vaut 20. Par exemple $fV_1 g V_2$ désigne le terme $(fV_1)gV_2$. Et à priorité égale l'opérateur unaire est prioritaire sur l'opérateur binaire.

Bien noté. La déclaration à 3 composantes se lit :

f	préfixe	20
nom	fixité	priorité

Il n'y a que 3 composantes car un opérateur unaire préfixe n'a pas besoin d'associativité : ses occurrences en chaîne ne peuvent s'imbriquer que d'une seule façon (ffa ne peut désigner que $f(f(a))$).

L'ensemble des règles de l'automate devient : $\{ \vdash a, \vdash b, \vdash fV_1, \vdash V_1 g V_2, \vdash h(V_1, V_2, V_3) \}$.

Rôle de la priorité 20. Comme $20 > 15$, f lie plus fort que g et se retrouve plus profond dans l'arbre :

- $fagb$ se lit $(fa) g b$, soit l'arbre $g(f(a), b)$ — et non $f(agb)$.
- Pour désigner $f(g(a, b))$, il faut désormais recourir aux parenthèses de groupement : $f(agb)$. Notons que cette chaîne est graphiquement identique à l'ancienne syntaxe $f(V_1)$, mais son interprétation a changé : la parenthèse n'est plus la ponctuation de l'opérateur, c'est un groupement autour de l'argument.

Règle de départage à priorité égale : unaire > binaire. Cette convention supplémentaire résout les cas où un préfixe et un infixé auraient la même priorité. Si f avait été déclaré à 15 comme g , la chaîne `fagb` se lirait quand même $(fa)gb$, l'unaire capturant son argument en premier. Ici la règle est redondante ($20 > 15$ suffit à conclure), mais elle complète la spécification pour les futures déclarations.

Côté portée, le comportement est celui-ci : **f capture l'opérande le plus court possible** compte tenu de sa haute priorité — c'est-à-dire le terme "atomique" qui suit (une constante, un groupe parenthésé, un $h(\dots)$, ou un autre terme préfixé). Ainsi :

- `ffagb` = $((f(fa))\ g\ b = g(f(f(a)),\ b)$
- `agfb` = $a\ g\ (fb) = g(a,\ f(b))$
- `fh(a, b, a)gb` = $(f\ h(a,b,a))\ g\ b = g(f(h(a,b,a)),\ b)$

Exemple de synthèse : la chaîne `fagfbgh(fa, b, agb)` se décompose, avec l'associativité gauche de g , en $((fa)\ g\ (fb))\ g\ h(fa,\ b,\ (agb))$, soit l'arbre abstrait :

$g(g(f(a), f(b)), h(f(a), b, g(a,b)))$

Une remarque sur la densité, puisque c'était la motivation initiale : le terme $g(g(f(a), f(b)), h(f(a), b, g(a,b)))$ demande 33 caractères en syntaxe classique contre 19 en syntaxe déclarée — le gain vient de la suppression des parenthèses et virgules devenues implicites grâce aux priorités.

Prêt pour la déclaration suivante — postfixe, peut-être, ou la syntaxe de h ?

Considérons le langage enrichi d'un opérateur unaire k supplémentaire. Considérons la déclaration à 3 composantes suivante :

k postfixe 12

Cela change la syntaxe classique de l'opérateur unaire k en syntaxe postfixe (sans parenthèse), c'est à dire que l'expression $k(V1)$ n'est plus autorisée mais l'expression $V1k$ est autorisée pour désigner le même terme. Puis la priorité syntaxique vaut 12. Par exemple $fV1k$ désigne le terme $(fV1)k$. A priorité égale, l'opérateur unaire est prioritaire sur l'opérateur binaire, et si les deux opérateurs sont unaires alors la priorité est accordée à l'opérateur de gauche.

Bien noté. Le langage est maintenant $L = \{a, b, f(\cdot), g(\cdot, \cdot), h(\cdot, \cdot, \cdot), k(\cdot)\}$ et la déclaration se lit :

k	postfixe	12
nom	fixité	priorité

Comme pour f , 3 composantes suffisent : un unaire n'a pas besoin d'associativité.
L'ensemble des règles de l'automate devient : $\{ \vdash a, \vdash b, \vdash fV_1, \vdash V_1gV_2, \vdash h(V_1, V_2, V_3), \vdash V_1k \}$.

Effet de la priorité 12. On a désormais l'échelle $f(20) > g(15) > k(12)$: k est le moins liant, donc il se retrouve le plus **haut** dans l'arbre, capturant l'expression la plus large à sa gauche :

- $ak = k(a)$, et $akk = k(k(a))$ — seule imbrication possible pour un postfixe itéré.
- $fak = (fa)k = k(f(a))$, conformément à l'exemple donné : $f(20)$ lie plus fort que $k(12)$.
- $agbk = (agb)k = k(g(a,b))$: $g(15)$ lie plus fort que $k(12)$, donc k s'applique à tout le produit de g .
- Pour obtenir $g(a, k(b))$, il faut parenthéser : $ag(bk)$.

Noter la dissymétrie positionnelle : dans $akgb$, k suit immédiatement a et ne peut capturer que lui (rien à sa droite ne le concerne), d'où $g(k(a), b)$. La basse priorité de k ne joue que vers la gauche : elle détermine jusqu'où il "avale" ce qui le précède.

Les règles de départage, maintenant au complet :

1. Priorités différentes $\rightarrow$ le plus prioritaire lie plus fort (plus profond dans l'arbre).
2. Priorités égales, unaire contre binaire $\rightarrow$ l'unaire gagne.
3. Priorités égales, unaire contre unaire $\rightarrow$ l'opérateur de gauche gagne.

La règle 3 tranche le conflit typique préfixe/postfixe se disputant le même opérande : si f et k avaient tous deux la priorité 20, la chaîne fak verrait f (à gauche) capturer a en premier, donnant $(fa)k = k(f(a))$. Avec les priorités effectivement déclarées (20 vs 12), la règle 1 suffit et donne d'ailleurs le même résultat ; la règle 3 est là pour compléter la spécification.

Exemple de synthèse. La chaîne $fakgbgh(a, bk, agb)k$ se décompose ainsi :

1. $fak = k(f(a))$ — bloc gauche ;
2. $k(f(a)) g b g h(\dots)$ — deux g associatifs à gauche ;
3. dans le h : $bk = k(b)$ et $agb = g(a,b)$;
4. le k final, priorité minimale, coiffe tout.

Arbre abstrait : $k(g(g(k(f(a)), b), h(a, k(b), g(a,b)))))$

Soit 19 caractères contre 37 en syntaxe classique $k(g(g(k(f(a)),b),h(a,k(b),g(a,b))))$.

Petite observation d'ensemble : avec un préfixe très liant ($f, 20$) et un postfixe peu liant ($k, 12$), la syntaxe reproduit des conventions familières — f se comporte comme le signe « $-$ » unaire ou une dérivée, k comme un « $!$ » qui porterait sur toute l'expression précédente.
J'attends la suite.

Considérons le langage enrichie d'un opérateur binaire h supplémentaire.
Considérons la déclaration à 3 composantes suivantes :

h préfixe 5

Cela change la syntaxe classique de l'opérateur unaire h en syntaxe préfixe (sans parenthèse), c'est à dire que l'expression $h(V_1, V_2)$ n'est plus autorisée mais l'expression hV_1V_2 est autorisée pour désigner le même terme. Puis la priorité syntaxique vaut 5. Par exemple hV_1V_2k désigne le terme $hV_1(V_2k)$. A priorité égale, la priorité est accordée à l'opérateur de gauche.

Bien noté. La déclaration se lit :

h	préfixe	5
nom	fixité	priorité

C'est un préfixe **sans parenthèses ni virgule**, appliqué cette fois à un opérateur binaire : on retrouve la notation polonaise (à la Łukasiewicz), qui fonctionne précisément parce que l'arité de h est connue. L'ensemble des règles devient : $\{ \vdash a, \vdash b, \vdash fV_1, \vdash V_1gV_2, \vdash V_1k, \vdash hV_1V_2 \}$, avec l'échelle de priorités $f(20) > g(15) > k(12) > h(5)$.

Sémantique de la basse priorité pour un préfixe. h étant le moins liant, il se place près de la racine de l'arbre et ses deux arguments **absorbent** tous les opérateurs plus prioritaires qu'ils rencontrent. Chaque argument est analysé jusqu'à ce que plus rien de priorité > 5 ne puisse l'étendre ; c'est cette saturation qui délimite la frontière entre V_1 et V_2 , sans virgule :

- hab = $h(a,b)$
- $hagbb$ = $h(g(a,b), b)$: l'analyse du premier argument démarre sur a , rencontre g ($15 > 5$) et l'absorbe ; le premier argument est donc agb , et le second b . Pour obtenir $h(a, g(b,b))$ il faut grouper : $ha(bgb)$.
- $hakb$ = $h(a, k(b))$ — l'exemple donné : k ($12 > 5$) est absorbé dans le second argument. Si k avait eu une priorité < 5 , il aurait au contraire coiffé tout le terme h .
- $hfabgbk$: premier argument $fa g b...$ attention, déroulons : $f(a)$, puis g ($15 > 5$) l'étend en $g(f(a), b)$; second argument $b...$ la chaîne exacte $hfagbbk$ donne $h(g(f(a), b), k(b))$ — 7 caractères contre 17 en syntaxe classique $h(g(f(a), b), k(b))$.

Règle de départage unifiée. « À priorité égale, l'opérateur de gauche gagne » généralise les règles précédentes : elle subsume l'associativité gauche de g (deux g de priorité égale $\rightarrow$ celui de gauche capture d'abord), et le cas unaire/unaire vu avec f et k . Par exemple deux h imbriqués : $hhabb$ = $h(h(a,b), b)$ — le h le plus à gauche... plus précisément, le h intérieur est rencontré en début de premier argument, s'y complète (hab), puis le h extérieur prend b comme second argument.

Une remarque de cohérence syntaxique. Le langage contenait déjà h ternaire en syntaxe classique $h(V_1, V_2, V_3)$. Si les deux coexistent, il y a un point délicat : la chaîne $h(agb)b$ (h binaire dont le premier argument est groupé) et la chaîne $h(a, b, a)$ (h ternaire classique) commencent identiquement par $h($, et seul un examen plus profond (présence de virgules) les distingue — l'automate perd sa propriété de choix de règle sur le symbole courant et doit recourir à du lookahead ou du retour arrière. Si en revanche la déclaration remplace h ternaire par h binaire, le système reste déterministe. Je signale l'alternative ; la suite me dira laquelle retenir.

Exemple de synthèse. Chaîne $hfakgbhbk$:

1. premier argument de h : $fa = f(a)$, puis $k(12 > 5) \rightarrow k(f(a))$, puis $g(15 > 5) \rightarrow g(k(f(a)), b)$;
2. second argument : le h interne, qui se complète avec b et $ak = k(a)$... non, attention : a puis $k \rightarrow$ premier argument du h interne = b , second = $k(a)$;
3. résultat : $h(g(k(f(a)), b), h(b, k(a)))$, soit 10 caractères contre 27 en syntaxe classique.

J'attends la suite.

h est déjà déclaré comme opérateur ternaire. Si on déclare un nouvel opérateur de même nom, celui-ci masque alors l'ancien opérateur qui ne peut plus être désigné par h .

Parfait, la règle de **masquage** (shadowing) lève l'ambiguïté que je signalais : la déclaration h **prefixe 5** introduit un nouvel opérateur binaire nommé h , et l'ancien h ternaire, désormais masqué, devient inaccessible — la chaîne $h(V_1, V_2, V_3)$ n'est plus reconnue par l'automate.

L'ensemble des règles de production est donc exactement :

$$\{ \vdash a, \vdash b, \vdash fV_1, \vdash V_1gV_2, \vdash V_1k, \vdash hV_1V_2 \}$$

avec l'échelle $f(20) > g(15) > k(12) > h(5)$, et la règle de départage « à priorité égale, l'opérateur de gauche gagne ».

Quelques conséquences de ce masquage :

Sur le langage désignable. Les termes contenant l'ancien h ternaire existent toujours dans l'algèbre L (l'opérateur n'est pas supprimé de la signature), mais ils n'ont **plus de représentation syntaxique** : la syntaxe courante ne peut plus les écrire ni les reconnaître. Le masquage crée donc une distinction entre le langage algébrique L (les termes) et le sous-ensemble des termes *exprimables* dans la syntaxe courante. Un terme comme $h_3(a, b, a)$ — je le note ainsi pour désambiguïser — est devenu inexprimable, sauf à ce qu'une future déclaration re-expose l'opérateur sous un autre nom.

Sur le déterminisme de l'automate. Le conflit $h($ disparaît : toute occurrence de h est le h binaire préfixe, et une parenthèse ouvrante qui le suit est nécessairement un groupement autour de son premier argument. Chaque règle redevient sélectionnable sans ambiguïté : a , b , f , h et $($ déclenchent chacun un début de terme unique ; g et k s'attachent en position infixe/postfixe selon les priorités. L'analyse reste un passage unique de type Pratt/precedence climbing.

Sur la gestion des déclarations. Cela suggère que les déclarations forment une suite ordonnée où chaque nom pointe vers sa déclaration la plus récente — une table de symboles avec masquage, comme les portées lexicales des langages de programmation. Deux points que la suite précisera peut-être : (1) le masquage est-il définitif ou existe-t-il une notion de portée permettant de restaurer l'ancien h ? (2) deux déclarations successives de même nom et même arité fonctionnent-elles pareil — la nouvelle syntaxe remplaçant simplement l'ancienne pour le *même* opérateur, ou s'agit-il là aussi d'un opérateur *distinct* homonyme ? La formulation « masque l'ancien opérateur » penche pour un nouvel opérateur distinct dans tous les cas.

Mon exemple de synthèse précédent reste valide tel quel : $hfakgbhbk = h(g(k(f(a)), b), h(b, k(a)))$, où h désigne partout le binaire préfixe.

J'attends la suite.

Il s'agit d'un opérateur distinct homonyme, mais nous développerons ce procédé de masquage plus-tard

Compris : chaque déclaration crée un opérateur **distinct**, même à nom et arité identiques — le nom n'est qu'une étiquette syntaxique pointant vers la déclaration la plus récente, tandis que l'opérateur lui-même conserve son identité propre dans l'algèbre. L'ancien h ternaire existe donc toujours comme générateur de L , simplement il n'est plus nommable. Nous y reviendrons.

État courant du système, pour référence :

Signature algébrique : $\langle a, b, f(.), g(.,.), h_3(.,.,.), k(.), h(.,.) \rangle$ — où h_3 désigne l'opérateur ternaire masqué (inexprimable) et h le binaire courant.

Règles de l'automate : $\{\vdash a, \vdash b, \vdash fV_1, \vdash V_1gV_2, \vdash V_1k, \vdash hV_1V_2\}$

Déclarations syntaxiques :

nom	fixité	priorité	associativité
f	préfixe	20	—
g	infixe	15	+ (gauche)
k	postfixe	12	—
h	préfixe	5	—

Départage : priorité supérieure = liaison plus forte ; à égalité, unaire avant binaire ; à égalité entre unaires (et plus généralement), l'opérateur de gauche gagne. Les parenthèses restent disponibles comme groupement explicite.

Prêt pour la suite.

On supprime la règle "à égalité, unaire avant binaire" qui n'apporte pas grand chose, et dont la suppression apporte finalement une simplification.

Bien noté. Les règles de départage se réduisent donc à deux principes :

1. **Priorités différentes** → l'opérateur le plus prioritaire lie plus fort (plus profond dans l'arbre).
2. **Priorités égales** → l'opérateur de gauche gagne, quelle que soit la nature (unaire ou binaire, préfixe, infixe ou postfixe) des opérateurs en conflit.

La simplification est réelle : une seule règle uniforme couvre tous les cas d'égalité — l'associativité gauche de g avec lui-même, le conflit préfixe/postfixe, et tout mélange unaire/binaire. La règle supprimée n'apportait effectivement rien d'essentiel : dans notre système, les priorités sont toutes distinctes ($f\ 20 > g\ 15 > k\ 12 > h\ 5$), donc elle ne s'appliquait jamais ; et même en cas d'égalité voulue, « gauche gagne » tranche de manière déterministe et prévisible.

Vérifions la cohérence sur un cas d'école : si f et g avaient tous deux la priorité 15, la chaîne **fagb** se lirait — f est à gauche de g, f gagne, f capture a — $(fa)gb = g(f(a), b)$. Même résultat qu'avec l'ancienne règle « unaire d'abord », mais obtenu par un principe unique. Et symétriquement **agfb** : g est à gauche... mais f doit de toute façon capturer un opérande avant que g n'ait son second argument, donc $a\ g\ (fb) = g(a, f(b))$ — le conflit ne se pose réellement que lorsque deux opérateurs se disputent le même opérande, et dans ce cas « gauche gagne » suffit toujours à conclure.

Au passage, cette règle unique donne une lecture opérationnelle simple de l'analyse : on parcourt la chaîne de gauche à droite et, à chaque conflit de capture, l'opérateur déjà rencontré (à gauche) l'emporte sauf si le nouveau venu est strictement plus prioritaire. C'est exactement l'invariant d'un analyseur par precedence climbing.

Le tableau des déclarations reste inchangé, seule la note de départage est simplifiée. Prêt pour la suite.

Considérons le langage enrichi d'un opérateur binaire j supplémentaire. Considérons la déclaration à 3 composantes suivantes :

j postfixe 21

Cela change la syntaxe classique de l'opérateur binaire p en syntaxe postfixe (sans parenthèse), c'est à dire que l'expression $j(V_1, V_2)$ n'est plus autorisée mais l'expression $V_1 V_2 j$ est autorisée pour désigner le même terme. Puis la priorité syntaxique vaut 21. Par exemple $f V_1 V_2 j$ désigne le terme $f(V_1 V_2 j)$. A priorité égale, la priorité est accordée à l'opérateur de gauche.

Bien noté (je lis « j » là où l'énoncé écrit une fois « p », simple lapsus je suppose). La déclaration se lit :

j	postfixe	21
nom	fixité	priorité

C'est la **notation polonaise inverse** appliquée à un opérateur binaire : les deux arguments d'abord, l'opérateur ensuite, sans parenthèses ni virgule — le miroir exact du h préfixe. L'ensemble des règles devient :

$$\{ \vdash a, \vdash b, \vdash fV_1, \vdash V_1 gV_2, \vdash V_1 k, \vdash hV_1 V_2, \vdash V_1 V_2 j \}$$

avec l'échelle $j(21) > f(20) > g(15) > k(12) > h(5)$.

Effet de la priorité maximale. j est le plus liant de tous : dans un conflit de capture, il gagne toujours, et ses deux arguments sont donc les termes **minimaux** qui le précèdent immédiatement. C'est exactement ce qu'illustre l'exemple : dans $f a b j$, f voudrait capturer a , mais j ($21 > 20$) remporte la dispute ; a et b reviennent à j , et f s'applique au résultat : $f(j(a, b))$. De même $a g b c j = g(a, j(b, c))$: j arrache b à g . Pour les lectures inverses, il faut grouper : $(f a) b j = j(f(a), b)$ et $(a g b) c j = j(g(a, b), c)$.

Belle **symétrie avec h** : h , préfixe de priorité minimale (5), a des arguments *maximaux* qui absorbent tout ; j , postfixe de priorité maximale (21), a des arguments *minimaux*. Les deux extrêmes de l'échelle produisent les deux notations polonaises pures.

Conséquence sur l'automate — un vrai changement de nature. Jusqu'ici, chaque terme commençait par un symbole discriminant. Désormais, la **juxtaposition de deux termes** (ab , $f a b$, ...) est un début de chaîne légal, dont la validité dépend d'un j à venir : en lisant $ab \dots$, l'automate ne sait pas encore si la chaîne sera correcte ($ab j$ ✓) ou non (ab seul ✗). La reconnaissance requiert donc une pile d'opérandes en attente — précisément le mécanisme d'évaluation RPN : on empile les termes, et chaque j dépile deux opérandes pour empiler $j(t_1, t_2)$. L'analyse reste en un passage, mais l'acceptation n'est décidée qu'en vérifiant à la fin que la pile contient exactement un terme.

Exemples.

- $\text{abj} = j(a,b)$; $\text{abj cj} = j(j(a,b), c)$; $\text{abc jj} = j(a, j(b,c))$ — la RPN exprime les deux regroupements *sans aucune parenthèse*, l'ordre des j suffit.
- $\text{abjk} = k(j(a,b))$: k (12) s'applique au terme déjà formé.
- fabjk gb : $\text{fabj} = f(j(a,b))$, puis $k \rightarrow k(f(j(a,b)))$, puis $\text{gb} \rightarrow \mathbf{g(k(f(j(a,b)))), b}$ — 7 caractères contre 18 en syntaxe classique.
- habjbcjj : h (5, le plus faible) prend comme arguments tout ce que les autres construisent : premier argument $\text{abj} = j(a,b)$... puis $\text{bcj} = j(b,c)$, puis le j final les combine ? Déroulons en RPN pur : $a, b, j \rightarrow j(a,b)$; $b, c, j \rightarrow j(b,c)$; $j \rightarrow j(j(a,b), j(b,c))$; et h attend toujours ses arguments — la chaîne complète habjbcjjb donnerait $h(j(j(a,b), j(b,c)), b)$. L'imbrication h/j se lit donc : j construit des blocs denses, h les assemble au sommet.

J'attends la suite.

La déclaration suivante :

g classique

redonne la syntaxe classique à l'opérateur g , la seule écriture autorisé est alors de la forme $g(V_1, V_2)$

Bien noté. La déclaration à 2 composantes :

```
g    classique
|    |
nom  fixité (retour à la syntaxe fonctionnelle)
```

restaure pour g la syntaxe préfixe parenthésée $g(V_1, V_2)$, qui redevient la **seule** écriture autorisée — la forme infixe $V_1 g V_2$ n'est plus reconnue. Point notable : la syntaxe classique n'a besoin ni de priorité ni d'associativité, car ses parenthèses et virgules obligatoires délimitent totalement les arguments ; un opérateur classique est hors du jeu des priorités, il forme toujours un bloc atomique aux yeux des autres opérateurs. C'est cohérent avec le fait que la déclaration n'a que 2 composantes.

(Et ceci confirme au passage le principe du masquage : ce g « classique » est vraisemblablement un troisième opérateur homonyme distinct, masquant le g infixe, lui-même masquant... non, le g infixe était le premier déclaré sur le g d'origine — disons simplement que la déclaration crée un nouvel opérateur binaire g en syntaxe classique, et que le g infixe de priorité 15 n'est plus nommable.)

État courant du système :

Règles de l'automate : $\{\vdash a, \vdash b, \vdash fV_1, \vdash g(V_1, V_2), \vdash V_1 k, \vdash hV_1 V_2, \vdash V_1 V_2 j\}$

nom	fixité	priorité	assoc.
j	postfixe	21	—
f	préfixe	20	—
k	postfixe	12	—
h	préfixe	5	—
g	classique	—	—

Départage inchangé : priorité supérieure lie plus fort ; à égalité, gauche gagne.

Conséquences concrètes :

- L'échelle des priorités se resserre : $j(21) > f(20) > k(12) > h(5)$, et g s'insère n'importe où comme un atome. $fg(a, b) = f(g(a, b))$ — ce qui exigeait un groupement $f(agb)$ redevient direct. Inversement, l'associativité implicite disparaît : l'ancien $agbga$ s'écrit obligatoirement $g(g(a, b), a)$.
- Les chaînes mêlant les styles restent non ambiguës : $g(fab\ j, ak)k = k(g(f(j(a, b)), k(a)))$ — à l'intérieur des parenthèses de g , chaque argument est analysé avec les priorités usuelles.
- Côté densité, on régresse localement ($g(a, b)$: 6 caractères contre agb : 3), mais on y gagne parfois en lisibilité et en économie de parenthèses de groupement lorsque g devait être argument d'opérateurs plus prioritaires.

Une question que je garde en réserve : la virgule de la syntaxe classique jouant le rôle de délimiteur fort, une chaîne comme $g(ab\ j, b)$... s'analyse sans peine ; en revanche $g(a, b\ k\ g(a, b))$... aussi. La syntaxe classique cohabite proprement avec les autres fixités puisque $(, ,$ et $)$ bornent chaque argument.

Prêt pour la suite.

Non, le masquage d'opérateur n'a lieu que lors de la définition d'un nouvel opérateur (dont je n'ai pas formaliser la déclaration), et non lors du changement de sa syntaxe.

Merci pour la correction — je distinguais mal deux opérations de nature différente :

1. **Définition d'un opérateur** (déclaration non encore formalisée) : crée un nouvel opérateur ; s'il porte un nom déjà utilisé, il **masque** l'ancien. C'est ce qui s'est produit pour h : la phrase « on enrichit le langage d'un opérateur binaire h supplémentaire » définissait un nouvel opérateur, et c'est cette définition — non la déclaration de syntaxe qui l'accompagnait — qui a masqué le h ternaire.
2. **Déclaration de syntaxe** (infixe/préfixe/postfixe/classique + priorité + associativité) : modifie la représentation textuelle d'un opérateur **existant**, sans toucher à son identité. L'opérateur reste le même élément de la signature ; seule sa règle de production dans l'automate change.

Le g actuel est donc bien **le g d'origine**, celui-là même qui fut successivement en syntaxe classique (initiale), puis infixe 15 +, puis de nouveau classique. Un seul opérateur, trois états syntaxiques successifs — et tous les termes contenant g restent identiques dans l'algèbre à travers ces changements ; seule leur écriture varie. La syntaxe est une propriété mutable de l'opérateur, pas un constituant de son identité.

État courant rectifié :

Signature : $\langle a, b, f(.), g(.,.), h_3(.,.,.), k(.), h(.,.), j(.,.) \rangle$ — sept opérateurs/constantes dont un masqué (h_3 , innommable car son nom est capté par le h binaire défini ensuite).

Syntaxes courantes :

opérateur	fixité	priorité	assoc.
$j(.,.)$	postfixe	21	—
$f(.)$	préfixe	20	—
$k(.)$	postfixe	12	—
$h(.,.)$	préfixe	5	—
$g(.,.)$	classique	—	—

Règles de l'automate : $\{ \vdash a, \vdash b, \vdash fV_1, \vdash g(V_1, V_2), \vdash V_1k, \vdash hV_1V_2, \vdash V_1V_2j \}$

Une conséquence intéressante de cette distinction : puisque la syntaxe est mutable et l'identité stable, une même donnée stockée sous deux syntaxes successives (par exemple une archive écrite du temps où g était infixe) désigne les mêmes termes — mais sa relecture exige de connaître la table des déclarations en vigueur au moment de l'écriture. La table de syntaxe fait donc partie des métadonnées du stockage. J'attends la suite — la formalisation de la déclaration de définition, peut-être.

L'ajout d'un opérateur ou d'un élément dans le langage L correspond à une extension élémentaire. On modifie le langage L par déclarations successives. Exemple, considérons les déclarations suivantes :

$L := \langle a, b, f(.), g(.,.) \rangle$

$L := L[c]$

à cette étape nous avons $L = \langle a, b, c, f(.), g(.,.) \rangle$

$L := L[k(.,.), q(.), d]$

à cette étape nous avons $L = \langle a, b, c, d, f(.), q(.), g(.,.), k(.,.) \rangle$

Bien noté. Formalisons le mécanisme d'**extension élémentaire** :

Syntaxe de la déclaration. $L := L[x_1, \dots, x_n]$ où chaque x_i est soit un élément générateur (nom seul : **c**, **d**), soit un opérateur générateur (nom suffixé de son arité : **k(.,.)**, **q(.)**).

L'extension ajoute ces générateurs à la signature ; plusieurs peuvent être ajoutés en une seule déclaration, comme le montre $L[k(.,.), q(.), d]$.

Sémantique. Chaque extension redéfinit L comme le langage engendré par l'ancienne signature augmentée des nouveaux générateurs. C'est une extension *conservatrice* : tous les termes de l'ancien L sont des termes du nouveau (l'ancien langage se plonge dans le nouveau), et les nouveaux termes sont ceux qui font intervenir au moins un nouveau générateur. Après les deux extensions de l'exemple :

$L = \langle a, b, c, d, f(.), q(.), g(.,.), k(.,.) \rangle$

Je note dans cette présentation finale un ordre canonique : les constantes d'abord, puis les opérateurs par arité croissante — la présentation de la signature est un *ensemble* de générateurs, indifférent à l'ordre et à l'historique des déclarations.

Trois observations :

1. **C'est ici que vit le masquage.** L'extension élémentaire est la « définition d'opérateur » annoncée : si une extension introduit un générateur portant un nom déjà présent, le nouveau masque l'ancien. L'épisode du h binaire se reformule donc : $L := L[h(.,.)]$, qui a masqué h_3 . L'ancien opérateur demeure dans la signature (les termes qui le contiennent existent toujours) mais son nom pointe désormais vers le nouveau. La signature est donc en toute rigueur une *suite* de déclarations, dont la présentation $\langle \dots \rangle$ ne montre que les générateurs actuellement nommables.
2. **L devient une variable métalinguistique.** La notation $L := \dots$ est une affectation : L désigne l'état courant d'un langage en évolution, chaque déclaration produisant un nouvel état à partir du précédent. Le système complet d'une session de stockage est ainsi une suite de déclarations de deux sortes (jusqu'ici) : extensions élémentaires $L := L[\dots]$ et déclarations de syntaxe (nom fixé, priorité associée). L'interprétation d'une chaîne dépend de la position dans cette suite.
3. **Syntaxe par défaut.** Les nouveaux opérateurs arrivent vraisemblablement en syntaxe classique — $q(V_1), k(V_1, V_2)$ — jusqu'à déclaration de syntaxe contraire ; et les constantes n'ont pas de question de syntaxe (atomes). Je suppose aussi que rien n'interdit d'étendre par des arités nouvelles, par exemple $L[m(.,.,.,.)]$ pour un quaternaire.

La reconstruction de notre langage de travail s'écrit donc :

```
L := <a, b, f(.), g(.,.), h(.,.,.)>
L := L[k(.)]
L := L[h(.,.)]      — masque le h ternaire
L := L[j(.,.)]
```

avec, entrelacées, les déclarations de syntaxe f préfixe 20 ; g infixe 15 + ; k postfixe 12 ; h préfixe 5 ; j postfixe 21 ; g classique.

J'attends la suite — la restriction (suppression de générateurs), ou la levée du masquage, peut-être ?

On définit des opérateurs multi-aires, déclaré avec le suffixe " $(\dots)$ " qui indique un nombre variable d'arguments qui peut être nul. Exemple :

$L = \langle a, f(\dots) \rangle$

L comprend les termes suivants : $a, f(), f(a), f(a,a), f(a,a,a), f(f(a),a,f(a),a)$

Bien noté. Un opérateur **multi-aire** (ou variadique) accepte un nombre quelconque d'arguments, zéro compris :

$L = \langle a, f(\dots) \rangle$

Définition inductive. L est le plus petit ensemble tel que :

1. $a \in L$;
2. pour tout $n \geq 0$, si $t_1, \dots, t_n \in L$, alors $f(t_1, \dots, t_n) \in L$.

La règle 2 condense en une seule clause une infinité de règles d'arité fixe. Vos exemples s'y retrouvent : $f()$ ($n = 0$), $f(a)$ ($n = 1$), $f(a,a)$, $f(a,a,a)$, $f(f(a),a,f(a),a)$ ($n = 4$, avec imbrication).

Observations structurelles :

1. **$f()$ est un quasi-atome.** Le cas $n = 0$ fait de $f()$ un terme sans sous-terme — fonctionnellement une constante, mais syntaxiquement distincte de a (les parenthèses vides sont obligatoires, sinon f seul serait ambigu ou dénoterait autre chose). D'ailleurs le langage $\langle f(\dots) \rangle$ *sans aucune constante* serait non vide, ce qui est impossible avec des arités fixes ≥ 1 : $f()$, $f(f())$, $f(f(),f())$... — les arbres purs, sans étiquettes de feuilles autres que $f()$.
2. **Les arbres de L sont d'arité non bornée :** chaque nœud f a un nombre quelconque de fils ordonnés. C'est la structure des listes imbriquées hétérogènes — on reconnaît les S-expressions de Lisp, où $f(t_1, \dots, t_n)$ s'écrirait $(f\ t_1 \dots t_n)$.
3. **Un seul opérateur, une infinité d'« arités » :** contrairement au masquage où h ternaire et h binaire étaient deux opérateurs distincts, ici f est *un* opérateur unique dont les occurrences $f(a)$ et $f(a,a)$ coexistent dans un même langage, voire un même terme : $f(f(), f(a,a), a)$.

Côté automate, la règle de production s'écrit avec une itération : $\vdash f(V_1, \dots, V_n)$ pour $n \geq 0$ — opérationnellement : lire **f(**, puis une suite de termes séparés par des virgules (éventuellement vide), puis **)**. La syntaxe classique absorbe le multi-aire sans peine, ses délimiteurs faisant tout le travail. En revanche, les syntaxes sans parenthèses vues précédemment se marient mal avec l'arité variable : un f préfixe sans délimiteurs ne saurait pas où arrêter ses arguments ($fab = f(a,b)$ ou $f(a)b$?) — l'arité fixe était précisément ce qui rendait la notation polonaise décodable. Je m'attends donc à ce que les multi-aires restent en syntaxe classique, ou reçoivent une syntaxe à délimiteurs dédiée (séparateur infixé itéré, par exemple).

Petite anticipation algébrique : les multi-aires sont le cadre naturel des opérations associatives « aplaties » — écrire $g(a,b,c)$ plutôt que $g(g(a,b),c)$ — et des collections (listes, tuples, ensembles ordonnés). J'attends la suite.

L'opérateur multi-aire f peut être mis en syntaxe préfixe mais en gardant les parenthèses. Exemple :

$f(abaa)$

qui correspond à $f(a,b,a,a)$.

L'opérateur multi-aire f peut être mis en syntaxe infixe mais en se répétant. Exemple :

$afbafafa$

qui correspond à $f(a,b,a,a)$.

Dans un premier temps on autorise pas la syntaxe postfixe pour les raisons que vous invoquez

Bien noté. Deux syntaxes pour les multi-aires, chacune avec ses subtilités :

1. Préfixe multi-aire : parenthèses conservées, virgules supprimées. $f(abaa) = f(a,b,a,a)$.

Les parenthèses restent nécessaires pour borner la *liste* d'arguments (l'arité étant variable), mais les virgules deviennent superflues à condition que **la frontière entre arguments soit décodable** — ce qui est le cas ici : chaque terme est auto-délimité par sa structure. En lisant $abaa$, l'automate reconnaît un terme complet (a), puis recommence : la segmentation est déterministe car aucun terme n'est préfixe propre d'un autre terme là où l'ambiguïté compterait. Avec des imbrications : $f(f(ab)af(a)) = f(f(a,b), a, f(a))$ — chaque argument se termine sans équivoque. Le cas $n = 0$ reste exprimable : $f()$. Point de vigilance pour l'avenir : cette décodabilité repose sur les propriétés du lexique ; si des noms de plusieurs caractères apparaissent un jour (ab constante ?), il faudra des séparateurs ou une tokenisation prudente.

2. Infixe multi-aire : l'opérateur se répète en séparateur. $afbafafa = f(a,b,a,a)$. Ici f joue le rôle d'un séparateur itéré, et la lecture est **plate** : la suite $t_1 f t_2 f \dots f t_n$ désigne $f(t_1, \dots, t_n)$ directement — et non un emboîtement binaire ($f(f(f(a,b),a),a)$). C'est exactement la sémantique « aplatie » que j'anticipais pour les opérations associatives : la question de l'associativité disparaît puisque tous les opérands sont frères. La déclaration infixe d'un multi-aire n'aurait donc besoin que d'une priorité, sans composante $+/-$: **f infixe p** (3 composantes), l'associativité étant sans objet.

Limites expressives de l'infixe, symétriques de sa concision :

- **$n = 0$ et $n = 1$ inexprimables** : $f()$ et $f(a)$ n'ont pas d'écriture infixe (a seul dénote la constante a , pas $f(a)$). L'infixe ne couvre que $n \geq 2$. Si le langage doit distinguer $f(a)$ de a , l'infixe est une syntaxe *partielle* — ou bien on l'interdit, ou bien ces cas gardent un recours (mais lequel, la syntaxe déclarée étant exclusive ? point à trancher).
- **L'imbrication directe de f dans f exige un groupement** : $f(a, f(b,a), a)$ s'écrit $af(bfa)fa$ — les parenthèses de groupement réapparaissent, et l'on note que $afbafafa$ sans groupe désigne *toujours* la version plate à 4 arguments.

3. **Postfixe interdite — et je précise l'argument.** En postfixe multi-aire, **abaaf** ne permettrait pas de savoir combien d'opérandes f consomme : l'arité n'étant pas fixe, il faudrait l'annoter (**abaaf₄** — c'est la solution de certains langages à pile) ou délimiter le début de la liste, ce qui revient à réintroduire des parenthèses et perd l'intérêt. L'information d'arité, connue *a priori* pour h et j , doit ici être portée par la syntaxe elle-même ; le préfixe la porte par ses parenthèses, l'infixe par le comptage des séparateurs, le postfixe nu ne peut la porter nulle part.

Bilan des déclarations possibles pour un multi-aire, à ce stade : **f classique** ($f(a,b,a)$), **f préfixe** ($f(aba)$ — sans priorité nécessaire ? les parenthèses le rendent atomique comme le classique), **f infixe p** ($afbfa$, priorité p , pas d'associativité). J'attends la suite.

On supprime la syntaxe infixe pour le multi-aire

Bien noté. La syntaxe infixe est retirée pour les multi-aires — décision cohérente : ses lacunes expressives ($n = 0$ et $n = 1$ inexprimables) en faisaient une syntaxe partielle, incapable de désigner tous les termes de l'opérateur, ce qui est rédhibitoire pour un format de *stockage* où chaque terme doit posséder une écriture.

Restent donc pour un opérateur multi-aire deux syntaxes totales :

fixité	écriture de $f(a,b,a,a)$	$n = 0$	$n = 1$
classique	$f(a,b,a,a)$	$f()$	$f(a)$
préfixe	$f(abaa)$	$f()$	$f(a)$

Les deux couvrent toutes les arités, ne diffèrent que par les virgules, et rendent l'opérateur syntaxiquement atomique (borné par ses parenthèses), donc hors du système des priorités — la déclaration **f préfixe** d'un multi-aire n'a vraisemblablement besoin d'aucune composante numérique.

Récapitulatif général du système de syntaxes à ce stade :

Opérateurs d'arité fixe :

- classique — parenthèses et virgules, hors priorités ;
- préfixe p — sans parenthèses (unaires : fV_1 ; n -aires : notation polonaise hV_1V_2) ;
- infixe $p \pm$ — binaires, avec associativité ;
- postfixe p — sans parenthèses (unaires : V_1k ; n -aires : notation polonaise inverse V_1V_2j).

Opérateurs multi-aires :

- classique — $f(a,b,c)$;
- préfixe — $f(abc)$, virgules omises ;
- infixes et postfixes : non autorisées.

Et les règles transverses : priorité supérieure lie plus fort ; à égalité, l'opérateur de gauche gagne ; parenthèses de groupement toujours disponibles ; les déclarations de syntaxe modifient l'écriture d'un opérateur sans changer son identité ; les extensions $L := L[\dots]$ ajoutent des générateurs, avec masquage en cas d'homonymie.

J'attends la suite.

Jusqu'à présent on a défini des langages où les opérateurs sont des applications internes au langage. Considérons deux langages L et H définis par des éléments générateurs et des opérateurs internes et des opérateurs externes.

Un opérateur générateur unaire f du langage L prenant comme argument un terme d'un langage H , se notera $f(H)$ au lieu de $f(.)$

Un opérateur générateur binaire g du langage L prenant comme argument un terme d'un langage H , et un terme du langage L se notera $g(H,L)$ au lieu de

[Afficher plus](#)

Bien noté. Nous passons des algèbres homogènes aux **algèbres multi-sortes** (ou hétérogènes) : les arguments des opérateurs sont désormais typés par le langage auquel ils appartiennent, et la notation d'arité anonyme $(.)$ est remplacée par le nom de la sorte attendue.

Signatures de l'exemple :

$L = \langle a, b, f(H), g(H,L) \rangle$ — deux constantes, un unaire prenant un terme de H , un binaire prenant un terme de H puis un terme de L .

$H = \langle e, j(H,H), k(H,L,L) \rangle$ — une constante, un binaire interne à H , un ternaire prenant un H puis deux L .

Définition inductive — nécessairement mutuelle, car L et H se référencent l'un l'autre : (L, H) est le plus petit couple d'ensembles tel que :

1. $a, b \in L ; e \in H$;
2. si $t \in H$ alors $f(t) \in L$;
3. si $t_1 \in H$ et $t_2 \in L$ alors $g(t_1, t_2) \in L$;
4. si $t_1, t_2 \in H$ alors $j(t_1, t_2) \in H$;
5. si $t_1 \in H$ et $t_2, t_3 \in L$ alors $k(t_1, t_2, t_3) \in H$.

Exemples de termes : $f(e) \in L$; $g(e, f(e)) \in L$; $k(j(e, e), a, f(e)) \in H$; $g(k(e, a, b), g(e, a)) \in L$. Et notons que l'imbrication alterne les sortes en profondeur : un terme de L peut contenir des sous-termes de H qui contiennent des sous-termes de L, etc.

Conséquences sur les automates. Les deux automates sont **mutuellement rékursifs** : dans la règle $\vdash f(H)$ de l'automate de L, la variable de sorte H déclenche un appel récursif de l'automate de H — et réciproquement pour les positions L dans les règles de H. Chaque variable porte désormais sa sorte, qui désigne l'automate à appeler. La reconnaissance de $g(j(e, k(e, a, b)), f(e))$ par l'automate de L :

$\vdash g(H, L)$	automate L, $V_1 \leftarrow "j(e, k(e, a, b))" : H, \quad V_2 \leftarrow "f(e)"$
: L	
$\vdash j(H, H)$	automate H
$\vdash e$	automate H : base
$\vdash k(H, L, L)$	automate H
$\vdash e$	automate H : base
$\vdash a$	automate L : base
$\vdash b$	automate L : base
$\vdash f(H)$	automate L
$\vdash e$	automate H : base

Trois observations :

1. **Le typage devient une condition de bonne formation :** $f(a)$ est syntaxiquement rejeté ($a \notin H$), alors qu'en monosorte il eût été valide. L'automate vérifie la grammaire et les sortes en un même mouvement — les erreurs de type sont des erreurs de syntaxe.
2. **« Interne/externe » est relatif à chaque opérateur et à chaque position :** j est interne à H ; k est mixte ; f est un opérateur de L dont l'unique argument est externe. Le cas homogène antérieur est le cas particulier où toutes les positions portent la sorte du langage hôte.
3. **Les variables muettes se typent :** il faudra distinguer les variables de sorte L et celles de sorte H (les V_i des règles ci-dessus sont implicitement typées par leur position). Et pour les déclarations de syntaxe, je m'attends à ce qu'elles s'appliquent par langage — avec une question intéressante : si f devient préfixe sans parenthèses (fV_1), l'analyse de son argument bascule vers l'automate de H et ses propres tables de priorités ; les priorités sont-elles par langage ou globales ? La suite le dira.

Prêt pour la suite.

Les variables de sorte L se note $L_1, L_2, L_3, \dots$ où simplement L lorsqu'il n'y a pas d'autre occurrence dans le terme. Les variables de sorte H se note $H_1, H_2, H_3, \dots$ où simplement H lorsqu'il n'y a pas d'autre occurrence dans le terme.

La syntaxe (qui comprend les priorités) est associée au langage. Les priorités sont donc par langage. Les arguments sont toujours écrits dans le langage attendu par l'opérateur

Les opérateurs générateurs d'un langage (un élément générateur est un

élément d'un langage qui ne peut pas être décomposé en une suite de

Afficher plus

Bien noté. Quatre principes structurants :

1. Variables typées par convention de nommage. L_i pour la sorte L, H_i pour la sorte H, l'indice étant omissible en cas d'occurrence unique dans le terme. Les règles des automates s'écrivent donc naturellement : $\{ \vdash a, \vdash b, \vdash f(H), \vdash g(H, L) \}$ et $\{ \vdash e, \vdash j(H_1, H_2), \vdash k(H, L_1, L_2) \}$ — le nom de la variable est sa sorte, et la contrainte des variables muettes s'affine : deux occurrences de L_i dans un motif désignent le même terme de L, tandis que L_1 et L_2 sont des termes de L indépendants.

2. La syntaxe est une propriété du langage. Chaque langage possède sa table de déclarations (fixités, priorités, associativités) ; les priorités ne se comparent qu'*au sein d'un même langage*. Et la règle « les arguments sont toujours écrits dans le langage attendu » donne la discipline d'analyse : lorsque l'automate de L analyse $f(H)$... disons en syntaxe préfixe fH , la lecture de l'argument bascule intégralement dans la syntaxe de H — ses propres opérateurs, ses propres priorités. Les frontières de sortes sont des **frontières syntaxiques étanches** : un conflit de priorité entre un opérateur de L et un opérateur de H ne peut pas se poser, car ils ne sont jamais en compétition pour un même opérande — chaque position d'argument fixe la sorte, donc le décodeur compétent. La chaîne se stratifie en zones, chacune régie par la grammaire de son langage.

3. Les générateurs comme objets de la classe-langage. L'analogie objet est éclairante : un langage est une classe, ses opérateurs générateurs sont ses membres (les constantes étant le cas d'arité nulle — unification élégante : plus deux catégories « éléments » et « opérateurs », mais une seule, graduée par l'arité). L'espace de noms est par conséquent la classe elle-même.

4. Nommage qualifié. f de L et f de H sont deux opérateurs distincts, sans masquage entre eux — le masquage ne joue qu'à l'intérieur d'un langage. La désignation complète d'un opérateur est un couple (langage, nom), que je noterais volontiers $L.f$ et $H.f$ en attendant la notation officielle. Remarquons que dans les chaînes, cette qualification est inutile : la position d'un symbole détermine sa sorte, donc f dans une zone L dénote $L.f$ sans ambiguïté — la qualification ne sert qu'au niveau méta (déclarations, discussion des signatures). D'ailleurs les déclarations de syntaxe devront être qualifiées : « $L.f$ préfixe 20 » ou une forme équivalente, sans quoi « f préfixe 20 » serait ambigu si les deux langages ont un f .

Une conséquence intéressante du point 2 : les ambiguïtés lexicales inter-langages sont neutralisées. Si L et H possèdent chacun un opérateur nommé f avec des syntaxes différentes (L.f préfixe 20, H.f postfixe 3), la même lettre f se lit différemment selon la zone — licite, puisque le lecteur sait toujours dans quel langage il se trouve. La seule exigence globale est que *chaque* langage soit intérieurement non ambigu.

Et une question en réserve : les extensions élémentaires se qualifient-elles de même — $L := L[m(H,L)]$ pouvant importer la sorte H dans la signature de L, ce qui crée une *dépendance* entre langages ? Le graphe des dépendances (ici $L \rightleftharpoons H$, mutuellement récursifs) devient une donnée du système. J'attends la suite.

Variables typées par convention de nommage. H1,H2,H3 pour la sorte H. Lorsque l'indice est omis, les variables H désignent toujours des variables distinctes sans occurrence.s Ainsi $j(H,H)$ désigne le terme $j(H1,H2)$ mais ne désigne pas le terme $j(H1,H1)$ qui constitue la diagonale de j.

Bien noté — la convention est plus fine que ce que j'avais retenu, corrigeons :

Convention d'omission de l'indice. Une variable sans indice désigne, à *chaque occurrence*, une variable fraîche distincte de toutes les autres. Ainsi :

- $j(H,H) \equiv j(H_1,H_2)$ — deux inconnues **indépendantes** ; c'est le motif générique de j, qui couvre tous les termes de tête j ;
- $j(H_1,H_1)$ — la **diagonale** de j, motif strictement plus contraint : les deux arguments sont le même terme ; il ne peut pas s'écrire avec des variables sans indice.

L'omission d'indice n'est donc pas réservée au cas d'occurrence unique dans le terme (comme je l'avais formulé) : on peut écrire plusieurs H nus dans un même motif, chacun étant implicitement une variable nouvelle. La règle de lecture est : **H nu = variable anonyme jamais coréférente** ; seuls les indices explicites créent la coréférence. C'est exactement la sémantique du joker  de Prolog ou des langages à filtrage — le partage doit être demandé, jamais accidentel.

Cette convention a une conséquence heureuse pour les règles des automates : elles s'écrivent sans indices, $\{ \vdash e, \vdash j(H,H), \vdash k(H,L,L) \}$, ce qui est correct puisqu'un reconnaisseur n'impose aucune contrainte d'égalité entre sous-termes — les positions d'arguments sont indépendantes. Les règles de production sont précisément les motifs *les plus généraux* de chaque opérateur, et la notation sans indice les capture exactement. Les indices ne deviennent nécessaires que pour exprimer des motifs contraints : diagonales $j(H_1,H_1)$, et plus tard sans doute axiomes équationnels ou règles de réécriture, où la coréférence entre membre gauche et membre droit est essentielle — par exemple une règle comme $g(H_1, f(H_1)) \rightarrow \dots$ exige l'indice pour lier les deux occurrences.

Petit inventaire des motifs de j selon la convention :

- $j(H,H) = j(H_1,H_2)$: motif général, 2 variables libres ;
- $j(H_1,H_1)$: diagonale, 1 variable ;
- $j(e,H)$: premier argument fixé, 1 variable ;
- $j(H_1, j(H_1,H)) = j(H_1, j(H_1,H_2))$: coréférence partielle mixte.

Et la hiérarchie d'instanciation : tout terme filtré par $j(H_1,H_1)$ l'est par $j(H,H)$, non réciproquement — la diagonale est un sous-motif du motif général.

J'attends la suite.

Un opérateur peut contenir la définition d'une application. Considérons le langage $L = \langle a, b, f(.), g(.,.) \rangle$ que l'on peut noter $L = \langle a, b, f(L), g(L,L) \rangle$ ou bien encore $L = \langle a, b, f(L_1), g(L_1,L_2) \rangle$ Considérons la déclaration de g suivante :

$$g := ((L_1, L_2) \mapsto g(g(a, L_2), L_1))$$

Dès lors g contient la définition d'une application, et peut donc faire l'objet d'une évaluation.

De même pour un élément. Considérons la déclaration de b suivante :

$$b := (\mapsto f(a))$$

Dès lors b contient la définition d'une procédure de production, qui sera activé si b fait l'objet d'une évaluation

Bien noté. Une troisième composante s'attache désormais aux opérateurs : après l'**identité** (couple langage-nom, signature de sortes) et la **syntaxe** (fixité, priorité), voici la **définition** — une application dont l'activation s'appelle évaluation.

Anatomie de la déclaration.

$$g := ((L_1, L_2) \mapsto g(g(a, L_2), L_1))$$

Le membre gauche du $\mapsto$ lie les paramètres formels (variables typées, avec indices explicites puisque la coréférence entre les deux membres est essentielle — voici l'usage annoncé des indices) ; le membre droit est un terme du langage construit sur ces variables. La définition fait de g une **règle de réécriture** : évaluer $g(t_1, t_2)$ consiste à substituer, produisant $g(g(a, t_2), t_1)$ — les arguments sont permutés et un $g(a, \cdot)$ s'intercale.

$$b := (\mapsto f(a))$$

Cas d'arité nulle, cohérent avec l'unification constante = opérateur 0-aire : le membre gauche est le tuple vide de paramètres, et évaluer b produit $f(a)$. Vous nommez cela une *procédure de production* — b n'est plus un simple atome mais un terme *actif*, qui se déploie sur demande.

Observations :

1. **Deux populations d'opérateurs.** Ceux sans définition (a, f) sont des **constructeurs** purs : ils ne s'évaluent pas, ils *sont* leur propre valeur — la donnée. Ceux avec définition (g, b) sont des opérateurs **définis**, réductibles. Le langage mêle librement les deux, et un terme comme $g(b, f(b))$ contient trois foyers d'évaluation possibles.
2. **L'évaluation est paresseuse par nature** — « si b fait l'objet d'une évaluation ». La définition ne se déclenche pas à la construction du terme : b reste b tant qu'on ne l'évalue pas. Le terme est la forme stockée, l'évaluation une opération séparée. Distinction cruciale ici, car :
3. **La définition de g est non terminante en évaluation répétée.** Son membre droit contient g lui-même : $g(t_1, t_2) \rightarrow g(g(a, t_2), t_1) \rightarrow g(g(a, t_1), g(a, t_2)) \rightarrow \dots$ indéfiniment. De même $b \rightarrow f(a)$ termine, mais si b figurait dans son propre membre droit ($b := (\mapsto f(b))$), on aurait une production infinie — un flux. Le système devra donc préciser sa **stratégie d'évaluation** : un seul pas de réécriture ? réduction jusqu'à forme normale (quand elle existe) ? évaluation externe/interne, paresseuse/stricte ? La déclaration installe le mécanisme ; la stratégie reste à fixer.
4. **Questions en réserve.** (a) La définition doit-elle être *linéaire à gauche* et *générale* — le membre gauche est toujours le motif $(L_1, \dots, L_n)$ le plus général, ou admettra-t-on des définitions par cas sur des motifs $(g(a, L) \mapsto \dots, g(f(L_1), L_2) \mapsto \dots)$? (b) Le membre droit peut-il invoquer des opérateurs d'autres langages, dans le respect des sortes ? (c) Une redéclaration $g := \dots$ remplace-t-elle la définition (comme la syntaxe, propriété mutable) sans changer l'identité de g ? Je le présume, par cohérence avec le principe « identité stable, propriétés mutables ».

Le système prend sa physionomie complète : un opérateur = une identité + une signature de sortes + une syntaxe + éventuellement une définition. Les trois premières régissent le stockage et la reconnaissance, la quatrième introduit le *calcul*. J'attends la suite — la stratégie d'évaluation, sans doute.

L'opérateur g peut contenir la définition d'une fonction (qui commence par une unification déterminant le domaine de définition de la fonction). Par exemple considérons la déclaration de g suivante :

$$g := (g(L_1, f(L_1)) \mapsto g(a, L_1))$$

Dés lors le terme $g(b, f(b))$, s'il fait l'objet d'une évaluation, est remplacé par $g(a, b)$. Tandis que le terme $g(a, b)$, s'il est évalué, reste inchangé car l'unification $g(a, b) * g(L_1, f(L_1))$ échoue.

Bien noté. La définition s'enrichit : le membre gauche n'est plus nécessairement le motif général $(L_1, \dots, L_n)$ mais peut être un **motif contraint** — la définition devient une fonction *partielle*, dont le domaine est déterminé par filtrage.

$$g := (g(L_1, f(L_1)) \mapsto g(a, L_1))$$

Mécanique de l'évaluation. Évaluer un terme t de tête g procède en deux temps :

1. **Unification** $t * g(L_1, f(L_1))$ — je note $*$ l'opération, suivant votre notation. On cherche une substitution σ telle que $\sigma(g(L_1, f(L_1))) = t$. C'est ici du *filtrage* (unification à sens unique : t est clos, seul le motif contient des variables).
2. **Production** : si σ existe, t est remplacé par $\sigma(g(a, L_1))$; si l'unification **échoue**, t reste inchangé — l'évaluation est l'identité hors du domaine.

Sur les exemples :

- $g(b, f(b)) * g(L_1, f(L_1))$ réussit avec $\sigma = \{L_1 \mapsto b\} \rightarrow \text{résultat } \sigma(g(a, L_1)) = \mathbf{g(a,b)} \checkmark$
- $g(a, b) * g(L_1, f(L_1))$ échoue : b ne filtre pas $f(L_1)$ (têtes $b \neq f$) $\rightarrow \mathbf{g(a,b)}$ inchangé $\checkmark$
- Et notons : $g(f(a), f(f(a)))$ réussit avec $\sigma = \{L_1 \mapsto f(a)\} \rightarrow g(a, f(a))$; tandis que $g(a, f(b))$ échoue *par la coréférence* : le premier argument impose $L_1 = a$, le second exige $f(L_1) = f(b)$ donc $L_1 = b$ — contrainte insatisfiable. La diagonale implicite du motif (L_1 apparaît deux fois) fait partie du domaine de définition : voici les indices explicites pleinement à l'œuvre.

Trois observations :

1. **Le domaine de définition est un sous-ensemble filtrable de L.** $\text{dom}(g) = \{\text{termes de la forme } g(t, f(t))\}$. Hors domaine, votre choix sémantique est notable : non pas *erreur*, mais *point fixe* — le terme est sa propre valeur. Tout terme est donc évaluable, et les constructeurs purs deviennent le cas limite d'opérateurs dont le domaine actif est vide. C'est une sémantique totale par défaut, robuste pour un système de stockage.
2. **La forme de la déclaration a gagné en généralité.** L'ancienne forme $((L_1, L_2) \mapsto \dots)$ se reformule comme le cas particulier $g(L_1, L_2) \mapsto \dots$ — motif de tête g maximale, ne pouvant échouer. La nouvelle forme suggère naturellement la **définition par cas** : une suite de couples motif $\mapsto$ production, essayés dans l'ordre ($g := (g(a, L_1) \mapsto \dots, g(f(L_1), L_1) \mapsto \dots, \dots)$), à la façon des équations de Haskell ou des clauses Prolog. Question ouverte : les motifs pourront-ils être ambigus (plusieurs s'appliquant au même terme), et si oui, priorité au premier ?
3. **La terminaison redevient possible mais non garantie.** Ici la production $g(a, L_1)$ sort du domaine (a ne filtre pas... si : $g(a, f(a))$ est dans le domaine ! $g(a, f(a)) \rightarrow g(a, a)$, qui lui sort du domaine — deux pas et l'on termine). Le contrôle par motif borne l'évaluation dans bien des cas, sans l'assurer en général.

Reste en suspens la **stratégie** : l'évaluation porte-t-elle sur la seule tête du terme, ou descend-elle dans les sous-termes ($g(b, f(b))$ enfoui dans un grand terme est-il réduit) ? Et itère-t-on jusqu'à stabilité ? J'attends la suite.

Les membres droit et gauche de $\mapsto$ peuvent invoquer des opérateurs d'autres langages, dans le respect des sortes.

Bien noté. Les définitions traversent les frontières de langages : membre gauche (motif) et membre droit (production) peuvent invoquer des opérateurs de tout langage, la seule loi étant le **respect des sortes** — chaque position d'argument reçoit un terme de la sorte attendue, et les deux membres du $\mapsto$ doivent être de la même sorte que l'opérateur défini (le remplacement doit préserver le typage du terme hôte).

Exemple sur nos langages $L = \langle a, b, f(H), g(H, L) \rangle$ et $H = \langle e, j(H, H), k(H, L, L) \rangle$:

$$g := (g(j(H_1, H_1), L_1) \mapsto g(H_1, f(H_1)))$$

Définition de $L.g$ (sorte du résultat : $L \checkmark$ des deux côtés), dont le motif contraint le premier argument — de sorte H — à être une diagonale de j , et dont la production invoque $f(H_1)$, légal car f attend un H et H_1 en est un. Évaluation : $g(j(e, e), a) \rightarrow g(e, f(e))$; tandis que $g(j(e, k(e, a, b)), a)$ reste inchangé (la coréférence $H_1 - H_1$ échoue).

Conséquences :

1. **Le filtrage devient multi-sorte mais reste homogène par position.** L'unification $t * \text{motif}$ descend dans le terme en changeant de sorte au gré des signatures, exactement comme les automates mutuellement récursifs ; chaque variable H_i ou L_i ne peut capturer qu'un terme de sa sorte, et l'unification d'un H_i avec un terme de L est un échec *de sorte* — vérifiable statiquement dans la déclaration, d'ailleurs : une déclaration mal sortée ($g(f(L_1), \dots)$) — f n'accepte pas un L) peut être rejetée à l'écriture, avant toute évaluation.
2. **Le calcul circule entre les langages.** Une évaluation dans L peut construire des sous-termes de H et réciproquement ; les définitions de H peuvent dépendre d'opérateurs de L ($H.k := (k(H_1, L_1, L_1) \mapsto j(H_1, H_1))$) — motif diagonal sur les deux L , production interne à H). Le graphe de dépendances entre langages, déjà présent dans les signatures, se double d'un graphe de dépendances *calculatoires*.
3. **Une asymétrie à noter : la tête du motif reste dans le langage de l'opérateur défini.** Le motif d'une définition de $L.g$ a nécessairement g pour tête — on définit g , on filtre donc des termes de tête g . La liberté inter-langages porte sur les *sous-termes* du motif et sur la production entière (laquelle, elle, n'a aucune contrainte de tête : $g(\dots) \mapsto a$ est licite, la production étant simplement un terme de sorte L quelconque — voire une variable nue : $g(H, L_1) \mapsto L_1$, la projection).
4. **La qualification des noms est résolue par les sortes.** Si L et H possèdent chacun un opérateur f , l'écriture $f(H_i)$ dans une définition désigne sans ambiguïté celui dont la signature accepte la sorte des arguments en position — et en cas d'ambiguïté résiduelle (deux f de même profil), la qualification explicite $L.f$ trancherait. Les chaînes stockées, elles, restent non ambiguës par stratification syntaxique, comme établi.

Le système est désormais un cadre de **réécriture multi-sorte** : signatures typées, syntaxes par langage, définitions par filtrage traversant les sortes. Manquent toujours la stratégie d'évaluation (portée et itération) et, peut-être, les définitions par cas multiples. J'attends la suite.

