UNIVERSITÉ PARIS XI

U.E.R. MATHÉMATIQUE

91405 ORSAY FRANCE

N° 156_75.45

Georges KIREMITDJIAN

SISGOF

Note ECSTASM N° 12

Publication Mathématique d'Orsay

SISGOF

Réalisation du système SISGOF : Système Interactif de Spécification, Génération d'Objets Formels.

Ce système aide l'utilisateur à définir un langage par la donnée de

. sa syntaxe (sous forme de régles de Backus)

. sa sémantique (à l'aide des attributs de Knuth)

. les traitements à effectuer en cas d'erreur.

La définition étant acceptée, il fournit les tables syntaxiques et les fonctions sémantiques nécessaires au programme qui effectue le traitement des lignes écrites dans le langage ainsi défini . Par sa souplesse, ce système apporte une aide appréciable lors de la définition d'un nouveau langage.

S I S G O F

Système interactif de spécification et de génération d'objets formels.

Il existe dans la littérature informatique, des descriptions de générateur de compilateurs ou en anglais Translator Writing Systems. Ces systèmes construisent un programme, appelé compilateur qui accepte un langage défini par l'utilisateur lors de sa génération. Accepter un langage, signifie que le compilateur effectue un certain traitement qui est fonction des phrases introduites. Dans le cas où une phrase ne satisfait pas à la définition du langage, le compilateur la rejette, en essayant de diagnostiquer la cause de l'erreur. Dans le cas contraire, il peut par exemple, générer un programme "équivalent" dans un autre langage (en général langage-machine). On peut considérer la définition d'un compilateur comme un système formel particulier. L'analyse syntaxique de phrase source étant la vérification de son appartenance au langage, de la même façon qu'une formule est valide dans un système formel, si elle est dérivable de l'axiome, en utilisant les règles de déduction. La sémantique en est alors une interprétation au sens de la logique. Les règles sémantiques définissent la correspondance entre ces deux langages, de façon à avoir des programmes "équivalents".

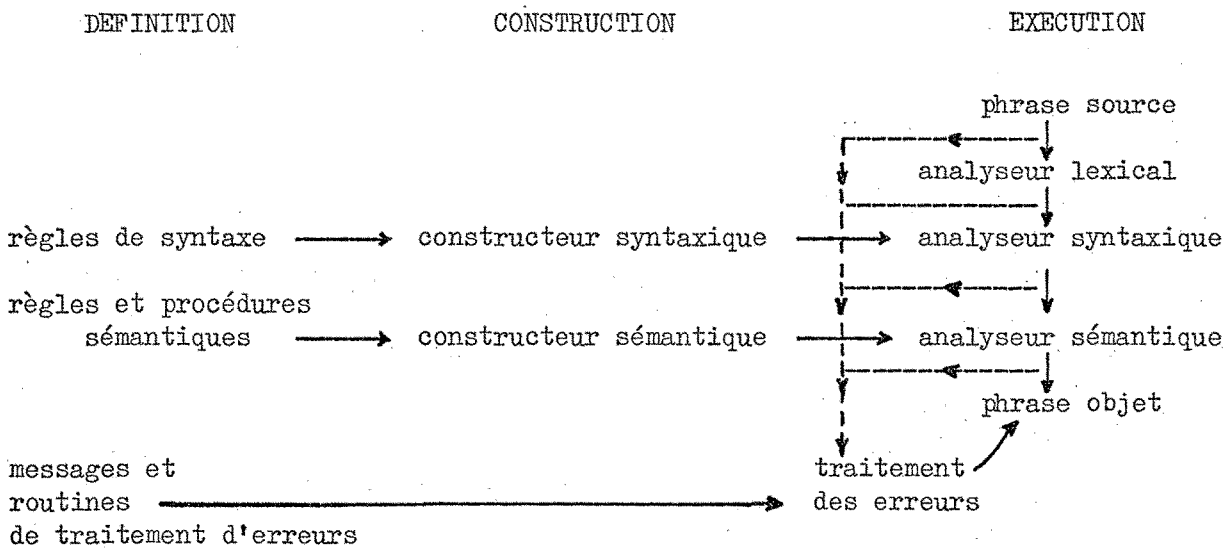
Les progrès faits dans le domaine des générateurs de compilateurs peuvent donc s'appliquer à la construction de systèmes formels. On peut ainsi arriver à décrire un système formel de façon très simple. Les règles syntaxiques sont sous forme de Backus, par exemple, et les règles sémantiques sous une forme voisine. On libère ainsi l'utilisateur de toute la partie fastidieuse de la réalisation effective, en restant au niveau d'une définition purement logique.

Les avantages d'une génération automatique tels la fiabilité, la modifiabilité ... compensent la relative lenteur des systèmes générés. Un tel système comprend deux parties :

. définition et construction du système formel

. traitements sur des objets du système formel effectués suivant les règles propres à ce système.

Pour ces deux parties, on retrouve les deux aspects principaux qui caractérisent un système formel : sa syntaxe et sa sémantique. L'aspect pragmatique se situe au niveau de l'analyse lexicale et du programme de traitement des erreurs.



La programmation du système est modulaire. On peut changer une des fonctions précédentes en respectant le format des données utilisées en argument. La description des différents modules est l'objet des chapitres suivants. Le dernier chapitre est consacré à un exemple d'utilisation.

ANALYSE LEXICALE

Le rôle de cette phase est purement pragmatique. Elle sert d'interface entre la forme d'entrée des lignes et le reste du système.

Son principal objet est de vérifier les caractères introduits, supprimer les blancs et faire correspondre aux mots leurs catégories syntaxiques. Sur le vocabulaire initial, on définit une partition en classes syntaxiques, car pour l'analyse syntaxique certains mots sont équivalents. Par exemple, il faut savoir si un mot est un identificateur un nombre ou un opérateur primitif, mais pour l'analyse syntaxique il importe peu que le nombre soit 1 ou 3.14. Cette valeur sera utilisée pour l'évaluation lors de l'analyse sémantique.
On pourrait se passer de cette phase et demander à l'analyse syntaxique de réaliser cette reconnaissance. L'analyseur syntaxique pouvant accepter un langage de Chomsky sera moins performant sur la partie de reconnaissance des classes syntaxiques car étant plus général, il doit choisir entre plus de possibilitées qu'un automate à nombre fini d'état. De plus cette tâche augmentera le nombre de règles de syntaxe et diminuera ainsi l'efficacité globale de l'analyseur syntaxique.

Cette séparation des tâches dues à des raisons pragmatiques d'efficacité est aussi logique car elle clarifie le processus.

Chaque étape ayant un rôle bien défini, on peut avoir une véritable

modularité et une certaine independance des fonctions fondamentales entre elles. Voyons maintenant quelles sont les classes syntaxiques reconnues.

- Les identificateurs qui seront classés après reconnaissance en deux parties.

  les mots clés du langage (s'il y en a)

  les variables.

- les nombres ne sont pas divisés en sous-classes : booleen, entier, flottant car le système hôte (APL) fait les conversions automatiquement entre ces différents types.

- les chaînes de caractères
- les symboles spéciaux : séparateurs, opérateurs;ils sont limités à un caractère.

Définition de l'automate

Un automate fini est un quadruplet

$$< V_T , S , F_T , A >$$

ou V_T est le vocabulaire d'entrée

ici LETTRE U CHIFFRE U BL U QOTE U SYMB

S est l'ensemble fini des états

F_T est la fonction de transition

A est l'ensemble des actions.

$F_T : S \times V_T \rightarrow S \times A$

$(S_i, v) \rightarrow (S_f, a)$

Etant donné un état initial S_i et un caractère courant v l'automate se met dans l'état S_r et exécute l'action a .

Ici l'ensemble des états est :

S_o état initial (état pivot)

ID identificateur

NB nombre

VT symboles spéciaux

ST Chaîne de caractères

ST' état auxiliaire

ERR état utilitaire.

Fonction de transition :

	lettre	chiffre	quote	symb	bl	autre
SO	(ID,2)	(NB,2)	(ST,1)	(VT,4)	(SO,1)	(ERR,6)
ID	(ID,2)	(ID,2)	(SO,5)	(SO,5)	(SO,5)	(SO,5)
NB	(ERR,6)	(NB,2)	(ERR,6)	(SO,3)	(SO,3)	(SO,3)
VT	(SO,0)	(SO,0)	(SO,0)	(SO,0)	(SO,0)	(SO,0)
ST	(ST,2)	(ST,2)	(ST;3)	(ST,2)	(ST,2)	(ST,2)
ST'	(ERR,6)	(ERR,6)	(SO,1)	(SO,1)	(SO,1)	(ERR,6)

Le nombre qui apparaît correspond aux actions suivantes :

0 : ne rien faire

1 : passer au caractère suivant

2 : former le mot et passer au caractère suivant

3 : empiler la classe syntaxique du mot et un pointeur dans la table des symboles dans la matrice résultat et (1) .

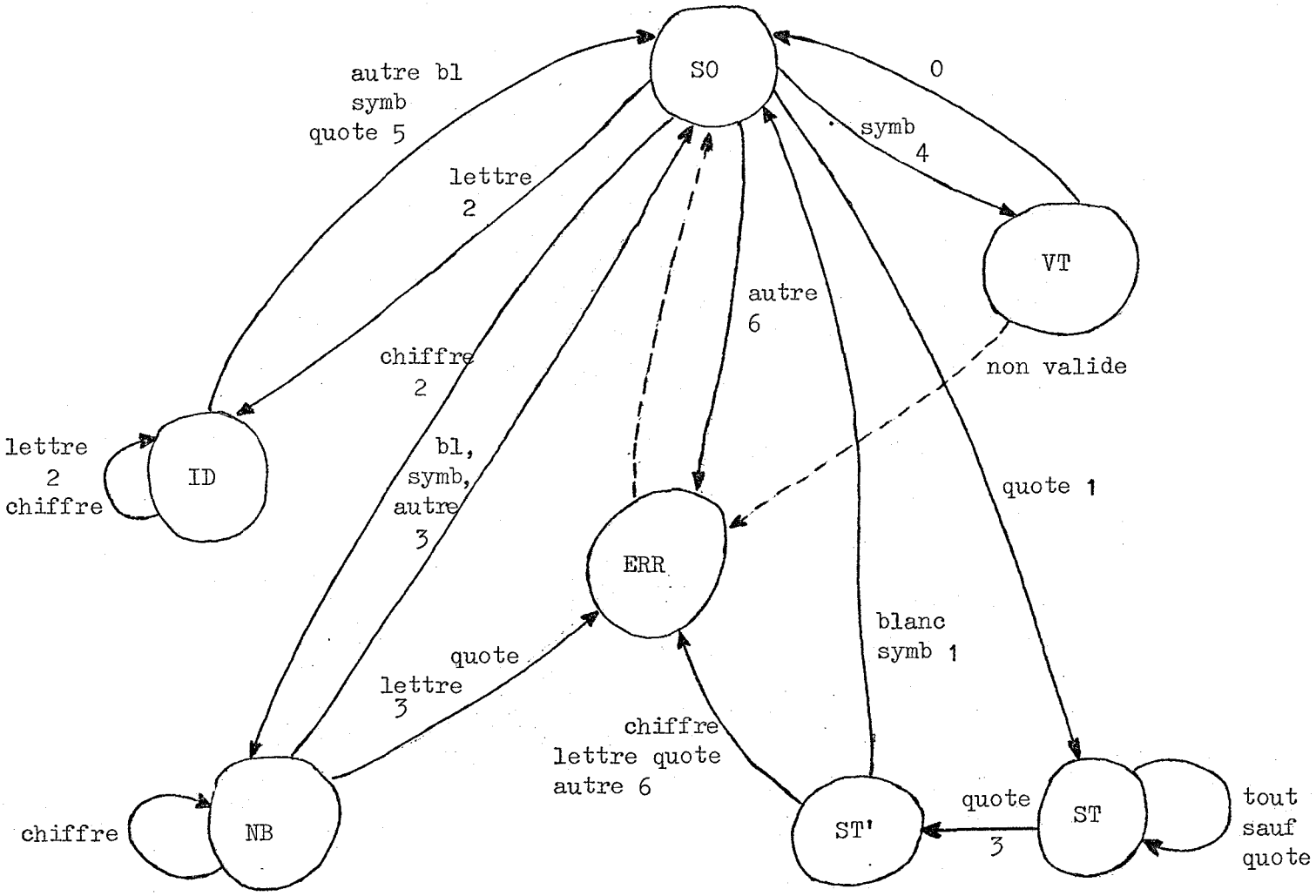
4 : tester si l'opérateur ou le séparateur est utilisé dans la grammaire, si oui (3*) sinon (6) .

5 : tester si c'est un mot-clé de la grammaire si oui (3*) sinon (3) .

6 : éditer un message d'erreur et (1)

(3*) Comme on ne range pas le mot dans la table des symboles, le pointeur vaut alors 0 .

On peut aussi en donner une représentation sous forme de diagramme



Pour l'état SO il y a un test à cinq directions vers VT, ST, ID, NB, ERR suivant le caractère lu.

De même en NB et ST' il y a des tests à deux directions.

SO est un état pivot car pour minimiser les tests aux différents états on préfère repasser par SO .

Les arcs qui conservent l'état sont réalisés de façon particulière.

Pour NB et ID ont utilisé une fonction match qui balaye la chaîne d'entrée (TEXT) tant que le caractère lu appartient à l'ensemble défini comme premier argument

ID : (LETTRE, CHIFFRE) MATCH. TEXT.

Pour ST on utilise scan qui balaye le texte jusqu'à trouver une apostrophe : ST : QOTE SCAN TEXT.

Après la description de l'automate voyons la deuxième composante fonctionnelle fondamentale de l'analyseur lexical.

Nous avons vu que l'automate transforme un texte source en une matrice de pointeurs.

La première ligne de celle-ci se réfère au dictionnaire de la grammaire du langage choisi.

La deuxième vers la table des symboles qui est la seconde composante de l'analyseur lexical.

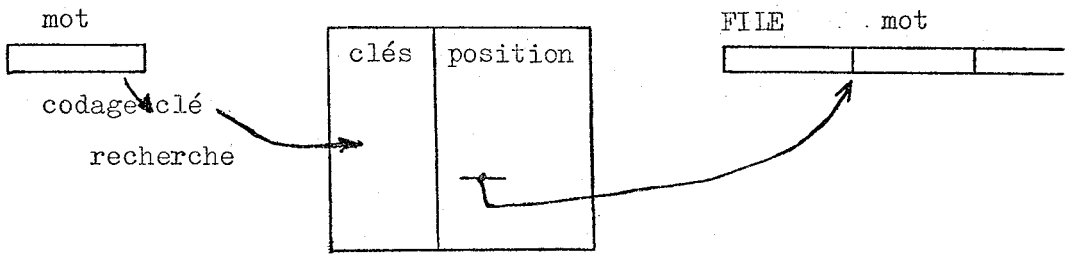
Organisation de la table des symboles :

Il s'agit de ranger en mémoire des chaînes de caractères qui correspondent aux mots reconnus. Si un même mot apparaît plusieurs fois,il est rangé une fois et toutes ses occurences pointent sur lui.

La fonction à réaliser est donc définie comme suit :
+ si un mot est nouveau, il est rangé et on renvoie sa position.
+ si un mot est déjà en mémoire, on renvoie la position où on l'a trouvé.

Les mots pouvant être de longueur variable on utilise une technique de codage du mot et on passe par une table de correspondance entre le code d'un mot et sa position dans la zone de rangement.

TABLE



Il se pose donc trois problèmes :

- fonction de codage mot - clé
- recherche d'une clé dans la table
- recherche d'un mot connaissant sa position.

* Fonction de codage choisie.

la clé contiendra les informations suivantes :

. longueur du mot

. premier caractère du mot.

. dernier caractère du mot.

Cette fonction n'est pas biunivoque mais dans beaucoup de cas, elle doit pratiquement réduire le nombre de mot qui correspond à la même clé. La clé doit être un nombre pour cela on convertit ces trois nombres : longueur, code des caractères en un nombre de base 256 (les codes caractères et la longueur étant inférieurs à 256).

clé $(\text{longueur} \times 256^2) + (\text{code 1er car} \times 256) + \text{code dernier car}$

*Recherche d'une clé dans la table.

Cette recherche peut être séquentielle ou dichotomique suivant la taille de la table.

Dans les deux cas, où bien la clé n'y est pas, alors on la range à la fin en séquentiel (on l'insère dans l'ordre de façon à conserver une table triée en recherche dichotomique).
le mot étant nouveau, on le range dans FILE, on note la position dans TABLE et on la renvoie comme résultat. Par contre si la clé existe déjà, on note la position du mot trouvé et on le compare au mot à introduire. S'ils sont identique on renvoie la position trouvée comme résultat. Sinon il faut faire de même pour les autres clés identiques. Si onne l'a pas trouvé après avoir épuisé toutes les clés de même code, on range le mot et on ajoute à la table la clé et la position du mot nouveau ainsi introduit. On renvoie la position de rangement.

* Lecture d'un mot connaissant sa position :

On utilise la table en y accédant par les positions.
Si la position indiquée, ne correspond à aucun mot on renvoie un résultat vide. Si un mot lui est associé, on décode la clé qui fournit alors la longueur du mot. Ayant la position du début du mot et sa longueur il est facile de l'extraire et de le renvoyer comme résultat de la lecture.

ANALYSE SYNTAXIQUE

Cette phase a pour objet de vérifier la syntaxe du texte introduit. Pour cela elle doit en exhiber la structure grammaticale. Si la syntaxe est correcte, le résultat de cette phase est un arbre représentant cette structure. A l'aide de la grammaire du langage, on explicite, l'information structurelle contenue dans le texte introduit. L'analyse sémantique se fera en utilisant l'arbre d'analyse syntaxique, la valeur des éléments terminaux et les règles sémantiques qui en combinant ces deux sources d'information permettent d'interpréter le texte. Pour formaliser un langage et donc le rendre apte a être traité de façon automatique, il faut définir sa syntaxe et sa sémantique. La définition de la syntaxe se fait par la donnée d'une grammaire. Les phrases non conformes à la grammaire ne sont pas syntaxiquement correctes et de ce fait n'appartiennent pas au langage. Un grammaire non ambigüe permet de générer un langage unique (du point de vue syntaxe). Mais un langage peut-être défini de manière équivalente par plusieurs grammaires qui peuvent être de complexitées différentes.

Pour un langage de programmation, on peut donc dans la grande majorité des cas, trouver une grammaire de Chomsky qui soit satisfaisante. De plus, une forme de définition de la syntaxe des langages de programmation très répendue, celle de Backus-Naur, permet de définir des grammaires de Chomsky. Nous ne nous intéresserons donc qu'aux grammaires de Chomsky, cette restriction n'étant pas très gênante dans la pratique.

Définition d'une grammaire et du langage associé.

Une grammaire est définie par le quadruplet.

$$G = < V_T , V_N , A, P >$$

où V_T est l'alphabet terminal

V_N est l'alphabet non terminal

A est un élément distingué de V_N : l'axiome

P est l'ensemble fini des règles de dérivation.

$$V = V_T \cup V_N \quad \text{et} \quad P = \{\varphi_i \in V^* , \psi_i \in V^* / \varphi_i \to \psi_i\}_{i\in(1,n)}$$

V^* est le semi-groupe engendré par V .

$\to$ est une relation de substitution.

Condition d'invariance par changement de contexte.

Dans un mot si on trouve une occurence de φ_i on peut la remplacer par ψ_i .

Si $\varphi_i \to \psi_i$ et $\forall \alpha,\beta \in V^* \cup \emptyset$ alors $\alpha\varphi_i\beta \Longrightarrow \alpha\psi_i\beta$

Notons $\overset{*}{\Longrightarrow}$ la clôture transitive, réflexive de $\Longrightarrow$.

Si $\alpha\omega\beta \overset{*}{\Longrightarrow} \alpha\omega'\beta$ alors $\exists \{\varphi_i\}_{i\in I}$ I ensemble fini tel que

$\{\varphi_i \to \varphi_{i+1}\}_{i\in(o,n)} \in P$ et $\varphi_o = \omega$, $\varphi_n = \omega'$.

Quand on a $A \overset{*}{\Longrightarrow} \omega$, on dit que ω est une forme sententielle. Les formes sententielle appartenant à V_T^* sont les mots du langage L engendré par la grammaire G .

$$L(G) = V_T^* \cap \{\omega / A \overset{*}{\Longrightarrow} \omega\} .$$

Grammaire de Chomsky (context-free).

Les règles de dérivation ne peuvent être que de la forme $N \to \omega$ avec $N \in V_N$ et $\omega \in V^*$. Il y en a bien sûr du moins une ayant l'axiome A comme partie gauche.

Dérivation canonique.

$\varphi_i \overset{}{\underset{c}{\Longrightarrow}} \psi_i$ si $\varphi_i = \alpha N \tau$ et $\psi_i = \alpha \omega \tau$ et $(N \to \omega) \in P$

avec $\alpha \in V^*$ et $N \in V_N$, $\omega \in V^*$ et $\tau \in V_T^*$.

On peut alors dire que ψ_i est obtenue à partir de φ_i par remplacement du non-terminal le plus à gauche. On définit de même que précédemment $\overset{*}{\underset{c}{\Longrightarrow}}$ par la clôture reflexive et transitive de $\overset{}{\underset{c}{\Longrightarrow}}$.

On peut montrer que $\forall t \in L(G)$, $A \overset{*}{\underset{c}{\Longrightarrow}} t$ il existe une dérivation canonique $\{\varphi_i\}_{i \in (o,n)}$ pour tout $t \in L(G)$ $A = \varphi_o \overset{}{\underset{c}{\Longrightarrow}} \varphi_1 \ldots$ $\varphi_i \overset{}{\underset{c}{\Longrightarrow}} \varphi_{i+1} \overset{}{\underset{c}{\Longrightarrow}} \overset{}{\underset{c}{\Longrightarrow}} \varphi_n = t$ ou $\varphi_i = \alpha N \tau$ et $\varphi_{i+1} = \alpha \omega \tau$ $\alpha \in V^*$, $\tau \in V_T^*$ et $(N, \omega) \in P$.

Une grammaire de Chomsky est ambigüe s'il existe plusieurs dérivations canoniques pour un mot du langage.

Après la définition d'une grammaire de Chomsky non ambigüe, voyons comment faire la liaison avec la définitionsous la forme de Backus-Naur.

Il faut pour cela imposer quelques conditions à la forme de la définition de la grammaire.

1. l'axiome est défini à gauche de la première règle de dérivation. on impose aussi qu'il n'apparaisse jamais à droite d'une règle de dérivation.

2. la grammaire est réduite tout les non-terminaux sont accessibles

$\forall N \in V_N \quad \exists \alpha,\beta \in V^* \text{ et } \tau \in V_T^* \qquad tq \quad A \overset{*}{\Longrightarrow} \alpha N \beta \overset{*}{\Longrightarrow} \tau$

Ce qui impose que tous les non-terminaux apparaissent à gauche d'au moins une règle de dérivation.

3. Il n'y a pas d'impasse

$$\forall N \in V_N \quad \text{on n'a jamais} \quad N \overset{*}{\Longrightarrow} \phi$$

4. Il n'y a pas de boucle

$$\forall N \in V_N \quad \text{on n'a jamais} \quad N \overset{*}{\Longrightarrow} N \ .$$

La définition de Backus-Naur est en fait le donnée des règles de dérivation sous la forme

$$Ni = \varphi / \psi / \omega$$

Ce qui se traduit par $(N,\varphi) \in P$ et $(N,\psi) \in P$ et $(N,\omega) \in P$.

Maintenant partant de P il reste à définir V_N , V_T et A .

. A est la partie gauche de la première règle de production d'après la condition 1).

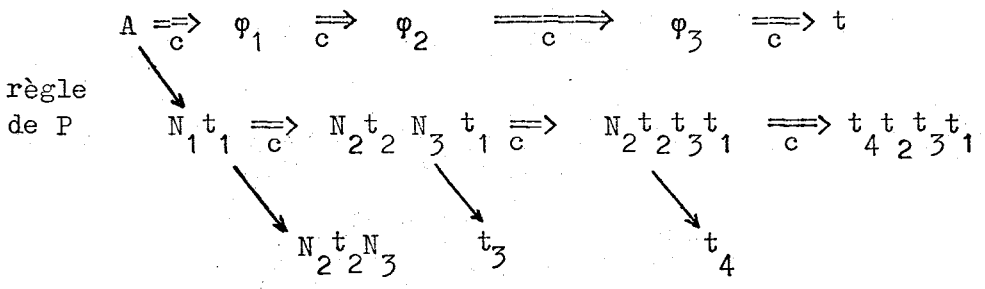
. $V_N = \{N \mid \exists \omega \text{ et } (N,\omega) \in P\}$ l'ensemble des parties gauches des règles de P d'après 2) .

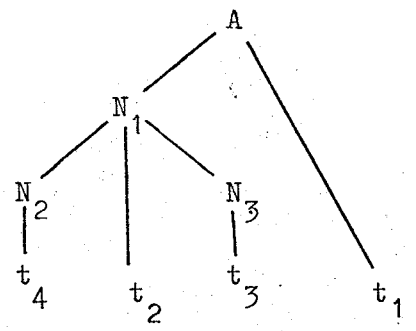
. $V_T = \{u \mid \exists N , \exists \varphi$ et $\phi \in V^* \cup \{\phi\} / (A, \varphi a \phi) \in P\} - V_N$ donc les lettres des parties droites qui n'apparaissent jamais à gauche d'une règle.

Le constructeur d'analyseur syntaxique utilise comme données les règles de syntaxes sous forme de Backus-Naur. On définit bien ainsi une grammaire de Chomsky.

Arbre d'analyse syntaxique.

On peut donner une représentation de la dérivation canonique d'un mot appartenant à un langage de Chomsky. Cette représentation sous forme d'arbre est unique si la grammaire est non ambiguë. Elle contient toute l'information syntaxique associée au mot.





Le sommet de l'arbre est l'axiome A .

Les noeuds sont des non-terminaux de la grammaire .

Les feuilles sont des terminaux et on passe d'un noeud à ses fils par application d'une règle

$$(N \to N_1 \text{ -- } N_p) \in P \qquad N \in V_N \text{ et } N_i \in V$$

Le problème est donc, ayant d'une part la définition de la syntaxe et, d'autre part, un mot de décider si celui-ci appartient au langage (s'il est syntaxiquement correct). Pour cela, il faut trouver l'arbre syntaxique ayant pour sommet l'axiome et pour feuille le mot. L'analyse syntaxique consiste en la construction de cet arbre. Si l'on échoue, le mot n'appartient pas au langage. Si l'on réussit, le mot est syntaxiquement correct, et on transmet l'information syntaxique contenue dans l'arbre à la phase suivante d'analyse sémantique.

Peut-on trouver un algorithme pour la construction de l'arbre syntaxique ?

Les grammaires de Chomsky que nous avons définies n'ayant pas d'impasse, le passage d'une forme sententielle à une forme dérivée ne peut qu'augmenter le nombre de lettres. Ayant un mot de longueur ℓ , partant de l'axiome et en appliquant toutes les dérivations canoniques possibles tant que la longueur de la forme sententielle obtenue est inférieure ou égale à ℓ on obtient un nombre fini de formes ξ sententielles de longueur ℓ . (le nombre de règles de dérivations est fini).

Parmi celles appartenant à V_T^* , il faut trouver le mot.
Le problème est donc décidable. Mais ce n'est pas suffisant car il faut que l'algorithme ait un temps de réponse "raisonnable".

Voyons donc les différentes techniques possibles.

Pour la construction de l'arbre syntaxique on peut adopter deux approches : on part de l'axiome et on essaie d'atteindre les feuilles. C'est la méthode descendante. Ou bien on part du texte (les feuilles) pour atteindre l'axiome. C'est la méthode ascendante.

Dans les deux cas, il faut imposer une restriction aux langages analysables pour essayer d'obtenir un temps de réponse raisonnable (proportionnel à la longueur du texte analysé).

Les langages déterministes.

A tout langage de Chomsky, on peut associer un automate à pile le reconnaissant. Si l'on impose à cet automate, une condition de non retour sur la partie du texte déjà analysé (c'est-à-dire qu'il ne s'engage pas dans une impasse ce qui l'obligerait à revenir en arrière) on peut espérer avoir un temps de réponse raisonnable. Cette restriction entraîne une diminution de la classe des langages acceptés. Il a été démontré par Knuth que la classe des langages de Chomsky déterministes est engendrée par les grammaires LR(k) .

Choix d'une méthode d'analyse.

Au point de vue étendue des langages acceptés, la meilleure méthode est celle de Knuth par les langages LR(k). Cependant, étant la plus

générale, on peut penser que pour des grammaires simples, elle sera moins performante que des méthodes plus simples qui acceptent une classe de langages plus réduite.

Du point de vue de l'utilisation pratique, il faut donc faire l'essai de différentes méthodes pour pouvoir faire un choix justifié. On peut aussi penser que ce choix peut dépendre de la structure de la grammaire étudiée. Etant donnée la conception modulaire du système on peut réaliser le constructeur et l'analyseur de son choix, il suffit de respecter les formats des données d'entrée et de sortie. On peut par exemple envisager trois types d'analyseur

- LR(1) et ses dérivées LALR(1) et SLR(1) de Knuth
- RCF méthode descendante des diagrammes de transition de Conway et Tixier
- et la méthode que nous avons implantée, qui est une méthode de précédence étendue de Mc Keeman qui l'appelle MSP(2,1;1,1) (mixed strategy precedence).

Nous allons donner ici une description de l'algorithme d'analyse. La façon de construire les tables qu'il utilise sera décrite dans la partie consacrée au constructeur syntaxique.

Principe de l'analyseur.

C'est un analyseur ascendant déterministe. Pour une grammaire non ambiguë, chaque mot a une dérivation canonique unique.

$$\forall\, t \in L(G) \quad \exists\, \{\varphi_i\}_{i \in (o,n)} \text{ tq}$$

$$A = \varphi_o \underset{c}{\Longrightarrow} \varphi_1 \underset{c}{\Longrightarrow} \cdots \varphi_i \underset{c}{\Longrightarrow} \varphi_{i+1} \underset{c}{\Longrightarrow} \cdots \varphi_{n-1} \underset{c}{\Longrightarrow} \varphi_n = t$$

$$\varphi_i \underset{c}{\Longrightarrow} \varphi_{i+1} \qquad \begin{matrix} \varphi_i = \alpha N \tau \\ \varphi_{i+1} = \alpha \omega \tau \end{matrix} \qquad (N,\omega) \in P \qquad \begin{matrix} \alpha \in V^* \\ \tau \in V_T^* \end{matrix} .$$

Le problème est de trouver $\{\varphi_i\}$, c'est-à-dire, l'analyse étant ascendante, trouver la fonction $\mathcal{C}$ telle que $\mathcal{C}(\varphi_{i+1}) = \varphi_i$. Pratiquement la fonction $\mathcal{C}$ elle-même n'est pas utilisable étant donné le nombre de formes sententielles possibles. En fait, il suffit d'isoler ω dans φ_{i+1} pour pouvoir appliquer (N,ω) . Pour cela, on utilise une fonction $\mathcal{C}_1$ qui détermine la frontière droite de ω dans

φ_{i+1} et une fonction $\mathcal{C}2$ qui sélectionne la règle (N,ω) à appliquer.

σ étant une pile de travail

τ étant la partie du texte restant à analyser.

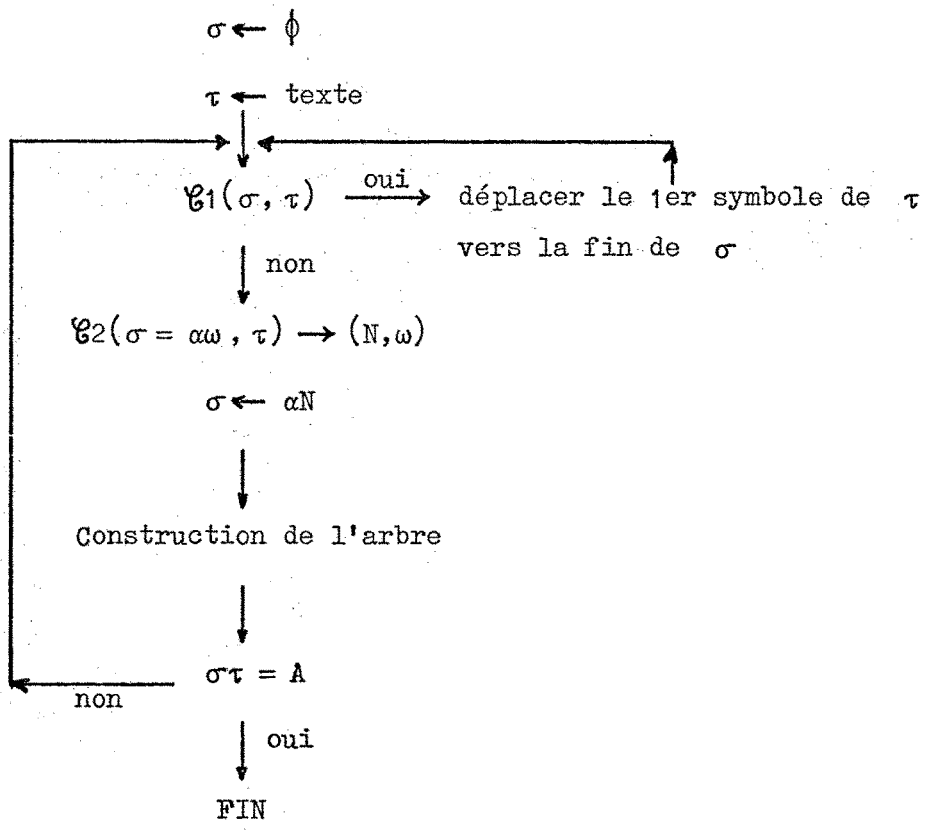
Si $\varphi_i \xRightarrow{c} \varphi_{i+1}$ on doit avoir $\sigma = \alpha\omega$ alors

$\mathcal{C}2(\sigma,\tau)$ donne la règle (N,ω) et après réduction

$\sigma = \alpha N$. $\mathcal{C}2$ est donc une fonction de sélection de règle.

Si $\sigma \neq \alpha\omega$ il faut déplacer le premier symbole de τ vers la fin de σ jusqu'à avoir $\sigma = \alpha\omega$. $\mathcal{C}1(\sigma,\tau)$ est la fonction de décision qui pilote cet empilement.

Algorithme de principe :



Comme il n'est pas possible de tabuler les fonctions $\mathcal{C}1$ et $\mathcal{C}2$ pour tout σ et τ , on limite la longueur des arguments. $\mathcal{C}1$ est limité au contexte (p,q) en prenant les p derniers caractères de σ et les q premiers de τ de même $\mathcal{C}2$ à (m,n).

On peut le faire car pour certaines classes de grammaire (dites à contexte borné Floyd) σ et τ pris en entiers sont redondants.

Dans ce cas par exemple

$$\mathcal{C}_1(\sigma, \tau) \equiv \mathcal{C}_1(1-\rho)\uparrow\sigma\ ,\ q\uparrow\tau)$$

la décision correcte peut être prise en regardant seulement les p derniers caractères de σ et les q premiers de τ. L'algorithme de construction des tables $\mathcal{C}_1$ et $\mathcal{C}_2$ utilise un contexte $(2,1)$ pour $\mathcal{C}_1$ et $(1,1)$ pour $\mathcal{C}_2$. Si la grammaire n'est pas MPS$(2,1;1,1)$ l'algorithme l'indique. Donc s'il fournit les tables on est sûr que les décisions prises seront correctes.

Pour $\mathcal{C}_1$ il construit une table pour un contexte $(1,1)$ qui indique

1) contexte valide, empiler

2) contexte valide, ne pas empiler

3) contexte valide, la décision ne peut être prise dans ce contexte

4) contexte invalide, erreur de syntaxe.

Dans le cas 3) seulement on étudie un contexte $(2,1)$. En agissant ainsi on utilise les triplets seulement si la grammaire n'est pas $(1,1)$ et seulement aux endroits où elle est localement ambiguë avec un contexte $(1,1)$.

Pour $\mathcal{C}_2$, le choix de la règle se fait en ne considérant que les règles dont le dernier caractère est identique au dernier symbole de σ. On applique d'abord la règle la plus longue dont la partie droite est identique au haut de pile σ. Mais cette règle ne suffit pas, en particulier dans le cas de règles d'égales longueurs.

Après avoir trouvé une règle applicable on consulte une table qui donne pour chaque règle l'action à entreprendre

- l'accepter et réduire
- tester le contexte droit
- tester le contexte gauche
- tester les contextes droit et gauche.

Si le test est positif on applique la réduction. Sinon on cherche une autre règle. Si l'on n'en trouve pas, une erreur de syntaxe est alors détectée. L'analyseur fournit cette table et les différents contextes valides.

Réalisation :

Le superviseur de compilation COMPILE effectue les initialisations de σ et τ appelés STAK et TEXT. Il appelle et contrôle les fonctions STAKING et REDUCE. Puis il teste la fin de l'analyse.

STAKING est la fonction d'empilement, elle utilise les tables de décisions C11T et TRIPLE.

REDUCE est la fonction de réduction, elle trouve la règle à appliquer parmi PROD (ensemble des règles) puis cherche dans CONTEXT s'il faut tester

le contexte droit avec C11T

le contexte gauche avec LC

les contextes gauche et droit avec LRC

TRUC est un marqueur (par exemple '|')

Algorithmes en pseudo-APL *

Compile (text)

```
                    stak ← truc
initialisation      text ← text , truc
                B : staking(stak , text)
contrôle de validité  → fin if ~ VAL
                    reduce(stak ; text)
                    → fin if ~ VAL
test de sortie      → B if (stak , text) ≠ truc, axiom, truc
                    fin
```

Staking (stak ; text)

```
C11T peut prendre les valeurs → C11T[-1↑stack ; 1↑text]
'-','V','F','C'             '-' : ERREUR
                            'V' : stak ← stak , 1↑text
                                  text ← 1↓text
empilage et appel                 staking(stak ; text)
récursif de staking         'F' : → fin
TRIPLE peut valoir 'V','F'  'C' : → TRIPLE[-2↑stak ; 1↑text]
                            fin
```

* Notations utilisées pages SY13

Reduce (stak ; text)

sélection de la règle

peut valoir : aucun, droit, gauche droit-et-gauche

autre : règle ← PROD 2 stak
ERREUR si pas de règle trouvée
→ CONTEXT [règle]

aucun : { stak ← N remplace ω dans stak
construction de ΔARBRE et ΔVAL

droit : { → autre si test avec LC négatif
sinon → aucun

gauche : { → autre si test avec LC négatif
sinon → aucun

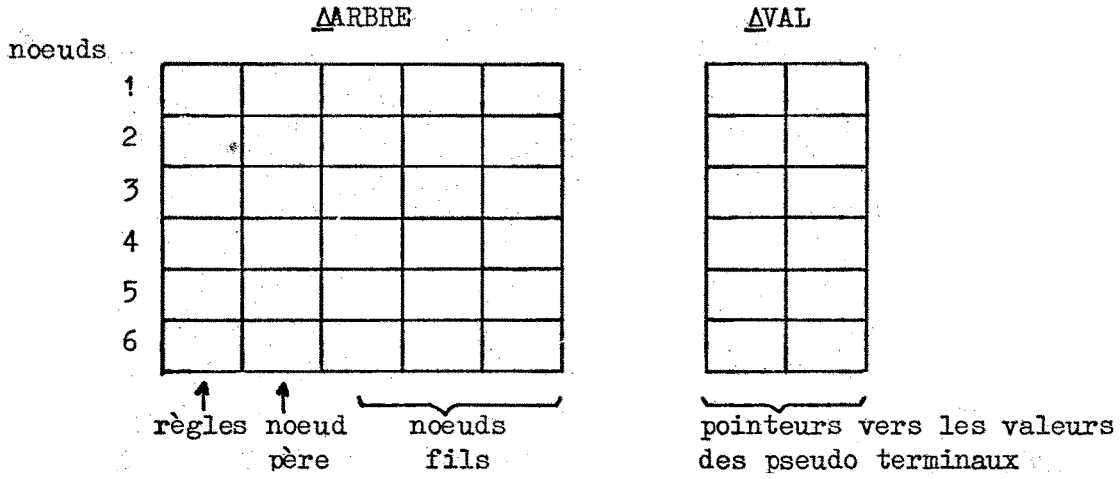
droit et gauche : { → autre si test avec LRC négatif
sinon → aucun.

fin

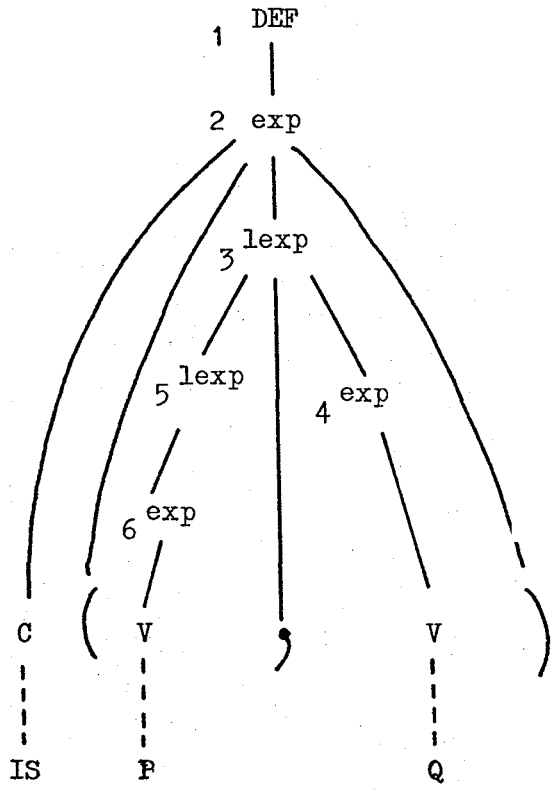
ERREUR positionne VAL à 0 et émet un message d'erreur.

Structure de l'arbre syntaxique construit

Il est en deux parties ΔARBRE et ΔVAL . ΔVAL contient les valeurs des pseudoterminaux.



exemple

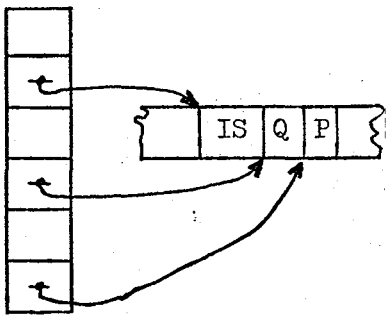


Règles : 1 Def → exp
2 exp → c (lexp)
3 exp → v
4 lexp → lexp , exp
5 lexp → exp

ΔARBRE

1	1	0	2	0
2	2	1	3	0
3	4	2	5	4
4	3	3	0	0
5	5	3	6	0
6	3	5	0	0

ΔVAL



Notations utilisées :

← affectation

A,B concaténation des vecteurs A et B

→ branchement

$n\uparrow A$ sélection des n premiers éléments de A

$-n\uparrow A$ sélection des n derniers éléments de A

$A[I;J]$ sélection de l'élément a_{ij} de la matrice A

$n\downarrow A$ sélection de A privé de ses n premiers éléments

LE CONSTRUCTEUR D'ANALYSEURS SYNTAXIQUES

Son objet est de générer à partir des règles de syntaxe, entrées sous la forme de Backus-Naur, les tables nécessaires à l'analyseur. Par des informations détaillées, il apporte une aide lors de la mise au point de la définition d'une grammaire acceptable pour le langage. De plus, une grammaire acceptée possède certaines propriétés dont la non-ambiguïté. Pour obtenir les tables des fonctions $\mathcal{C}_1$ et $\mathcal{C}_2$, on construit des tables auxiliaires ;

HDTB : table qui pour tout $y \in V$ donne l'ensemble des $x \in V$ qui sont premiers caractères des formes sententielles dérivées à partir de y (y lui-même appartient à cet ensemble).

F11 : table des contextes $(1,1)$ valides pour chaque non-terminal.

C'est à l'aide de ces tables que l'on obtient C11T , TRIPLE , TV , CONTEXT , LC , LRC ...

$$HDTB = \{(x,y) / x \overset{*}{\rightarrow} yw \quad \text{et} \quad x, y \in V, w \in V^*\} .$$

Après cette définition formelle, voyons comment en déduire un algorithme de calcul.

$\rightarrow$ est une relation binaire, on peut donc la représenter par une matrice booléenne r

$$\text{tq} \quad r(u,y) = \begin{cases} 1 & \text{si} \quad x \rightarrow y \\ 0 & \text{sinon} \end{cases}$$

exemple pour la grammaire

**** *PRODUCTIONS* *******

1	ΔE	::=	ΔT
2	ΔE	::=	ΔE + ΔT
3	ΔG	::=	ΔE
4	ΔP	::=	X
5	ΔP	::=	(ΔE)
6	ΔT	::=	ΔP
7	ΔT	::=	ΔT * ΔP

[1]	X	0	0	0	0	0	0	0	0	0	0
[2]	(	0	0	0	0	0	0	0	0	0	0
[3]	)	0	0	0	0	0	0	0	0	0	0
[4]	+	0	0	0	0	0	0	0	0	0	0
[5]	*	0	0	0	0	0	0	0	0	0	0
[6]	|	0	0	0	0	0	0	0	0	0	0
[7]	ΔE	0	0	0	0	0	0	1	0	0	1
[8]	ΔG	0	0	0	0	0	0	1	0	0	0
[9]	ΔP	1	1	0	0	0	0	0	0	0	0
[10]	ΔT	0	0	0	0	0	0	0	0	1	1

$\overset{+}{\to}$ est la clôture transitive de $\to$ on a :

1) Si $X \to Y$ alors $X \overset{+}{\to} Y$

2) Si $X \overset{+}{\to} Y$ et $Y \overset{+}{\to} Z$ alors $X \overset{+}{\to} Z$

r correspondant à $\to$ appelons r^+ la matrice correspondant à $\overset{+}{\to}$

$$r^+ = r^1 \vee r^2 \ldots \vee r^n \vee \ldots$$

avec $$r^{n+1} = r^n \wedge . \vee n$$

Si $x \overset{+}{\to} y$ alors il existe $\{\varphi_i\}$ tq $\varphi_o = x$ et $\varphi_n = y$ et $\varphi_i \to \varphi_{i+1}$ dans ce cas $r^n(x,y) = 1$.

Pour une grammaire donnée, il existe un n (inférieur au nombre de règles de productions) tel que $r^n = r^{n+1}$ alors $r^+ = r^1 \vee r^2 \vee \ldots \vee r^n$.

Méthode de calcul

On s'inspire de la technique qui consiste à former la suite des sommes partielles par avoir la limite d'une série.

On pose $r_i = r^1 \vee r^2 \ldots \vee r^i$

et en faisant le produit matriciel avec r

$$\begin{aligned} r_i \wedge . \vee r &= (r^1 \vee r^2 \ldots . \vee r^i) \wedge . \vee r \\ &= (r^1 \wedge . \vee r) \vee (r^2 \wedge . \vee r) \ldots . \vee (r^i \wedge . \vee r) \\ &= r^2 \vee r^3 \ldots . \vee r^{i+1} \end{aligned}$$

donc $r \vee r_i \wedge . \vee r = r \vee r^2 \vee r^3 \ldots . \vee r^{i+1} = r_{i+1}$ par définition

$$r_{i+1} = r \vee r_i \wedge . \vee r \quad .$$

C'est la relation de réccurence qui permet de calculer r^+ . On part de $r_1 = r$ et on s'arrête quand $r_{i+1} = r_i$ alors $r_i = r^+$. Pour obtenir $\overset{*}{\to}$ (clôture transitive et reflexive) à partir de $\overset{+}{\to}$, on fait un 'ou' de r^+ avec la matrice unité, ce qui revient à ajouter $x \to x \quad \forall x \in V$. Voici les différents pas du calcul par l'exemple

[1]	X	0	0	0	0	0	0	0	0	0	0
[2]	(	0	0	0	0	0	0	0	0	0	0
[3]	)	0	0	0	0	0	0	0	0	0	0
[4]	+	0	0	0	0	0	0	0	0	0	0
[5]	*	0	0	0	0	0	0	0	0	0	0
[6]	|	0	0	0	0	0	0	0	0	0	0
[7]	ΔE	0	0	0	0	0	0	1	0	1	1
[8]	ΔG	0	0	0	0	0	0	1	0	0	1
[9]	ΔP	1	1	0	0	0	0	0	0	0	0
[10]	ΔT	1	1	0	0	0	0	0	0	1	1

[1]	X	0	0	0	0	0	0	0	0	0	0
[2]	(	0	0	0	0	0	0	0	0	0	0
[3]	)	0	0	0	0	0	0	0	0	0	0
[4]	+	0	0	0	0	0	0	0	0	0	0
[5]	*	0	0	0	0	0	0	0	0	0	0
[6]	|	0	0	0	0	0	0	0	0	0	0
[7]	ΔE	1	1	0	0	0	0	1	0	1	1
[8]	ΔG	0	0	0	0	0	0	1	0	1	1
[9]	ΔP	1	1	0	0	0	0	0	0	0	0
[10]	ΔT	1	1	0	0	0	0	0	0	1	1

$r_5 = r_4$ et voici le résultat de r^*

[1]	X	0	0	0	0	0	0	0	0	0	0
[2]	(	0	0	0	0	0	0	0	0	0	0
[3]	)	0	0	0	0	0	0	0	0	0	0
[4]	+	0	0	0	0	0	0	0	0	0	0
[5]	*	0	0	0	0	0	0	0	0	0	0
[6]	|	0	0	0	0	0	0	0	0	0	0
[7]	ΔE	1	1	0	0	0	0	1	0	1	1
[8]	ΔG	1	1	0	0	0	0	1	0	1	1
[9]	ΔP	1	1	0	0	0	0	0	0	0	0
[10]	ΔT	1	1	0	0	0	0	0	0	1	1

.....TABLE DES DERIVATIONS....

X	1	0	0	0	0	0	0	0	0	0
(	0	1	0	0	0	0	0	0	0	0
)	0	0	1	0	0	0	0	0	0	0
+	0	0	0	1	0	0	0	0	0	0
*	0	0	0	0	1	0	0	0	0	0
|	0	0	0	0	0	1	0	0	0	0
ΔE	1	1	0	0	0	0	1	0	1	1
ΔG	1	1	0	0	0	0	1	1	1	1
ΔP	1	1	0	0	0	0	0	0	1	0
ΔT	1	1	0	0	0	0	0	0	1	1

TABLE F11 *

Etant donnée une forme sententielle $\varphi_{\omega\tau}$, et $(A,\omega) \in P$ $(\varphi A\tau, \varphi\omega\tau)$ est une réduction canonique si $\varphi A\tau$ est une forme sententielle canonique. Notons $F(\varphi,A,\tau)$ une fonction logique vraie, si et seulement si $\varphi A\tau$ est une forme sententielle canonique. En général, le domaine de définition de F est infini.

En conséquence, pour pouvoir tabuler une fonction de décision, nous nous limiterons au dernier caractère de φ et au premier de τ. Cette fonction $F11(-1\uparrow\varphi, A, 1\uparrow\tau)$ sera vraie s'il existe φ et τ tels que $F(\varphi,A,\tau)$ soit vrai. Pour tabuler F, on génère toutes les formes sententielles canoniques de la grammaire, par dérivation. On part de l'axiome et à chaque pas on applique toutes les productions possibles au non-terminal le plus à droite. On enregistre φ,A,τ à chaque pas et ainsi on construit une table (φ,A,τ) telle que $F(\varphi,A,\tau)$ soit vrai. Si la grammaire est récursive, le domaine de F est infini et l'algorithme ne se termine jamais.

* Note $A \in V_N$ mais ce n'est pas l'axiome

Mais F11 ayant un domaine fini, on peut modifier cet algorithme de façon à construire la table pour F11 .

Par définition d'une grammaire de Chomsky, les formes sententielles dérivées d'un non-terminal sont indépendantes de contexte. Donc, si un triplet est déjà enregistré dans la table, toutes les formes canoniques qui en sont issues ont déjà été analysées. On passe alors à une autre règle. Cependant pour accélérer le processus, tout en obtenant tous les triplets valides, quand le premier caractère de τ est un non-terminal, on le remplace par les terminaux de tête des formes sententielles dérivées. Ceci à l'aide de la table HDTB précédemment calculée.

On applique la fonction récursive PRODUCTION à truc, axion, truc.

PRODUCTION(σ,ϕ,Ψ)

Tant que ϕ n'est pas vide faire

- Si $\bar{}_1\uparrow\phi \in V_T$ alors dépacer $\bar{}_1\uparrow\phi$ vers le début de Ψ
- Sinon
  - déplacer $\bar{}_1\uparrow\phi$ vers A
  - Pour chaque $\beta \in$ HDTB[$1\uparrow\Psi$] faire
    - Si ($\bar{}_1\uparrow\phi$, A , β) n'est pas enregistré alors l'enregistrer et positionner NEW à 1
  - Si NEW alors pour chaque w tq (A,w) $\in$ P PRODUCE($\sigma\phi$, w , Ψ)
  - mettre A au début de Ψ .

Obtention de la fonction $\mathcal{C}1$ à partir de la fonction F .

$\mathcal{C}1(\sigma,\tau)$ est la fonction de décision qui permet d'empiler le premier caractère de τ dans la pile σ . Si l'on prend la décision de ne pas empiler, cela signifie que le sommet de la pile σ contient la partie droite d'une règle de production de la grammaire. C'est la fonction $\mathcal{C}2$ qui trouve cette règle et applique la réduction (subtitution dans la pile de la partie gauche à la partie droite de la règle en question).

Condition de non-empilement.

Soit une forme sententielle canonique dérivable de l'axiome

$$(\varphi,N,\alpha) \in F \quad \text{par définition de} \quad F$$

et soit la règle $(A,\omega) \in P$

alors pour $(\varphi\omega,\alpha)$ la valeur de $\mathcal{C}_1$ est bien sûr : ne pas empiler car la réduction est immédiate. Pour $\mathcal{C}_1$, si l'on suppose que la phrase à analyser est correcte, alors

$$\mathcal{C}_1(\varphi\omega,\alpha) = \begin{cases} 0 & \text{si } \exists N \text{ tq } (\varphi,N,\alpha) \in F \text{ et } (N,\omega) \in P \\ 1 & \text{sinon} \end{cases}$$

Mais si on limite le contexte pour prendre la décision, il faut aussi analyser la condition d'empilement. En effet, avec un contexte limité, il peut apparaître pour certaines formes sententielles la décision d'empiler et pour d'autres, bien qu'ayant le même contexte, la décision de ne pas empiler. En cas de conflit, il y a ambiguïté locale, et l'on doit élargir le contexte pour essayer de lever cette ambiguïté. Dans notre système, si le contexte $(1,1)$ ne suffit pas, on essaye le contexte $(2,1)$. Si l'ambiguïté subsiste, on refuse la grammaire. L'utilisateur peut alors essayer de la modifier pour supprimer les ambiguïtés locales qui lui sont signalées.

Quand doit-on empiler ?

Pour $(\varphi,N,\alpha) \in F$ et $(N,\rho\Psi) \in P$ $\quad \Psi \in V^*$ et $\alpha,\tau \in V_T^*$

	Pile		texte	
la réduction se fait	φ	N	α	on va empiler pour aboutir
dans la pile	φ	$\rho\phi$	α	à cette réduction
	φ	ρ	τ^*	

Si $\phi = \emptyset$ $\quad \alpha \overset{*}{\rightarrow} \tau \approx \alpha = \tau$ car $\alpha \in V_T^*$

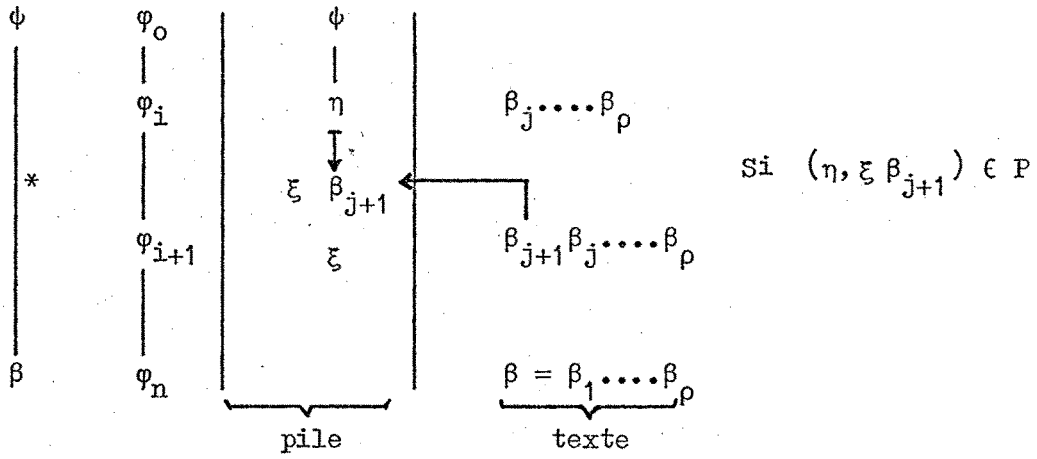
donc le schéma se réduit à

φ	N	α	C'est le cas de non-empilement
φ	ρ	α	$\mathcal{C}_1(\varphi\rho,\tau) \leftarrow 0$

Si $\psi \neq \phi$ alors $\tau = \beta\alpha$ et $\psi \overset{*}{\rightarrow} \beta$ donc $\exists(\varphi_i)_{i\in 0,n}$ tq

$$\varphi_o = \psi \rightarrow \varphi_1 \rightarrow \varphi_i \rightarrow \varphi_n = \beta \ .$$

dérivation canonique



Donc en plaçant pas à pas les caractères de β dans la pile on peut, par réductio successives, remonter à ψ . Alors dans ce cas $\mathcal{C}_1(\varphi\rho,\tau) \leftarrow 1$ car en empilant on va aboutir à la réduction $(N,\rho\psi)$.

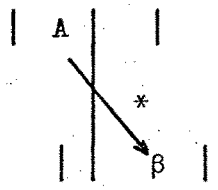
<u>En résumé</u>. Pour $(\varphi\rho,\tau)$ si $\exists N \in V_N$ tq $(\varphi,N,\alpha) \in F$ et $(N,\rho\psi) \in P$ et $\psi\alpha \overset{*}{\rightarrow} \tau$ alors

$$\mathcal{C}_1(\varphi\rho,\tau) \leftarrow \begin{cases} 0 & \text{si } \psi = \phi \\ 1 & \text{si } \psi \neq \phi \end{cases}$$

Algorithme pour $\mathcal{C}_{11}$ et TRIPLE.

La formulation précédente permet de donner l'algorithme pour les contextes limités ($c = 1$ pour C11 , 2 pour TRIPLE)

<u>Rq</u> : Dans le schéma $N \rightarrow \rho\psi \overset{*}{\rightarrow} \tau$, dans ces différentes formes sententielles tous les non-terminaux sauf l'axiome vont apparaître. Ainsi on aura examiné $(N \overset{*}{\rightarrow} \tau)$ pour $N \in V_N - \{A\}$. Cependant



doit être considéré. On ajoute donc à l'algorithme

$$\mathcal{C}_{11}(truc,\beta) \leftarrow 1 \quad \forall\beta \in V_T \cap HDTB(A) \ .$$

Pour chaque règle $(N,\omega) \in P$ faire :

  Pour chaque (φ,α) tq $(\varphi,N,\alpha) \in F11$ faire :

    Pour chaque ρ,ψ tq $\rho\psi = \omega$ et $\rho \neq \phi$ faire :

      Pour chaque $\beta \in V_T \cap HDTB(1|\psi\alpha)$ faire :

        $C1c(\bar{c}\uparrow\varphi\rho;\beta)$ — $(\psi \neq \phi)$ *

Pour chaque $\beta \in V_T \cap HDTB(A)$ faire :

  $C11(truc,\beta) \leftarrow 1$

$c = 1 \rightarrow C11$ et $c = 2 \rightarrow$ TRIPLE.

Rq : l'affectation * ne se fait pas ainsi

$\mathcal{C}11$ est divisé en deux tables l'une pour vrai, l'autre pour faux et suivant $(\psi \neq \phi)$ on met à un l'une ou l'autre. Si pour une position les deux tables sont à zéro alors la paire est invalide, si l'une des deux est à un alors 1 a la valeur de celle qui est à un. Si les deux sont à un alors il y a conflit.

Fonction $\mathcal{C}2$: sélection des règles.

Les règles sont groupées par dernier symbole identique et sont classées par longueur. La fonction $\mathcal{C}2$ consiste à trouver dans le groupe de règles, dont le dernier symbole est identique au sommet de la pile, la règle qui s'applique. Pour cela après avoir trouvé la règle la plus longue dont la partie droite est dans la pile, elle examine la table indiquant le type de contexte à tester. Suivant le résultat du test du contexte par rapport aux tables de contextes valides, elle applique la réduction ou cherche une autre règle à appliquer. Dans notre cas le contexte est $(1,1)$. Pour $\mathcal{C}2$ il faut donc :

premièrement ordonner les règles de production

deuxièmement établir la table indiquant le contexte de chaque règle : CONTEXT

puis les tables des contextes valides

- droit on utilise C11
- gauche LC
- droit et gauche LRC .

Ayant trouvé la plus longue règle applicable, parmi les règles du même groupe, y en a-t-il une autre qui le soit. Plusieurs cas peuvent se présenter.

1) Les parties droites des règles suivantes ne sont pas identiques à la fin de la règle en question

$$N \to N_1 \dots N_p$$

$\forall M$ tq $M_q = N_p$ et $p \geqslant q$ $M \to M_1 \dots M_q$ on a $M_{q-1} \neq N_{p-1}$ ou $\dots M_{q-i} \neq N_{p-i} \dots$ ou $M_1 \neq N_{p-q+1}$.

2) Il existe une règle dont la partie droite se confond avec la fin de la règle testée,

donc $\exists M$ tq $N \to N_1 - N_i\ M_1 - M_q$

$M \to M_1 - M_q$

3) Les parties droites de deux règles sont identiques

$N \to N_1 \dots N_p$

$M \to N_1 \dots N_p$.

Dans le 1er cas la règle trouvée est la bonne et aucun teste de contexte n'est nécessaire puisque c'est la seule applicable.

Cas 2 : F est une table qui pour $N \in V_N$ donne les contestes (1,1) acceptables pour N : $\{(\varphi,\alpha)/(\varphi,N,\alpha) \in F\}$ $\varphi \in V$ et $\alpha \in V_T$. Soit $(N,\phi\omega)$ et $(M,\omega) \in P$ et $\phi \neq \bar{\phi}$ si pour chaque $\varphi \in \{\varphi / (\varphi,M,\alpha) \in F\}$ $-1\uparrow\phi \neq \varphi$ alors $\phi\omega$ ne peut jamais être le haut de pile quand (M,ω) est la règle à appliquer, donc dans ce cas la règle la plus longue applicable est la bonne. Si $(N,\phi\omega)$ s'applique ce n'est pas la peine de regarder (M,ω) .

S'il existe un φ tq $\bar{1}\uparrow\phi = \varphi$ alors le test du contexte droit permet de prendre la décision si $\{\alpha/(\varphi,M,\alpha) \in F\}$ est disjoint de $\{\alpha'/(\varphi',N,\alpha') \in F\}$. Sinon on teste les deux contextes à la fois c'est-à-dire si $\{(\alpha,\varphi)/(\varphi,M,\alpha) \in F\}$ est disjoint de $\{(\alpha',\varphi')/(\varphi',N,\alpha') \in F\}$. Si ce n'est pas le cas, alors la décision ne peut être prise dans le contexte (1,1) et la grammaire est refusée.

Cas 3 : (N,ω) s'applique et il existe une règle (M,ω) le principe est le même. Si $\{\varphi/(\varphi,M,\alpha) \in F\}$ est disjoint de $\{\varphi'/(\varphi',N,\alpha') \in F\}$ alors en testant le contexte gauche on pourra prendre la décision. Sinon comme dans le cas 2, on essaiera le contexte droit, puis les deux contextes à la fois. En cas d'échec, la grammaire est refusée.

Algorithme

Classer les règles puis, pour chaque règle, examiner les règles du même groupe qui la suivent. Suivant les cas, déterminer le contexte à tester de la façon indiquée. De cette façon, on construit CONTEXT et les tables LC et LRC sont les extraits utiles de F11 .

Forme d'entrée des règles de syntaxe.

Les règles suivantes, sous forme de Backus sont transformées en

<A> := <B> | C<D> | E + <F>

ΔA ΔB

ΔA C ΔD

ΔA E + ΔF .

On remarque :

les alternances sont introduites une à une

les nonterminaux commencent par 'Δ' .

Restrictions :

ne pas utiliser les caractères suivants :

'|' qui correspond à truc (reste en interne)

'.' qualifieur } qui sont utilisées pour l'introduction des

'..' seconde occurence } règles sémantiques

(en cas de nécessité ces symboles peuvent être changés).

Un même non-terminal ne peut apparaître plus de deux fois dans la même règle.

EXEMPLE SIMPLE

introduction des règles

ligne vide, fin


```
                    ANALYSE

....ENTREZ LES REGLES....

 : ΔA  ΔN
 : ΔN  ΔL , ΔL
 : ΔN  ΔL
 : ΔL  ΔL   ΔB
 : ΔL   ΔB
 : ΔB  NB
 :

GRAMMAIRE LUE

*****SORT*****

****  DICTIONNAIRE    *******

[1] NB
[2] |
[3] ,
[4] ΔA
[5] ΔB
[6] ΔL
[7] ΔN

****  PRODUCTIONS     *******

   1        ΔA   ::= ΔN
   2        ΔB   ::= NB
   3        ΔL   ::= ΔL  ΔB
   4        ΔL   ::= ΔB
   5        ΔN   ::= ΔL  ,  ΔL
   6        ΔN   ::= ΔL
```

partie gauche ordonnée suivant l'ordre du dictionnaire

```
.....TABLE DES DERIVATIONS.....

N B   1 0 0 0 0 0 0
|     0 1 0 0 0 0 0
,     0 0 1 0 0 0 0
ΔA    1 0 0 1 1 1 1
ΔB    1 0 0 0 1 0 0
ΔL    1 0 0 0 1 1 0
ΔN    1 0 0 0 1 1 1
```

HDTB →

TABLE F11

des contextes valides

des nonterminaux

```
.... TABLE F11 ....

|       ΔA      |
ΔL      ΔB      |
ΔL      ΔB      NB
ΔL      ΔB      ,
|       ΔB      NB
|       ΔB      ,
,       ΔB      NB
,       ΔB      |
|       ΔB      |
,       ΔL      |
,       ΔL      NB
|       ΔL      ,
|       ΔL      NB
|       ΔL      |
|       ΔN      |
```

SORT PRODUCTIONS

**** *PRODUCTIONS* *******

Productions rangées par

groupe suivant le dernier

terme

```
1       ΔA  ::= ΔN
2       ΔN  ::= ΔL , ΔL
3       ΔN  ::= ΔL
4       ΔL  ::= ΔL ΔB
5       ΔL  ::= ΔB
6       ΔB  ::= NB
```

TABLE C11 ****

Table de décision $\mathfrak{C}_{11}$

il n'y a pas de conflit

donc pas besoin de TRIPLE

```
NB  FFF____
|   V______
,   V______
ΔA  _F_____
ΔB  FFF____
ΔL  VFV____
ΔN  _F_____
```

type de contextes à tester

suivant la règle

2 : signifie gauche

```
.....CONTEXTE.....

0 2 0 2 0 0
```

table des contextes

gauche valides

```
.. GAUCHE ...

|       ΔL
,       ΔL
|       ΔN
```

Table des contextes droit et

gauche vide

..GAUCHE ET DROIT...

Introduction des symboles choisis

comme pseudo-terminaux

PSEUDO TERMINAUX DE LA GRAMMAIRE
POUR LES IDENTIFICATEURS

POUR LES NOMBRES
NB
POUR LES CHAINES DE CARACTERES

```
****  PRODUCTIONS     *******

  1       ΔSTAT  ::= ΔAS
  2       ΔSTAT  ::= ΔIF THEN ΔSTAT
  3       ΔSTAT  ::= ΔIF THEN ΔSTAT ELSE ΔSTAT

......TABLE DES DERIVATION......

ELSE   1 0 0 0 0 0
THEN   0 1 0 0 0 0
|      0 0 1 0 0 0
ΔAS    0 0 0 1 0 0
ΔIF    0 0 0 0 1 0
ΔSTAT  0 0 0 1 1 1

 .... TABLE F11 ....

    |          ΔAS        |
    THEN       ΔAS        |
    ELSE       ΔAS        |
    THEN       ΔAS        ELSE
    ELSE       ΔAS        ELSE
    |          ΔIF        THEN
    THEN       ΔIF        THEN
    ELSE       ΔIF        THEN
    |          ΔSTAT      |
    THEN       ΔSTAT      |
    ELSE       ΔSTAT      |
    THEN       ΔSTAT      ELSE
    ELSE       ΔSTAT      ELSE

 SORT PRODUCTIONS


****  PRODUCTIONS     *******

  1       ΔSTAT  ::= ΔIF THEN ΔSTAT ELSE ΔSTAT
  2       ΔSTAT  ::= ΔIF THEN ΔSTAT
  3       ΔSTAT  ::= ΔAS

**  TABLE C11  ****

ELSE   ------
THEN   ------
|      ------
ΔAS    F_F___
ΔIF    _V____
ΔSTAT  C_F___   <-- conflit
ATTENTION : TRIPLE EN CONFLIT
    THEN       ΔSTAT      ELSE

7 CONFLICTS  ......

**  TRIPLES  **

    THEN       ΔSTAT      ELSE       I  } conflits
    ELSE       ΔSTAT      ELSE       I
```

EXEMPLE de grammaire non acceptée car ambiguë

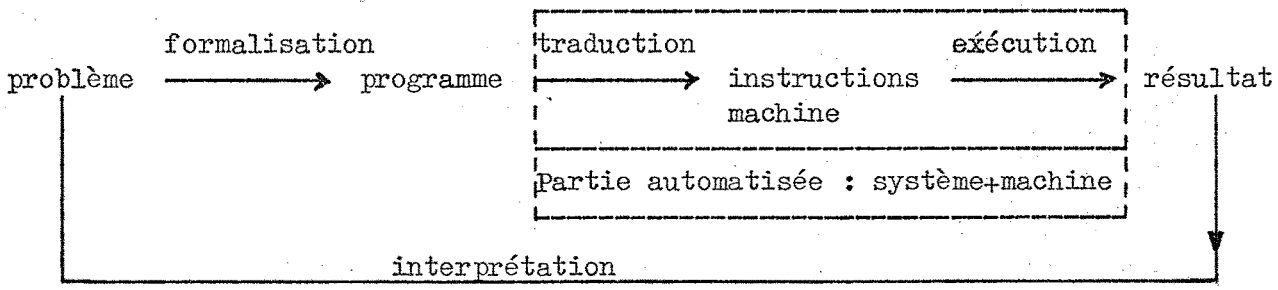
ex : ΔIF then ΔIF then ΔAS else ΔAS peut se dériver de deux façons différentes de l'axiome

L'ANALYSE SEMANTIQUE

L'objet de tout traitement informatique est l'obtention d'un certain résultat. Pour définir ce traitement, on utilise des langes artificielles, plus ou moins évoluées et plus ou moins adaptées au problème à résoudre.

Notre système permet de définir un langage exactement adapté à nos besoins. Bien sûr, les phrases de ce langage sont transformées en commandes machine. La machine exécutant ces commandes permet d'obtenir le résultat désiré.

Ce que l'on peut représenter ainsi



Pour le traitement automatique, il est nécessaire de définir un langage dans lequel sont formalisés les problèmes. Ce langage est défini par une syntaxe et une sémantique, que l'utilisateur doit connaître, s'il veut que les programmes qu'il écrit fournissent les résultats attendus.

D'autre part, il faut que le système traduction-exécution n'altère pas la signification du programme qu'il doit traiter.

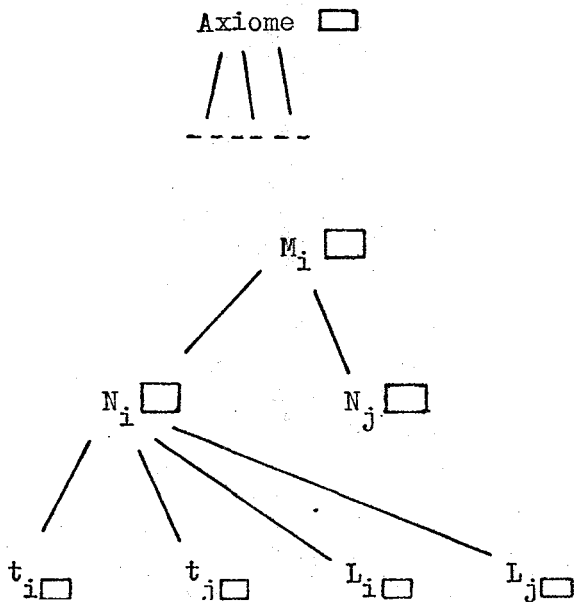
Le système doit donc accepter exactement le langage que l'on a défini.

Du point de vue de la syntaxe, on sait assez bien réaliser cette exigence. La définition sous forme de Backus, par exemple, permet à l'utilisateur d'écrire facilement des programmes syntaxiquement corrects. Elle permet aussi, nous l'avons vu, de construire systématiquement des analyseurs syntaxiques. Leur génération automatique permet d'obtenir une certaine fiabilité et une grande souplesse (modifiabilité) pour les produits obtenus.

Pour la sémantique, le problème qui se pose est le même. Mais là, les solutions sont plus empiriques. La définition sémantique de l'Algol, par exemple, est

faite en langage naturel. L'utilisateur voit sa tâche facilitée, dans un sens. Mais l'implantation de l'analyseur sémantique est plus difficile. Différentes interprétations sont possibles et des déviations sémantiques apparaissent suivant les systèmes.

D'autre part, des méthodes théoriques de définition de la sémantique des langages formels se développent, mais la liaison entre les définitions et leur mise en oeuvre, pour traiter l'aspect sémantique du langage n'est pas clairement définie. La méthode des attributs sémantiques de Knuth, a été choisie parce qu'elle permet une définition simple et précise de la sémantique (semblable à la forme de Backus). Mais, de plus, elle permet de construire systématiquement, à partir de ces définitions, l'analyseur sémantique correspondant. Elle satisfait donc aux objectifs que nous nous sommes fixés. Schématiquement, l'analyse sémantique consiste à reconstituer la signification globale d'une phrase. La signification des terminaux est considérée comme une donnée primitive. A la structure de la phrase est associée une série de règles de déduction de la sémantique globale à partir de la signification des terminaux. Les méthodes empiriques consistent à parcourir l'arbre syntaxique et suivant les cas à générer un code ou à positionner un indicateur. La méthode des attributs se caractérise par la localisation de la sémantique qui est portée par les non-terminaux.



les carrés représentent l'ensemble des attributs sémantiques du noeud.

La sémantique de N_i peut dépendre de la sémantique des non-terminaux M_i (père) et L_i , L_j (fils) et des terminaux t_i et t_j .

On remonte ainsi dans l'arbre (qui symbolise la structure de la phrase) les significations primitives portées par les terminaux, jusqu'à l'axiome qui porte la signification globale.

Donc localisation de la sémantique, par des attributs associés aux non-terminaux. Chaque attribut a une signification logique propre. Par exemple, attribut de longueur, de type ...

Ces attributs se séparent en deux classes disjointes :

- hérités si le calcul va des pères aux fils
- synthétisés si le calcul va des fils vers le père. Ainsi, la sémantique d'un non-terminal est définie par un ensemble d'attributs ayant une signification bien précise. La méthode est dirigée par la syntaxe. Mais elle est malgré tout déclarative, c'est-à-dire, que seule la structure est prise en compte, elle est indépendante de l'analyse syntaxique utilisée pour obtenir cette structure.

Cette méthode est très souple, il suffit de choisir judicieusement les attributs pour obtenir un interpréteur, un compilateur à un ou plusieurs passages (attributs de synchronisation) ou bien encore, comme l'application en a été faite, un système qui teste si une expression appartient à un certain système de calcul et qui, si oui, fournit l'expression sous forme normale et son type.

Après cette introduction nous allons présenter un exemple simple, puis une définition plus précise de la méthode et ensuite montrer la réalisation de l'analyseur sémantique.

Exemple simple.

Il s'agit d'un interpréteur qui donne la valeur décimale d'un nombre sous forme binaire. La syntaxe de cette représentation a déjà été définie dans les chapitres relatifs à la syntaxe.

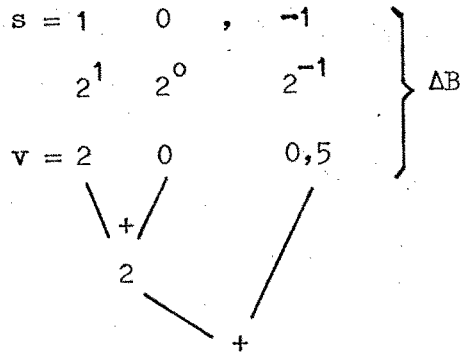
La définition sémantique peut se faire à l'aide de trois attributs :

v : valeur portée par les non-terminaux ΔA , ΔB , ΔL , ΔN

ℓ : longueur de la suite de bits ΔL

s : attribut hérité, il donne pour ΔB le poids du bit associé et pour ΔL le poids du ΔB qui lui est attaché

ex : 1 0 1



v(ΔA) = 2,5

Rq sur le calcul de s

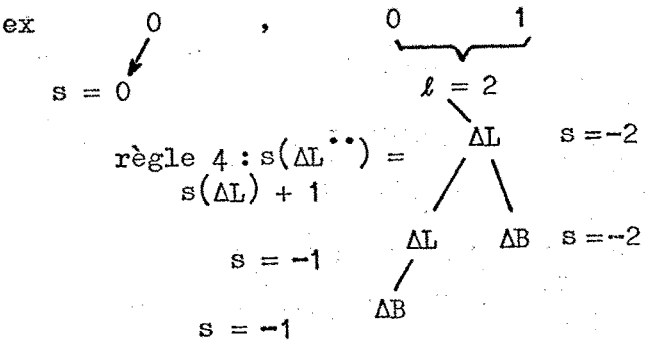
à la règle 2

s(Δℓ) donne le poids du terminal le plus à droite

donc s(ΔL) ← 0

et s(ΔL") ← - ℓ(ΔL")

ex 0 , 0 1



***** *POUR LA REGLE* *****
1 ΔA ::= ΔN
∘∘∘∘ *LA SEMANTIQUE INTRODUITE EST*
V.ΔA ← V.ΔN

***** *POUR LA REGLE* *****
2 ΔN ::= ΔL , ΔL
∘∘∘∘ *LA SEMANTIQUE INTRODUITE EST*
V.ΔN ← V.ΔL + V.ΔL"
S.ΔL ← 0
S.ΔL" ← - L.ΔL"

***** *POUR LA REGLE* *****
3 ΔN ::= ΔL
∘∘∘∘ *LA SEMANTIQUE INTRODUITE EST*
V.ΔN ← V.ΔL
S.ΔL ← 0

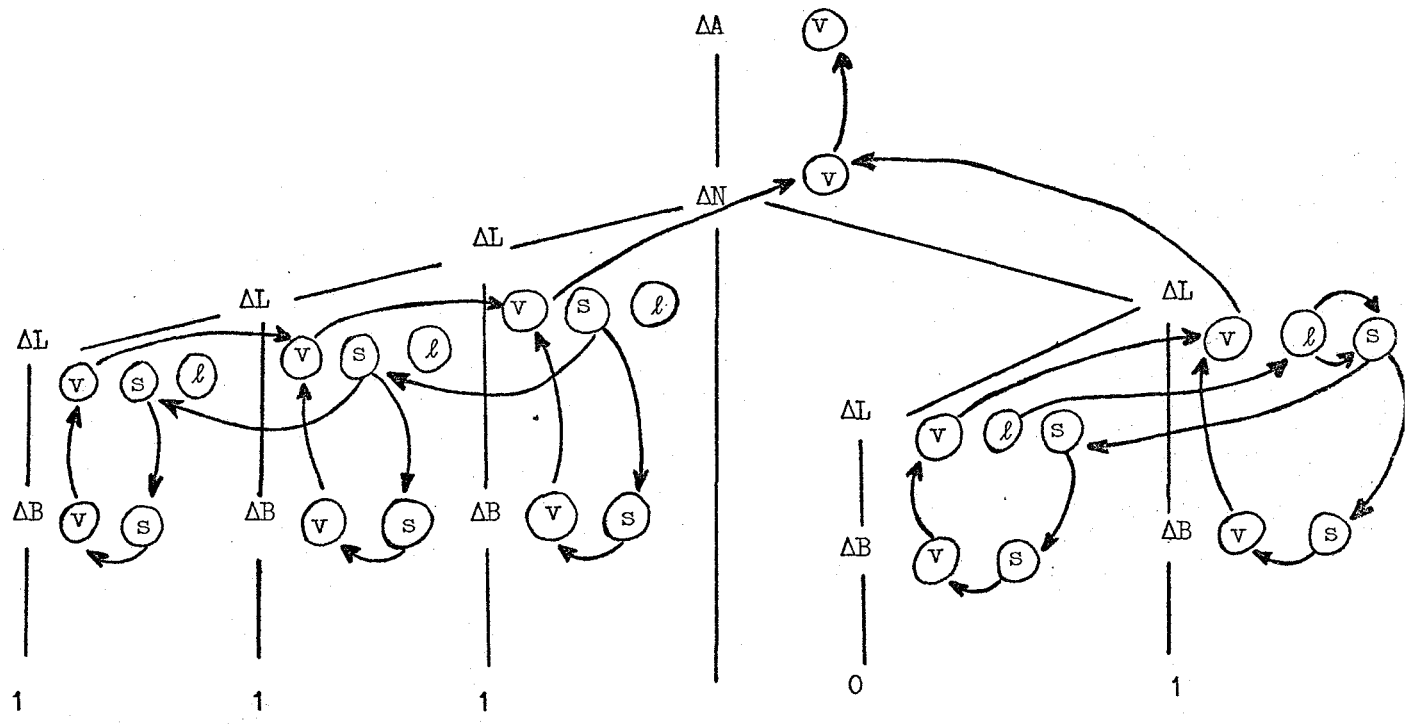
***** *POUR LA REGLE* *****
4 ΔL ::= ΔL ΔB
∘∘∘∘ *LA SEMANTIQUE INTRODUITE EST*
V.ΔL ← V.ΔL" + V.ΔB
S.ΔB ← S.ΔL
S.ΔL" ← S.ΔL + 1
L.ΔL ← L.ΔL" + 1

***** *POUR LA REGLE* *****
5 ΔL ::= ΔB
∘∘∘∘ *LA SEMANTIQUE INTRODUITE EST*
V.ΔL ← V.ΔB
S.ΔB ← S.ΔL
L.ΔL ← 1

***** *POUR LA REGLE* *****
6 ΔB ::= NB
∘∘∘∘ *LA SEMANTIQUE INTRODUITE EST*
*V.ΔB ← (2 * S.ΔB) × NOMBRE*

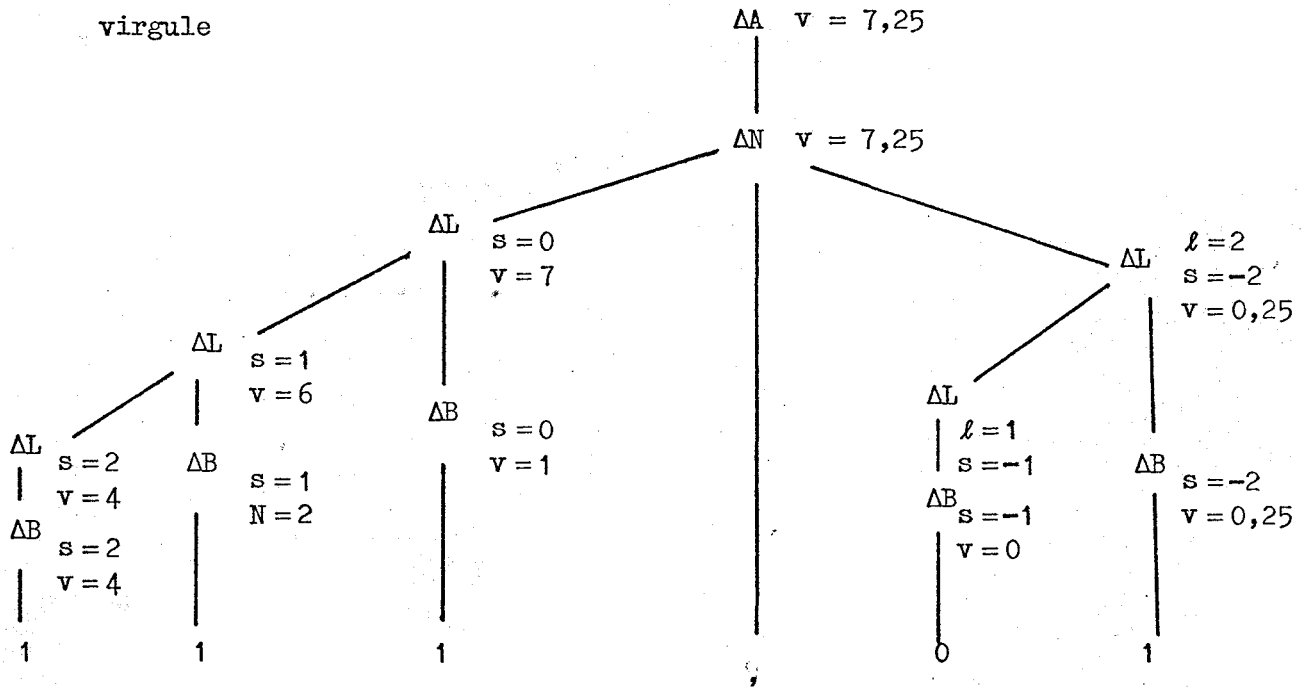
Structure correspondant à '1 1 1 , 0 1'

1) Ordre de calcul



2) Calcul effectif.

On remarque qu'il est inutile de calculer ℓ pour les ΔL à gauche de la virgule



Le processus de l'analyse sémantique se déroule donc ainsi :

1) à partir de l'arbre syntaxique et des règles sémantiques associées aux règles de syntaxe, construire les relations de dépendance entre attributs.

2) A l'aide d'un tri topologique, obtenir l'ordre de calcul des attributs.

3) Appliquer pour chaque attribut, la fonction sémantique qui permet son évaluation.

Définition des attributs.

Soit donc une grammaire de Chomsky $G = \langle V_T, V_N, A, P \rangle$ vérifiant les contraintes définies pour être acceptables par l'analyseur syntaxique. Les règles sémantiques lui sont adjointes de la façon suivante : à chaque non-terminal $X \in V_N$ on associe un ensemble fini d'attributs $A(X)$ qui est composé de deux sous-ensembles disjoints $H(X)$ attributs hérités et $S(X)$ attributs synthétisés. On a évidemment $H(A)$ vide (l'axiome n'a pas d'attributs hérités).

Pour chaque règle syntaxique $X_o \rightarrow X_1 \text{ --- } X_i \text{ --- } X_p$ une fonction sémantique permet d'évaluer chaque attribut.

. Un attribut synthétisé de X_o est calculé en fonction des attributs des X_i ($i \in 1$ à p) et des autres attributs de X_o.

. Un attribut hérité d'un des X_j est calculé en fonction des attributs des X_i ($i \in 0$ à p et $i \neq j$) et des autres attributs de X_j.

Dans l'exemple précédent à la règle 4 $\Delta L := \Delta L \ \Delta B$ sont associées 4 fonctions sémantiques. La 1ère et la 4ème permettent de calculer les attributs synthétisés v et ℓ de ΔL et les deux autres permettent de calculer l'attribut hérité s' pour ΔB et ΔL" (seconde occurrence de ΔL).

Remarques.

1. Pour les terminaux on ne définit pas d'attributs, mais des fonctions particulières permettent d'atteindre leur signification primitive. Dans l'exemple, la fonction NOMBRE donne la valeur du terminal NB qui est portée par la partie ΔVAL de l'arbre syntaxique.

2. On peut très bien se passer des attributs hérités, car théoriquement cette notion n'apporte rien de plus, mais pratiquement, elle apporte de la souplesse et facilite la définition logique de la sémantique.

3. Il faut bien entendu que la définition des attributs soit cohérente. Pour tout arbre, tous les attributs doivent être calculables, c'est-à-dire :

ils doivent être définis une et une seule fois et dans le graphe de dépendance des attributs, il ne doit pas y avoir de boucle.

La définition des attributs est l'objet du paragraphe relatif au constructeur sémantique, où ces conditions seront détaillées.

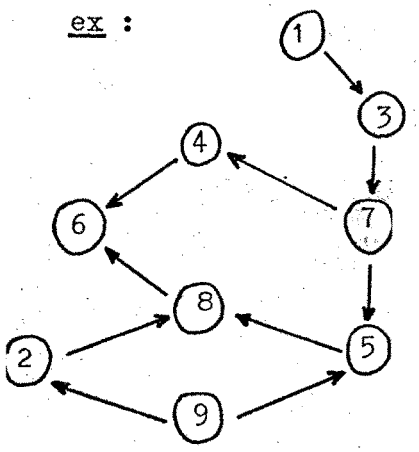
Evaluation sémantique.

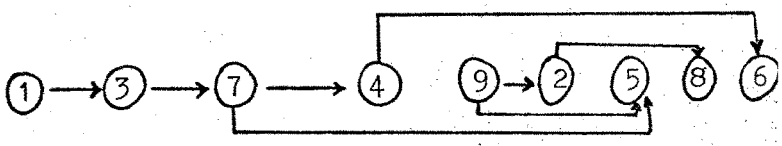
Le constructeur sémantique donne, pour la règle syntaxique i, une fonction ΔS_i qui donne les relations de dépendance des attributs associés à cette règle. L'analyseur sémantique, en balayant l'arbre, exécute pour chaque noeud la fonction ΔS_i correspondant à la règle appliquée. Cette opération dite de décoration de l'arbre syntaxique, permet d'obtenir un ensemble de relation de dépendance entre attributs. Cet ensemble est une forme de représentation de l'arbre des dépendances.

Tri topologique.

Soit un ensemble fini muni d'une relation d'ordre partiel (par exemple : a doit être calculé avant b). Les propriétés de l'ordre partiel impliquent qu'il ne peut pas y avoir de boucle dans le graphe de dépendance. Dans ces conditions, le tri topologique est l'opération qui dégage de l'ordre partiel un ordre linéaire (c'est-à-dire une suite telle que a_i doit être calculé avant a_j si $i < j$)

ex :





L'algorithme utilisé permet d'éliminer les impasses (les calculs des attribut qui ne contribuent pas à l'obtention des attributs de l'axiome)

exemple : dans l'exemple des nombres binaires, les ΔL à gauche de la virgule sont inutiles.

Pour cela, on part des attributs de l'axiome et on regarde de quoi ils dépendent et ainsi de suite ...

Algorithme :

$$\forall a \in S(A) \ , \ RP(a)$$

RP(x) : [$\forall$ z tq z $\in$ {y/y calculé avant x} si z $\notin$ liste alors RP(z)

liste $\leftarrow$ liste , z] .

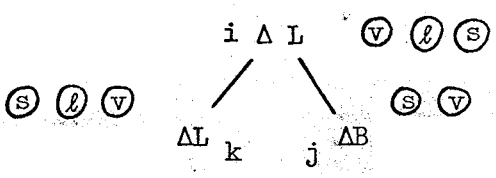
C'est la procédure récursive RP qui établit la liste ordonnée des attributs à calculer.

Evaluation de la valeur d'un attribut.

Un attribut est caractérisé par son nom et par le noeud auquel il est attaché

Les règles sémantiques sont locales et dépendent donc du contexte (noeud courant, père, fils) elles sont traduites par le constructeur sémantique, en fonctions dont les arguments sont donnés par le contexte.

Par exemple, la règle 4



$v(\Delta L) \leftarrow v(\Delta L") + v(\Delta B)$

$\ell(\Delta L) \leftarrow \ell(\Delta L") + 1$

la fonction sémantique pour la règle 4, le 1er non-terminal et l'attribut v a pour nom* $\Delta 4 \Delta 1 \Delta 1$ et pour arguments i , j , k et pour texte $(i,v) \leftarrow (j,v) + (k,$ et la fonction de nom $\Delta 4 \Delta 1 \Delta 3$ et mêmes arguments

$$(i,\ell) \leftarrow (k,\ell) + 1 \ .$$

---

* nom de la fonction : $\Delta r \ \Delta p \ \Delta a$

r : n° de règle p : position du non-terminal dans la règle

a :position de l'attribut dans la liste des attributs (ici v , s , ℓ) .

LE CONSTRUCTEUR D'ANALYSEURS SEMANTIQUE

Les objectifs du constructeur sémantique sont les mêmes que ceux du constructeur syntaxique. Il permet de décrire la sémantique au niveau logique, de façon claire et précise. Il fournit automatiquement les données nécessaires à l'analyseur sémantique. Les seules fautes possibles sont donc au niveau le plus élevé, dans la cohérence de la définition sémantique.

Bien sûr, le langage sémantique doit être très riche et les relations entre attributs peuvent être complexes. Aussi, il faut adjoindre, au système une bibliothèque de procédures sémantiques.

Le constructeur lit les règles sémantiques et effectue certaines vérifications simples permettant de détecter des erreurs de frappe (ex. les attributs des non-terminaux doivent être de la forme suivante a.ΔX) . Il peut aller jusqu'à la vérification de l'unicité de définition des attributs et de la cohérence des définitions entre elles. Il peut donner la liste des procédures sémantiques utilisées, ce qui permet de vérifier leur existence et, le cas échéant, de détecter une erreur. Dans la version actuelle, ces tests de cohérence des définitions sémantiques qui font tout l'intérêt du système ne sont pas implantées. Mais ils peuvent être ajoutés très facilement, car ils sont basés sur la fermeture transitive d'une relation de dépendance sur un ensemble. Or ce programme existe dans le système. c'est la fonction CMPH du constructeur syntaxique.

Nous allons décrire la réalisation sous sa forme actuelle, des améliorations étant possibles dans plusieurs directions

- tests de cohérence des définitions
- bibliothèque standard de manipulation de structures de données (listes, arbres ...).

L'exemple utilisé sera celui des nombres binaires. Des exemples complets et plus intéressants sont donnés dans un chapitre spécial.

Et pour les règles $s(\Delta B) \leftarrow s(\Delta L)$

et $s(\Delta L'') \leftarrow s(\Delta L) + 1$

les fonctions sémantiques sont

$\Delta 4 \Delta 3 \Delta 2\ (i, j, k)$	$\Delta 4 \Delta 2 \Delta 2\ (i, j, k)$
$(k,s) \leftarrow (i,s)$	$(j,s) \leftarrow (i,s) + 1$.

Plus précisément :

```
    ∇ Z←Δ4Δ1Δ1 V
[1]   ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←',('ΔTAS',(⍕V[2]),'Δ',⍕1),'+',('ΔTAS
      ',(⍕V[3]),'Δ',⍕1)
    ∇
```

```
    ∇ Z←Δ4Δ3Δ2 V
[1]   ⍎('ΔTAS',(⍕V[3]),'Δ',⍕2),'←',('ΔTAS',(⍕V[1]),'Δ',⍕2)
    ∇
```

```
    ∇ Z←Δ4Δ2Δ2 V
[1]   ⍎('ΔTAS',(⍕V[2]),'Δ',⍕2),'←',('ΔTAS',(⍕V[1]),'Δ',⍕2),'+1'
    ∇
```

```
    ∇ Z←Δ4Δ1Δ3 V
[1]   ⍎('ΔTAS',(⍕V[1]),'Δ',⍕3),'←',('ΔTAS',(⍕V[2]),'Δ',⍕3),'+1'
    ∇
```

On peut remarquer la différence entre attributs hérités et synthétisés. Pour calculer les attributs synthétisés du noeud i on applique la règle 4 associée à ce noeud, avec comme arguments les attributs des noeuds fils. Pour l'attribut hérité s , du noeud j ou k on applique les règles sémantiques associées à la règle syntaxique associée au noeud père.

Forme d'entrée des règles sémantiques

L'attribut a du non-terminal ΔN est noté

a . ΔN

de ce fait, le point '.' ne peut être utilisé pour un autre usage.

Pour noter la seconde occurrence d'un non-terminal on lui accole '..'

Du fait de la réalisation en APL , les procédures sémantiques sont de deux formes

```
Z ← P1(ARG1)
Z ← (ARG1) P2(ARG2) . .
```

Ces procédures ont toutes un résultat explicite et pour bien les distinguer, les arguments sont mis entre parenthèses.

Comme pour la syntaxe, le symbole d'affectation est omis lors de l'introduction des données, mais il apparaît lors de l'édition.

Exemple des nombres binaires

Sans commentaires la partie construction syntaxique

```
....ENTREZ LES REGLES....

 : ΔA ΔN
 : ΔN ΔL,ΔL
 : ΔN ΔL
 : ΔL ΔL  ΔE
 : ΔL  ΔB
 : ΔB NB
 :

GRAMMAIRE LUE

*****SORT*****

****  DICTIONNAIRE   *******

[1] NB
[2] |
[3] ,
[4] ΔA
[5] ΔB
[6] ΔL
[7] ΔN

****  PRODUCTIONS    *******

  1       ΔA  ::= ΔN
  2       ΔB  ::= NB
  3       ΔL  ::= ΔL ΔB
  4       ΔL  ::= ΔB
  5       ΔN  ::= ΔL , ΔL
  6       ΔN  ::= ΔL

.....TABLE DES DERIVATIONS.....

N B  1 0 0 0 0 0 0
|    0 1 0 0 0 0 0
,    0 0 1 0 0 0 0
ΔA   1 0 0 1 1 1 1
ΔB   1 0 0 0 1 0 0
ΔL   1 0 0 0 1 1 0
ΔN   1 0 0 0 1 1 1

.... TABLE F11 ....

    |       ΔA      |
   ΔL       ΔB      |
   ΔL       ΔB      NB
   ΔL       ΔB      .
    |       ΔB      NB
    |       ΔB      ,

    .       ΔB      NB
    .       ΔB      |
    |       ΔB      |
    .       ΔL      |
    .       ΔL      NB
    |       ΔL      .
    |       ΔL      NB
    |       ΔL      |
    |       ΔN      |

 SORT PRODUCTIONS

****  PRODUCTIONS    *******

  1       ΔA  ::= ΔN
  2       ΔN  ::= ΔL , ΔL
  3       ΔN  ::= ΔL
  4       ΔL  ::= ΔL ΔB
  5       ΔL  ::= ΔB
  6       ΔB  ::= NB

    TABLE C11  ****

NB FFF____
|  V______
,  V______
ΔA _F_____
ΔB FFF____
ΔL VFV____
ΔN _F_____

.....CONTEXTE.....

0 2 0 2 0 0

.. GAUCHE ...

|       ΔL
,       ΔL
|       ΔN

..GAUCHE ET DROIT...

PSEUDO TERMINAUX DE LA GRAMMAIRE
POUR LES IDENTIFICATEURS

POUR LES NOMBRES
NB
POUR LES CHAINES DE CARACTERES
```

Ce que tape l'utilisateur est précédé de ":"

```
      SYNTHESE
.. POUR LA REGLE :
ΔA ::= ΔN
 ENTREZ LA REGLE SEMANTIQUE 1
 : V.ΔA V.ΔN
 ENTREZ LA REGLE SEMANTIQUE 2
 :
.. POUR LA REGLE :
ΔN ::= ΔL , ΔL
 ENTREZ LA REGLE SEMANTIQUE 1
 : V.ΔN V.ΔL+V.ΔL"
 ENTREZ LA REGLE SEMANTIQUE 2
 : S.ΔL 0
 ENTREZ LA REGLE SEMANTIQUE 3
 : S.ΔL" -L.ΔL"
 ENTREZ LA REGLE SEMANTIQUE 4
 :
.. POUR LA REGLE :
ΔN ::= ΔL
 ENTREZ LA REGLE SEMANTIQUE 1
 : V.ΔN V.ΔL
 ENTREZ LA REGLE SEMANTIQUE 2
 : S.ΔL 0
 ENTREZ LA REGLE SEMANTIQUE 3
 :
.. POUR LA REGLE :
ΔL ::= ΔL ΔB
 ENTREZ LA REGLE SEMANTIQUE 1
 : V.ΔL V.ΔL"+V.ΔB
 ENTREZ LA REGLE SEMANTIQUE 2
 : S.ΔB S.ΔL
 ENTREZ LA REGLE SEMANTIQUE 3
 : S.ΔL" S.ΔL+1
 ENTREZ LA REGLE SEMANTIQUE 4
 : L.ΔL L.ΔL"+1
 ENTREZ LA REGLE SEMANTIQUE 5
 :
.. POUR LA REGLE :
ΔL ::= ΔB
 ENTREZ LA REGLE SEMANTIQUE 1
 : V.ΔL V.ΔB
 ENTREZ LA REGLE SEMANTIQUE 2
  S.ΔB S.ΔL
 ENTREZ LA REGLE SEMANTIQUE 3
 : L.ΔL 1
 ENTREZ LA REGLE SEMANTIQUE 4
 :
.. POUR LA REGLE :
ΔB ::= NB
 ENTREZ LA REGLE SEMANTIQUE 1
 :  V.ΔB (2*S,ΔB)×NOMBRE

  ENTREZ LA REGLE SEMANTIQUE 2
  :
 +++ SEMANTIQUE LUE +++
```

ligne vide pour passer à la règle syntaxique suivante

les " après ΔL indiquent la seconde occurrence

procédure sémantique sans argument : nombre

```
LISTE DES ATTRIBUTS

V
S
L
...ATTRIBUTS DES NON TERMINAUX ...

ΔA          1 0 0
ΔB          1 1 0
ΔL          1 1 1
ΔN          1 0 0

***** POUR LA REGLE *****
  1              ΔA ::= ΔN
∘∘∘∘ LA SEMANTIQUE INTRODUITE EST :
V.ΔA ← V.ΔN
***** POUR LA REGLE *****
  2              ΔN ::= ΔL , ΔL
∘∘∘∘ LA SEMANTIQUE INTRODUITE EST :
V.ΔN ← V.ΔL + V.ΔL"
S.ΔL ← 0
S.ΔL" ← - L.ΔL"
***** POUR LA REGLE *****
  3              ΔN ::= ΔL
∘∘∘∘ LA SEMANTIQUE INTRODUITE EST :
V.ΔN ← V.ΔL
S.ΔL ← 0
***** POUR LA REGLE *****
  4              ΔL ::= ΔL ΔB
∘∘∘∘ LA SEMANTIQUE INTRODUITE EST :
V.ΔL ← V.ΔL" + V.ΔB
S.ΔB ← S.ΔL
S.ΔL" ← S.ΔL + 1
L.ΔL ← L.ΔL" + 1
***** POUR LA REGLE *****
  5              ΔL ::= ΔB
∘∘∘∘ LA SEMANTIQUE INTRODUITE EST :
V.ΔL ← V.ΔB
S.ΔB ← S.ΔL
L.ΔL ← 1
***** POUR LA REGLE *****
  6              ΔB ::= NB
∘∘∘∘ LA SEMANTIQUE INTRODUITE EST :
V.ΔB ← ( 2 * S.ΔB ) × NOMBRE

CONSTRUCTION DES ΔSI TERMINEE
CONSTRUCTION DES ΔI TERMINEE
LISTE DES ATTRIBUTS HERITEES
Π:
      2
```

pour vérifications

→ édition récapitulative de la définition du langage

les fonctions sémantiques sont construites, les voir page suivant

introduction des positions des attributs hérités dans la liste des attributs, ici : 2 → S

⍝ *LISTE DU GROUPE DE <ΔS1>*

LE 30/9/1975 *A* 11*H* 7*MN*

```
     ∇ Z←ΔS1 V
[1]    V←V×100
[2]    Z← 0 2 ⍴0
[3]    Z←Z,[1](V[2]+1),V[1]+1
     ∇
```

```
     ∇ Z←ΔS2 V
[1]    V←V×100
[2]    Z← 0 2 ⍴0
[3]    Z←Z,[1](V[2]+1),V[1]+1
[4]    Z←Z,[1](V[3]+1),V[1]+1
[5]    Z←Z,[1](V[3]+3),V[3]+2
     ∇
```

```
     ∇ Z←ΔS3 V
[1]    V←V×100
[2]    Z← 0 2 ⍴0
[3]    Z←Z,[1](V[2]+1),V[1]+1
     ∇
```

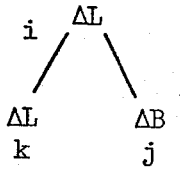
```
     ∇ Z←ΔS4 V
[1]    V←V×100
[2]    Z← 0 2 ⍴0
[3]    Z←Z,[1](V[2]+1),V[1]+1
[4]    Z←Z,[1](V[3]+1),V[1]+1
[5]    Z←Z,[1](V[1]+2),V[3]+2
[6]    Z←Z,[1](V[1]+2),V[2]+2
[7]    Z←Z,[1](V[2]+3),V[1]+3
     ∇
```

```
     ∇ Z←ΔS5 V
[1]    V←V×100
[2]    Z← 0 2 ⍴0
[3]    Z←Z,[1](V[2]+1),V[1]+1
[4]    Z←Z,[1](V[1]+2),V[2]+2
     ∇
```

```
     ∇ Z←ΔS6 V
[1]    V←V×100
[2]    Z← 0 2 ⍴0
[3]    Z←Z,[1](V[1]+2),V[1]+1
     ∇
```

Explication pour la règle 4

***** *POUR LA REGLE* *****
4 $\Delta L ::= \Delta L\ \Delta B$
∘∘∘∘ *LA SEMANTIQUE INTRODUITE EST :*
$V.\Delta L \leftarrow V.\Delta L'' + V.\Delta B$
$S.\Delta B \leftarrow S.\Delta L$
$S.\Delta L'' \leftarrow S.\Delta L + 1$
$L.\Delta L \leftarrow L.\Delta L'' + 1$



vecteur des n° des nœuds

ici V = i j k

(i,1) dépend de (j,1) : V.ΔL , V.ΔB

(i,1) dépend de (k,1) : V.ΔL , V.ΔL"

(j,2) dépend de (i,2) : S.ΔB , S.ΔL

⍝ *LISTE DU GROUPE DE* <Δ1Δ1Δ1> *PAGE* 1

LE 30/9/1975 *A* 11*H* 12*MN*

```
    ∇ Z←Δ1Δ1Δ1 V
[1]   ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←',('ΔTAS',(⍕V[2]),'Δ',⍕1)
    ∇

    ∇ Z←Δ2Δ1Δ1 V
[1]   ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←',('ΔTAS',(⍕V[2]),'Δ',⍕1),'+',('ΔTAS'
      ,(⍕V[3]),'Δ',⍕1)
    ∇

    ∇ Z←Δ2Δ2Δ2 V
[1]   ⍎('ΔTAS',(⍕V[2]),'Δ',⍕2),'←0'
    ∇

    ∇ Z←Δ2Δ3Δ2 V
[1]   ⍎('ΔTAS',(⍕V[3]),'Δ',⍕2),'←-',('ΔTAS',(⍕V[3]),'Δ',⍕3)
    ∇

    ∇ Z←Δ3Δ1Δ1 V
[1]   ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←',('ΔTAS',(⍕V[2]),'Δ',⍕1)
    ∇

    ∇ Z←Δ3Δ2Δ2 V
[1]   ⍎('ΔTAS',(⍕V[2]),'Δ',⍕2),'←0'
    ∇

    ∇ Z←Δ4Δ1Δ1 V
[1]   ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←',('ΔTAS',(⍕V[2]),'Δ',⍕1),'+',('ΔTAS'
      ,(⍕V[3]),'Δ',⍕1)
    ∇

    ∇ Z←Δ4Δ3Δ2 V
[1]   ⍎('ΔTAS',(⍕V[3]),'Δ',⍕2),'←',('ΔTAS',(⍕V[1]),'Δ',⍕2)
    ∇

    ∇ Z←Δ4Δ2Δ2 V
[1]   ⍎('ΔTAS',(⍕V[2]),'Δ',⍕2),'←',('ΔTAS',(⍕V[1]),'Δ',⍕2),'+1'
    ∇

    ∇ Z←Δ4Δ1Δ3 V
[1]   ⍎('ΔTAS',(⍕V[1]),'Δ',⍕3),'←',('ΔTAS',(⍕V[2]),'Δ',⍕3),'+1'
    ∇

    ∇ Z←Δ5Δ1Δ1 V
[1]   ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←',('ΔTAS',(⍕V[2]),'Δ',⍕1)
    ∇
```

⍝ *LISTE DU GROUPE DE <Δ1Δ1Δ1>* *PAGE* 2

LE 30/9/1975 *A* 11*H* 13*MN*

```
    ∇ Z←Δ5Δ2Δ2 V
[1]   ⍎('ΔTAS',(⍕V[2]),'Δ',⍕2),'←',('ΔTAS',(⍕V[1]),'Δ',⍕2)
    ∇

    ∇ Z←Δ5Δ1Δ3 V
[1]   ⍎('ΔTAS',(⍕V[1]),'Δ',⍕3),'←1'
    ∇

    ∇ Z←Δ6Δ1Δ1 V
[1]   ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←(2*',('ΔTAS',(⍕V[1]),'Δ',⍕2),')×NOMBR
      E'
    ∇
```

Voici maintenant la procédure sémantique nombre

```
      ∇ Z←NOMBRE
[1]   Z←⍎LIRE,ΔVAL[1↑V;]
      ∇
```

Utilisation de l'interpréteur généré.

Sur cet exemple simple

le temps des différentes

phases est proportionnel

à la longueur du texte entré.

```
    GO
  : 1,0 0 0 1
  : FIN
... TEMPS ANALYSE LEXICALE :0 0 30
°°° TEMPS ANALYSE SYNTAXIQ :0 0 57
VALEUR EN DECIMAL : 1.0625
ooo TEMPS ANALYSE SEMANTIQ :0 2 17
```

```
    GO
  : 1 1 1
  : FIN
... TEMPS ANALYSE LEXICALE :0 0 16
°°° TEMPS ANALYSE SYNTAXIQ :0 0 37
VALEUR EN DECIMAL : 7
ooo TEMPS ANALYSE SEMANTIQ :0 1 12
```

```
    GO
  : 1 1 1 1 1 1
  : FIN
... TEMPS ANALYSE LEXICALE :0 0 31
°°° TEMPS ANALYSE SYNTAXIQ :0 1 14
VALEUR EN DECIMAL : 63
ooo TEMPS ANALYSE SEMANTIQ :0 2 22
```

```
    GO
  : 1 1 1 1 1 1 1 1 1 1 1 1
  : FIN
... TEMPS ANALYSE LEXICALE :0 1 0
°°° TEMPS ANALYSE SYNTAXIQ :0 2 2
VALEUR EN DECIMAL : 4095
ooo TEMPS ANALYSE SEMANTIQ :0 5 1
```

Un mini compilateur

Ce compilateur permet de tester notre système. Il est volontairement simple.

Le langage de programmation permet de manipuler des entiers. En voici la description. Un programme est une suite d'instructions séparées par un ';' .

Les instructions sont soit avec étiquette ΔL : ΔS , soit sans étiquette ΔS , l'instruction peut prendre deux formes : affectation V ← ΔEX

branchement conditionnel ΔB → ΔL .

Si ΔB = 1 alors aller à ΔL , sinon en séquence

ΔB est le résultat d'une comparaison entre deux expressions

ΔEx : une expression est une combinaison d'additions et de soustractions entre des variables et des nombres.

Un programme écrit dans ce langage très simple est transformé en un programme écrit en assembleur pour IBM 360 . Le programme objet n'utilise qu'un seul registre : le registre 2. Du fait de la simplicité de la définition sémantique on fait une compilation en un seul passage le problème des boucles est traité de façon peu rationnelle.

Quand on rencontre une étiquette, on définit une zone associée qui contiendra son adresse. On génère des instructions qui, placées au début du programme, chargeront cette adresse dans cette zone au début de l'exécution du programme. Ensuite lors de l'instruction de branchement, on génère une séquence de saut conditionnel avec adressage indirect.

Aucune vérification n'est faite quant à l'initialisation des variables, la définition et la double définition des étiquettes ...

La seule optimisation consiste à tester, lois de la concaténation du code généré, l'existence d'une séquence ST 2,A , L 2,A pour la supprimer.

Malgré tout, pour un exemple aussi simple on utilise quatre attributs dont NUM est hérité. NBST et NUM sont d'un intérêt limité, ils servent à numéroter

les instructions. On peut faire descendre NUM à ΔS en ajoutant aux règles 4 et 5 NUM.ΔS ← NUM.ΔST et ainsi de suite ... ce qui permettrait d'éditer des messages d'erreurs, en indiquant le n° de l'instruction source en cause, si l'on effectuait des tests de validité sémantique.

La définition de ce compilateur, bien imparfait, demande déjà une page de définitions du langage et 21 procédures sémantiques très simples.

On utilise en plus des attributs, trois variables globales. CPT un compteur pour définir des noms de zones de données auxiliaires :

DSCT une matrice contenant le code des définitions de données en assembleur

CSCT une matrice contenant le code nécessaire à la mise en place des adresses des étiquettes.

La construction des tables par l'analyseur syntaxique a demandé un peu plus d'une minute. La génération des fonctions sémantiques demande trente secondes. L'écriture des règles sémantiques et des procédures associées a dû être reprise trois fois, pour la mise au point de ce mini compilateur qui a été "écrit" en deux jours. Beaucoup d'améliorations sont possibles, mais aux dépends de la simplicité. Le but de cet exemple est de montrer les possibilités du système et non de définir un compilateur.

CONSTRUCTION SYNTAXIQUE

```
        ANALYSE

. ....ENTREZ.LES REGLES....

   : ΔA ΔSL
   : ΔSL ΔSL;ΔST
   : ΔSL ΔST
   : ΔST ΔS
   : ΔST ΔL:ΔS
   : ΔS V←ΔEX
   : ΔS ΔB→ΔL
   : ΔB ΔEX=ΔEX
   : ΔB ΔEX<ΔEX
   : ΔB ΔEX>ΔEX
   : ΔL V
   : ΔEX ΔP
   : ΔEX ΔEX+ΔP
   : ΔEX ΔEX-ΔP
   : ΔP V
   : ΔP C
   : ΔE
      v
      P (ΔEX)
   :
  GRAMMAIRE LUE
```

règles de syntaxe introduites par l'utilisateur

Informations données par le programme constructeur

```
****  DICTIONNAIRE    *******

[1]  C
[2]  V
[3]  (
[4]  )
[5]  ;
[6]  ←
[7]  →
[8]  +
[9]  -
[10] |
[11] <
[12] =
[13] >
[14] :
[15] ΔA
[16] ΔB
[17] ΔEX
[18] ΔL
[19] ΔP
[20] ΔS
[21] ΔSL
[22] ΔST
```

```
****  PRODUCTIONS     *******

 1      ΔA    ::= ΔSL
 2      ΔB    ::= ΔEX = ΔEX
 3      ΔB    ::= ΔEX < ΔEX
 4      ΔB    ::= ΔEX > ΔEX
 5      ΔEX   ::= ΔP
 6      ΔEX   ::= ΔEX + ΔP
 7      ΔEX   ::= ΔEX - ΔP
 8      ΔL    ::= V
 9      ΔP    ::= V
10      ΔP    ::= C
11      ΔP    ::= ( ΔEX )
12      ΔS    ::= V ← ΔEX
13      ΔS    ::= ΔB → ΔL
14      ΔSL   ::= ΔSL ; ΔST
15      ΔSL   ::= ΔST
16      ΔST   ::= ΔS
17      ΔST   ::= ΔL : ΔS
```

TABLE DES DERIVATIONS

C	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
V	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
(	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
)	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
;	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
←	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
→	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
+	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
|	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
<	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
=	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
>	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
:	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
Δ*A*	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Δ*B*	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	0
Δ*EX*	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0
Δ*L*	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
Δ*P*	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
Δ*S*	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	0
Δ*SL*	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1
Δ*ST*	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	1

TABLE F11

|	Δ*A*	|	|	Δ*EX*	+	-	Δ*P*	→
;	Δ*B*	→	|	Δ*EX*	-	<	Δ*P*	→
:	Δ*B*	→	|	Δ*EX*	<	<	Δ*P*	<
|	Δ*B*	→	→	Δ*L*	;	>	Δ*P*	<
←	Δ*EX*	|	;	Δ*L*	:	;	Δ*P*	<
←	Δ*EX*	;	|	Δ*L*	|	+	Δ*P*	+
(	Δ*EX*	)	→	Δ*P*	|	>	Δ*P*	-
(	Δ*EX*	+	←	Δ*P*	;	>	Δ*P*	-
(	Δ*EX*	-	←	Δ*P*	)	:	Δ*P*	>
←	Δ*EX*	+	(	Δ*P*	)	:	Δ*P*	+
←	Δ*EX*	-	+	Δ*P*	+	:	Δ*P*	-
=	Δ*EX*	→	(	Δ*P*	+	:	Δ*P*	<
;	Δ*EX*	=	+	Δ*P*	+	:	Δ*P*	>
;	Δ*EX*	+	-	Δ*P*	+	|	Δ*P*	=
;	Δ*EX*	-	(	Δ*P*	-	|	Δ*P*	+
=	Δ*EX*	+	+	Δ*P*	-	|	Δ*P*	-
=	Δ*EX*	-	-	Δ*P*	-	|	Δ*P*	<
<	Δ*EX*	→	-	Δ*P*	)	|	Δ*P*	>
;	Δ*EX*	<	+	Δ*P*	;	+	Δ*P*	|
<	Δ*EX*	+	←	Δ*P*	+	-	Δ*P*	|
<	Δ*EX*	-	←	Δ*P*	-	;	Δ*S*	|
>	Δ*EX*	→	-	Δ*P*	;	;	Δ*S*	;
;	Δ*EX*	>	=	Δ*P*	→	:	Δ*S*	;
>	Δ*EX*	+	;	Δ*P*	=	|	Δ*S*	;
>	Δ*EX*	-	+	Δ*P*	=	:	Δ*S*	|
:	Δ*EX*	=	;	Δ*P*	+	|	Δ*S*	|
:	Δ*EX*	+	;	Δ*P*	-	|	Δ*SL*	|
:	Δ*EX*	-	-	Δ*P*	=	|	Δ*SL*	;
:	Δ*EX*	<	+	Δ*P*	→	;	Δ*ST*	|
:	Δ*EX*	>	=	Δ*P*	+	;	Δ*ST*	;
|	Δ*EX*	=	=	Δ*P*	-	|	Δ*ST*	;
							|	Δ*ST*	|

Productions triées

Table C11

```
****  PRODUCTIONS    *******

  1       ΔSL  ::= ΔSL ; ΔST
  2       ΔSL  ::= ΔST
  3       ΔA   ::= ΔSL
  4       ΔST  ::= ΔL : ΔS
  5       ΔST  ::= ΔS
  6       ΔEX  ::= ΔEX + ΔP
  7       ΔEX  ::= ΔEX - ΔP
  8       ΔEX  ::= ΔP
  9       ΔS   ::= ΔB → ΔL
 10       ΔB   ::= ΔEX = ΔEX
 11       ΔB   ::= ΔEX < ΔEX
 12       ΔB   ::= ΔEX > ΔEX
 13       ΔS   ::= V ← ΔEX
 14       ΔP   ::= ( ΔEX )
 15       ΔL   ::= V
 16       ΔP   ::= V
 17       ΔP   ::= C
```

```
C     ___FF_FFFFFFF_________
V     ___FFVFFFFFFFF________
(     VVV___________________
)     ___FF_FFFFFFF_________
;     VVV___________________
←     VVV___________________
→     _V____________________
+     VVV___________________
-     VVV___________________
|     VVV___________________
<     VVV___________________
=     VVV___________________
>     VVV___________________
:     VVV___________________
ΔA    _________F____________
ΔB    _______V______________
ΔEX   ___VF_FVVFVVV_________
ΔL    ____F___F___V_________
ΔP    ___FF_FFFFFFF_________
ΔS    ____F___F_____________
ΔSL   ____V___F_____________
ΔST   ____F___F_____________
```

```
.....CONTEXTE.....

0 0 0 2 0 0 0 0 0 0 0 0 0 0 3 0 0

... GAUCHE ...

;       ΔST
|       ΔST

...GAUCHE ET DROIT...

    →       ΔL      ;
    ;       ΔL      :
    |       ΔL      :
    →       ΔL      |
PSEUDO TERMINAUX DE LA GRAMMAIRE
POUR LES IDENTIFICATEURS

V
POUR LES NOMBRES

C
POUR LES CHAINES DE CARACTERES
```

informations introduites par l'utilisateur pour l'analyseur lexical

Les tables produites ne sont pas jointes car elles sont peu lisibles et les informations qu'elles contiennent sont éditées ici sous une forme plus agréable.

CONSTRUCTION SEMANTIQUE

L'introduction des règles sémantiques n'est pas complète car elles ont été introduites puis modifiées. Ici l'on est en mode modification. Seules les tables de règles RSi non modifiées sont conservées. Tout le reste du travail est repris.

```
      COSEM 4
.. POUR LA REGLE :
ΔST ::= ΔS
 ENTREZ LA REGLE SEMANTIQUE 1
 : CODE.ΔST,CODE.ΔS
 ENTREZ LA REGLE SEMANTIQUE 2
 :
+++ SEMANTIQUE.LUE +++

LISTE DES ATTRIBUTS

CODE
NBST
NUM
AD
...ATTRIBUTS DES NON TERMINAUX ...

ΔA              1 1 0 0
ΔB              1 0 0 1
ΔEX             1 0 0 1
ΔL              0 0 0 1
ΔP              1 0 0 1
ΔS              1 0 0 0
ΔSL             1 1 0 0
ΔST             1 0 1 0

***** POUR LA REGLE *****
  1          ΔA ::= ΔSL
oooo LA SEMANTIQUE INTRODUITE EST :
CODE.ΔA ← PROG ( CODE.ΔSL )
NBST.ΔA ← NBST.ΔSL
***** POUR LA REGLE *****
  2          ΔSL ::= ΔSL ; ΔST
oooo LA SEMANTIQUE INTRODUITE EST :
CODE.ΔSL ← ( CODE.ΔSL" ) CAT ( CODE.ΔST )
NBST.ΔSL ← NBST.ΔSL" + 1
NUM.ΔST ← NBST.ΔSL
***** POUR LA REGLE *****
  3          ΔSL ::= ΔST
oooo LA SEMANTIQUE INTRODUITE EST :
CODE.ΔSL ← CODE.ΔST
NBST.ΔSL ← 1
NUM.ΔST ← NBST.ΔSL
***** POUR LA REGLE *****
  4          ΔST ::= ΔS
oooo LA SEMANTIQUE INTRODUITE EST :
CODE.ΔST ← CODE.ΔS
***** POUR LA REGLE *****
  5          ΔST ::= ΔL : ΔS
oooo LA SEMANTIQUE INTRODUITE EST :
CODE.ΔST ← ( ( AD.ΔL ) LABEL ( NUM.ΔST ) ) CAT ( CODE.ΔS )
```

```
***** POUR LA REGLE *****
  6          ΔS ::= V ← ΔEX
∘∘∘∘ LA SEMANTIQUE INTRODUITE EST :
CODE.ΔS ← ( CODE.ΔEX ) CAT ( ( DS ( NOM ) ) AFFEC ( AD.ΔEX ) )
***** POUR LA REGLE *****
  7          ΔS ::= ΔB → ΔL
∘∘∘∘ LA SEMANTIQUE INTRODUITE EST :
CODE.ΔS ← ( CODE.ΔB ) CAT ( ( AD.ΔB ) BRNC ( AD.ΔL ) )
***** POUR LA REGLE *****
  8          ΔB ::= ΔEX = ΔEX
∘∘∘∘ LA SEMANTIQUE INTRODUITE EST :
CODE.ΔB ← ( AD.ΔB ) LOC ( ( AD.ΔEX ) EQU ( AD.ΔEX" ) )
AD.ΔB ← PLACE
***** POUR LA REGLE *****
  9          ΔB ::= ΔEX < ΔEX
∘∘∘∘ LA SEMANTIQUE INTRODUITE EST :
CODE.ΔB ← ( AD.ΔB ) LOC ( ( AD.ΔEX ) INF ( AD.ΔEX" ) )
AD.ΔB ← PLACE
***** POUR LA REGLE *****
 10          ΔB ::= ΔEX : > ΔEX
∘∘∘∘ LA SEMANTIQUE INTRODUITE EST :
CODE.ΔB ← ( AD.ΔB ) LOC ( ( AD.ΔEX ) SUP ( AD.ΔEX" ) )
AD.ΔB ← PLACE
***** POUR LA REGLE *****
 11          ΔL ::= V
∘∘∘∘ LA SEMANTIQUE INTRODUITE EST :
AD.ΔL ← DS ( NOM )
***** POUR LA REGLE *****
 12          ΔEX ::= ΔP
∘∘∘∘ LA SEMANTIQUE INTRODUITE EST :
CODE.ΔEX ← CODE.ΔP
AD.ΔEX ← AD.ΔP
***** POUR LA REGLE *****
 13          ΔEX ::= ΔEX + ΔP
∘∘∘∘ LA SEMANTIQUE INTRODUITE EST :
CODE.ΔEX ← ( AD.ΔEX ) LOC ( ( CODE.ΔP ) CAT ( CODE.ΔEX" ) CAT ( ( AD
    EX" ) ADD ( AD.ΔP ) ) )
AD.ΔEX ← PLACE
***** POUR LA REGLE *****
 14          ΔEX ::= ΔEX - ΔP
∘∘∘∘ LA SEMANTIQUE INTRODUITE EST :
CODE.ΔEX ← ( AD.ΔEX ) LOC ( ( CODE.ΔP ) CAT ( CODE.ΔEX" ) CAT ( ( AD
      EX" ) SOUS ( AD.ΔP ) ) )
AD.ΔEX ← PLACE
***** POUR LA REGLE *****
 15          ΔP ::= V
∘∘∘∘ LA SEMANTIQUE INTRODUITE EST :
CODE.ΔP ← VIDE
AD.ΔP ← DS ( NOM )
***** POUR LA REGLE *****
 16          ΔP ::= C
∘∘∘∘ LA SEMANTIQUE INTRODUITE EST :
CODE.ΔP ← VIDE
AD.ΔP ← DC ( NOM )
***** POUR LA REGLE *****
 17          ΔP ::= ( ΔEX )
∘∘∘∘ LA SEMANTIQUE INTRODUITE EST :
CODE.ΔP ← CODE.ΔEX
AD.ΔP ← AD.ΔEX
CONSTRUCTION DES ΔSI TERMINEE
CONSTRUCTION DES ΔI TERMINEE
LISTE DES ATTRIBUTS HERITEES
□:
```

Ci-dessous les règles de productions introduites lors de la définition et à droite, le dictionnaire complet. Jusqu'à 22, on a la partie syntaxique, ensuite on voit apparaître les attributs et les noms des procédures.

```
. ΔPROD EDIT 0
```

1	ΔA	::= ΔSL
2	ΔSL	::= ΔSL ; ΔST
3	ΔSL	::= ΔST
4	ΔST	::= ΔS
5	ΔST	::= ΔL : ΔS
6	ΔS	::= V ← ΔEX
7	ΔS	::= ΔB → ΔL
8	ΔB	::= ΔEX = ΔEX
9	ΔB	::= ΔEX < ΔEX
10	ΔB	::= ΔEX > ΔEX
11	ΔL	::= V .
12	ΔEX	::= ΔP
13	ΔEX	::= ΔEX + ΔP
14	ΔEX	::= ΔEX - ΔP
15	ΔP	::= V .
16	ΔP	::= C
17	ΔP	::= (ΔEX)

```
**** DICTIONNAIRE

[1]  C
[2]  V
[3]  (
[4]  )
[5]  ;
[6]  ←
[7]  →
[8]  +
[9]  -
[10] |
[11] <
[12] =
[13] >
[14] :
[15] ΔA
[16] ΔB
[17] ΔEX
[18] ΔL
[19] ΔP
[20] ΔS
[21] ΔSL
[22] ΔST
[23] CODE
[24] .
[25] PROG
[26] HBST
[27] "
[28] CAT
[29] 1
[30] NUM
[31] AD
[32] LABEL
[33] DS
[34] NOM
[35] AFFEC
[36] BRNC
[37] EQU
[38] INF
[39] SUP
[40] EX
[41] ADD
[42] SOUS
[43] VIDE
[44] DC
[45] PLACE
[46] LOC
```

```
⍝ LISTE DU GROUPE DE <ΔS1>

⍝ LE 25/9/1975 A 11H 26MN
```

```
    ∇ Z←ΔS1 V
[1]   V←V×100
[2]   Z← 0 2 ⍴0
[3]   Z←Z,[1](V[2]+1),V[1]+1
[4]   Z←Z,[1](V[2]+2),V[1]+2
    ∇
```

```
    ∇ Z←ΔS2 V
[1]   V←V×100
[2]   Z← 0 2 ⍴0
[3]   Z←Z,[1](V[2]+1),V[1]+1
[4]   Z←Z,[1](V[3]+1),V[1]+1
[5]   Z←Z,[1](V[2]+2),V[1]+2
[6]   Z←Z,[1](V[1]+2),V[3]+3
    ∇
```

```
    ∇ Z←ΔS3 V
[1]   V←V×100
[2]   Z← 0 2 ⍴0
[3]   Z←Z,[1](V[2]+1),V[1]+1
[4]   Z←Z,[1](V[1]+2),V[2]+3
    ∇
```

```
    ∇ Z←ΔS4 V
[1]   V←V×100
[2]   Z← 0 2 ⍴0
[3]   Z←Z,[1](V[2]+1),V[1]+1
    ∇
```

```
    ∇ Z←ΔS5 V
[1]   V←V×100
[2]   Z← 0 2 ⍴0
[3]   Z←Z,[1](V[2]+4),V[1]+1
[4]   Z←Z,[1](V[1]+3),V[1]+1
[5]   Z←Z,[1](V[3]+1),V[1]+1
    ∇
```

```
    ∇ Z←ΔS6 V
[1]   V←V×100
[2]   Z← 0 2 ⍴0
[3]   Z←Z,[1](V[2]+1),V[1]+1
[4]   Z←Z,[1](V[2]+4),V[1]+1
    ∇
```

```
    ∇ Z←ΔS7 V
[1]   V←V×100
[2]   Z← 0 2 ⍴0
[3]   Z←Z,[1](V[2]+1),V[1]+1
[4]   Z←Z,[1](V[2]+4),V[1]+1
[5]   Z←Z,[1](V[3]+4),V[1]+1
    ∇
```

```
⍝ LISTE DU GROUPE DE <ΔS1>

⍝ LE 25/9/1975 A 11H 27MN
```

```
    ∇ Z←ΔS8 V
[1]   V←V×100
[2]   Z← 0 2 ⍴0
[3]   Z←Z,[1](V[1]+4),V[1]+1
[4]   Z←Z,[1] (V[2]+4),V[1]+1
[5]   Z←Z,[1](V[3]+4),V[1]+1
    ∇
```

```
    ∇ Z←ΔS9 V
[1]   V←V×100
[2]   Z← 0 2 ⍴0
[3]   Z←Z,[1](V[1]+4),V[1]+1
[4]   Z←Z,[1](V[2]+4),V[1]+1
[5]   Z←Z,[1](V[3]+4),V[1]+1
    ∇
```

```
    ∇ Z←ΔS10 V
[1]   V←V×100
[2]   Z← 0 2 ⍴0
[3]   Z←Z,[1](V[1]+4),V[1]+1
[4]   Z←Z,[1](V[2]+4),V[1]+1
[5]   Z←Z,[1](V[3]+4),V[1]+1
    ∇
```

```
    ∇ Z←ΔS11 V
[1]   V←V×100
[2]   Z← 0 2 ⍴0
    ∇
```

```
    ∇ Z←ΔS12 V
[1]   V←V×100
[2]   Z← 0 2 ⍴0
[3]   Z←Z,[1] (V[2]+1),V[1]+1
[4]   Z←Z,[1] (V[2]+4),V[1]+4
    ∇
```

```
    ∇ Z←ΔS13 V
[1]   V←V×100
[2]   Z← 0 2 ⍴0
[3]   Z←Z,[1] (V[1]+4),V[1]+1
[4]   Z←Z,[1] (V[3]+1),V[1]+1
[5]   Z←Z,[1](V[2]+1),V[1]+1
[6]   Z←Z,[1](V[2]+4),V[1]+1
[7]   Z←Z,[1](V[3]+4),V[1]+1
    ∇
```

```
    ∇ Z←ΔS14 V
[1]   V←V×100
[2]   Z← 0 2 ⍴0
[3]   Z←Z,[1](V[1]+4),V[1]+1
[4]   Z←Z,[1](V[3]+1),V[1]+1
```

```
⍝ LISTE DU GROUPE DE <ΔS1>

  LE  25/9/1975  A  11H 30MN

[5]    Z←Z,[1] (V[2]+1),V[1]+1
[6]    Z←Z,[1](V[2]+4),V[1]+1
[7]    Z←Z,[1](V[3]+4),V[1]+1
     ∇


     ∇ Z←ΔS15 V
[1]    V←V×100
[2]    Z← 0 2 ρ0
     ∇


     ∇ Z←ΔS16 V
[1]    V←V×100
[2]    Z← 0 2 ρ0
     ∇


     ∇ Z←ΔS17 V
[1]    V←V×100
[2]    Z← 0 2 ρ0
[3]    Z←Z,[1](V[2]+1),V[1]+1
[4]    Z←Z,[1](V[2]+4),V[1]+4
     ∇
```

Les fonctions des groupes <ΔS1> et <Δ1Δ1Δ1> sont générées par le constructeur sémantique. Celles du groupe <ADD> sont écrites par l'utilisateur. Ce sont les procédures sémantiques.

⍝ *LISTE DU GROUPE DE* <Δ1Δ1Δ1> *PAGE*

LE 25/9/1975 *A* 11*H* 31*MN*

```
     ∇ Z←Δ1Δ1Δ1 V
[1]    ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←PROG(',('ΔTAS',(⍕V[2]),'Δ',⍕1),')'
     ∇

     ∇ Z←Δ1Δ1Δ2 V
[1]    ⍎('ΔTAS',(⍕V[1]),'Δ',⍕2),'←',('ΔTAS',(⍕V[2]),'Δ',⍕2)
     ∇

     ∇ Z←Δ2Δ1Δ1 V
[1]    ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←(',('ΔTAS',(⍕V[2]),'Δ',⍕1),')CAT(',
       ΔTAS',(⍕V[3]),'Δ',⍕1),')'
     ∇

     ∇ Z←Δ2Δ1Δ2 V
[1]    ⍎('ΔTAS',(⍕V[1]),'Δ',⍕2),'←',('ΔTAS',(⍕V[2]),'Δ',⍕2),'+1'
     ∇

     ∇ Z←Δ2Δ3Δ3 V
[1]    ⍎('ΔTAS',(⍕V[3]),'Δ',⍕3),'←',('ΔTAS',(⍕V[1]),'Δ',⍕2)
     ∇

     ∇ Z←Δ3Δ1Δ1 V
[1]    ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←',('ΔTAS',(⍕V[2]),'Δ',⍕1)
     ∇

     ∇ Z←Δ3Δ1Δ2 V
[1]    ⍎('ΔTAS',(⍕V[1]),'Δ',⍕2),'←1'
     ∇

     ∇ Z←Δ3Δ2Δ3 V
[1]    ⍎('ΔTAS',(⍕V[2]),'Δ',⍕3),'←',('ΔTAS',(⍕V[1]),'Δ',⍕2)
     ∇

     ∇ Z←Δ4Δ1Δ1 V
[1]    ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←',('ΔTAS',(⍕V[2]),'Δ',⍕1)
     ∇

     ∇ Z←Δ5Δ1Δ1 V
[1]    ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←((',('ΔTAS',(⍕V[2]),'Δ',⍕4),')LABEL(
       ,('ΔTAS',(⍕V[1]),'Δ',⍕3),'))CAT(',('ΔTAS',(⍕V[3]),'Δ',⍕1),')'
     ∇

     ∇ Z←Δ6Δ1Δ1 V
[1]    ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←(',('ΔTAS',(⍕V[2]),'Δ',⍕1),')CAT((DS
       NOM))AFFEC(',('ΔTAS',(⍕V[2]),'Δ',⍕4),'))'
     ∇
```

⍝ *LISTE DU GROUPE DE* <Δ1Δ1Δ1> *PAGE* 2

LE 25/9/1975 *A* 11*H* 33*MN*

```
     ∇ Z←Δ7Δ1Δ1 V
[1]    ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←(',('ΔTAS',(⍕V[2]),'Δ',⍕1),')CAT ((',(
       'ΔTAS',(⍕V[2]),'Δ',⍕4),')BRNC (',('ΔTAS',(⍕V[3]),'Δ',⍕4),'))'
     ∇

     ∇ Z←Δ8Δ1Δ1 V
[1]    ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←(',('ΔTAS',(⍕V[1]),'Δ',⍕4),')LOC((',(
       'ΔTAS',(⍕V[2]),'Δ',⍕4),')EQU(',('ΔTAS',(⍕V[3]),'Δ',⍕4),'))'
     ∇

     ∇ Z←Δ8Δ1Δ4 V
[1]    ⍎('ΔTAS',(⍕V[1]),'Δ',⍕4),'←PLACE'
     ∇

     ∇ Z←Δ9Δ1Δ1 V
[1]    ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←(',('ΔTAS',(⍕V[1]),'Δ',⍕4),')LOC((',(
       'ΔTAS',(⍕V[2]),'Δ',⍕4),')INF(',('ΔTAS',(⍕V[3]),'Δ',⍕4),'))'
     ∇

     ∇ Z←Δ9Δ1Δ4 V
[1]    ⍎('ΔTAS',(⍕V[1]),'Δ',⍕4),'←PLACE'
     ∇

     ∇ Z←Δ10Δ1Δ1 V
[1]    ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←(',('ΔTAS',(⍕V[1]),'Δ',⍕4),')LOC((',(
       'ΔTAS',(⍕V[2]),'Δ',⍕4),')SUP(',('ΔTAS',(⍕V[3]),'Δ',⍕4),'))'
     ∇

     ∇ Z←Δ10Δ1Δ4 V
[1]    ⍎('ΔTAS',(⍕V[1]),'Δ',⍕4),'←PLACE'
     ∇

     ∇ Z←Δ11Δ1Δ4 V
[1]    ⍎('ΔTAS',(⍕V[1]),'Δ',⍕4),'←DS ⍵OM)'
     ∇

     ∇ Z←Δ12Δ1Δ1 V
[1]    ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←',('ΔTAS',(⍕V[2]),'Δ',⍕1)
     ∇

     ∇ Z←Δ12Δ1Δ4 V
[1]    ⍎('ΔTAS',(⍕V[1]),'Δ',⍕4),'←',('ΔTAS',(⍕V[2]),'Δ',⍕4)
     ∇
```

A *LISTE DU GROUPE DE* <Δ1Δ1Δ1> *PAGE*

LE 25/9/1975 *A* 11*H* 36*MN*

```
    ∇ Z←Δ13Δ1Δ1 V
[1]   ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←(',('ΔTAS',(⍕V[1]),'Δ',⍕4),')LOC((',
      'ΔTAS',(⍕V[3]),'Δ',⍕1),')CAT (',('ΔTAS',(⍕V[2]),'Δ',⍕1),')CAT((('
      ('ΔTAS',(⍕V[2]),'Δ',⍕4),')ADD(',('ΔTAS',(⍕V[3]),'Δ',⍕4),')))'
    ∇

    ∇ Z←Δ13Δ1Δ4 V
[1]   ⍎('ΔTAS',(⍕V[1]),'Δ',⍕4),'←PLACE'
    ∇

    ∇ Z←Δ14Δ1Δ1 V
[1]   ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←(',('ΔTAS',(⍕V[1]),'Δ',⍕4),')LOC((',
      'ΔTAS',(⍕V[3]),'Δ',⍕1),')CAT(',('ΔTAS',(⍕V[2]),'Δ',⍕1),')CAT ((('
      ('ΔTAS',(⍕V[2]),'Δ',⍕4),')SOUS(',('ΔTAS',(⍕V[3]),'Δ',⍕4),')))'
    ∇

    ∇ Z←Δ14Δ1Δ4 V
[1]   ⍎('ΔTAS',(⍕V[1]),'Δ',⍕4),'←PLACE'
    ∇

    ∇ Z←Δ15Δ1Δ1 V
[1]   ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←VIDE'
    ∇

    ∇ Z←Δ15Δ1Δ4 V
[1]   ⍎('ΔTAS',(⍕V[1]),'Δ',⍕4),'←DS (NOM)'
    ∇

    ∇ Z←Δ16Δ1Δ1 V
[1]   ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←VIDE'
    ∇

    ∇ Z←Δ16Δ1Δ4 V
[1]   ⍎('ΔTAS',(⍕V[1]),'Δ',⍕4),'←DC(NOM)'
    ∇

    ∇ Z←Δ17Δ1Δ1 V
[1]   ⍎('ΔTAS',(⍕V[1]),'Δ',⍕1),'←',('ΔTAS',(⍕V[2]),'Δ',⍕1)
    ∇

    ∇ Z←Δ17Δ1Δ4 V
[1]   ⍎('ΔTAS',(⍕V[1]),'Δ',⍕4),'←',('ΔTAS',(⍕V[2]),'Δ',⍕4)
    ∇
```

⍝ LISTE DU GROUPE DE <ADD> PAGE 1

LE 1/10/1975 A 15H 59MN

```
    ∇ Z←A ADD B
[1]   Z←'    L  2,',A
[2]   Z←Z CAT '    A  2,',B
    ∇


    ∇ Z←A AFFEC B;P
[1]   Z←VIDE
[2]   →0 IF∧/(P↑A)=(P←(⍴A)⌈⍴B)↑B
[3]   Z← 0 2 ⍴''
[4]   Z←Z CAT '   MVC  ',A,'(4),',B
    ∇


    ∇ Z←A BRNC B;L
[1]   Z←'    L  2,',A
[2]   Z←Z CAT '    LTR 2,2'
[3]   Z←Z CAT '    L  2,',B
[4]   Z←Z CAT '    BCR  7,2'
    ∇


    ∇ Z←A CAT B;LA;FB
[1]   →S IF 6>⍴LA←,(¯1,¯1↑⍴A)↑A
[2]   →S IF 5>⍴FB←,(1,¯1↑⍴B)↑B
[3]   →OPT IF(∧/LA[5 6]='ST')∧(FB[5]='L')∧∧/(10↓LA)=9↓(¯1+⍴L
      A)↑FB
[4]  S:Z←A CAT B
[5]   →0
[6]  OPT:Z←(¯1 0 ↓A) CAT 1 0 ↓B
    ∇


    ∇ Z←DC N
[1]   Z←'A',⍕N
[2]   →0 IF∨/DSCT[;⍳⍴Z]∧.=Z
[3]   DSCT←DSCT CAT Z,'   DC  F''',(LIRE N),''''
    ∇


    ∇ Z←DS N
[1]   Z←'A',⍕N
[2]   →0 IF∨/DSCT[;⍳⍴Z]∧.=Z
[3]   DSCT←DSCT CAT Z,'   DS  F'
    ∇


    ∇ Z←A EQU B
[1]   Z←'    L  2,',A
[2]   Z←Z CAT '    C  2,',B
[3]   Z←Z CAT '    LA  2,0'
[4]   Z←Z CAT '    BNE  *+8'
[5]   Z←Z CAT '    LA  2,1'
    ∇
```

A LISTE DU GROUPE DE <ADD> PAGE 2

LE 1/10/1975 A 16H 1MN

```
     ∇ FINFONC
[1]    BR 14
     ∇


     ∇ Z←A INF B
[1]    Z←'     L  2,',A
[2]    Z←Z CAT '     C  2,',B
[3]    Z←Z CAT '     LA  2,0'
[4]    Z←Z CAT '     BNL  *+8'
[5]    Z←Z CAT '     LA  2,1'
     ∇


     ∇ INITFONC
[1]    *DEBUT DU PROG
[2]    PROG START 0
[3]    USING PROG,15
     ∇


     ∇ Z←A LABEL B;L
[1]    CSCT←CSCT CAT '     LA  2,I',⍕B
[2]    CSCT←CSCT CAT '     ST  2,',A
[3]    Z←(1,⍴Z)⍴Z←'I',(⍕B),'   EQU * '
     ∇


     ∇ Z←A LOC X
[1]    Z←X CAT '     ST  2,',A
     ∇


     ∇ Z←NOM
[1]    Z←⍙VAL[1↑V;1]
     ∇


     ∇ Z←NOMBRE
[1]    Z←⍎LIRE,⍙VAL[1↑V;]
     ∇


     ∇ Z←PLACE
[1]    Z←'B',⍕CPT←CPT+1
[2]    DSCT←DSCT CAT Z,'   DS  F'
     ∇


     ∇ Z←PROG A
[1]    →Z←0 IF~VAL
[2]   S1:Z←INIT CAT CSCT CAT '*INSTRUCTIONS' CAT A CAT FINAL
       CAT DSCT CAT '    END'
     ∇
```

⍝ LISTE DU GROUPE DE <ADD> PAGE 3

LE 1/10/1975 A 16H 2MN

```
    ∇ Z←A SOUS B
[1]   Z←'     L  2,',A
[2]   Z←Z CAT '     S  2,',B
    ∇


    ∇ Z←A SUP B
[1]   Z←'     L  2,',A
[2]   Z←Z CAT '     C  2,',B
[3]   Z←Z CAT '     LA  2,0'
[4]   Z←Z CAT '     BNH  *+8'
[5]   Z←Z CAT '     LA  2,1'
    ∇


    ∇ TRDEB
[1]   DSCT← 1 16 ⍴'*ZONE DE DONNEES'
[2]   CSCT← 1 15 ⍴'*AD. DES LABELS'
[3]   CPT←0
[4]   LD←LU←''
    ∇


    ∇ TRFIN
[1]   →ER IF~VAL
[2]  S:'......PROGRAMME......',SL
[3]   PG←ΔTAS1Δ1
[4]   PG
[5]   SL,'......NOMBRE INSTRUCTIONS.....',SL
[6]   INS←ΔTAS1Δ2
[7]   INS
[8]   SL
[9]   →0
[10] ER:'***** ATTENTION PROGRAMME GENERE FAUX'
[11]  →S
    ∇


    ∇ Z←VIDE
[1]   Z← 1 0 ⍴0
    ∇
```

EXEMPLE DE PROGRAMME

Ce programme fait la somme des 100 premiers entiers

```
      GO
   : MAX←100;SOMME←0;
   : BOUCLE:SOMME←SOMME+MAX;MAX←MAX-1;
   : MAX>0→BOUCLE        FIN
... TEMPS ANALYSE LEXICALE :0 1 54
```

```
GO[12]
      ETAT 1
```

On fait un stop pour montrer l'arbre syntaxique

REGLE	NOEUD	PERE	FILS		VALEUR
1	[1]	0	2	0	
2	[2]	1	11	3	
4	[3]	2	4	0	
7	[4]	3	6	5	
11	[5]	4	0	0	BOUCLE
10	[6]	4	9	7	
12	[7]	6	8	0	
16	[8]	7	0	0	0
12	[9]	6	10	0	
15	[10]	9	0	0	MAX
2	[11]	2	18	12	
4	[12]	11	13	0	
6	[13]	12	14	0	MAX
14	[14]	13	16	15	
16	[15]	14	0	0	1
12	[16]	14	17	0	
15	[17]	16	0	0	MAX
2	[18]	11	26	19	
5	[19]	18	25	20	
6	[20]	19	21	0	SOMME
13	[21]	20	23	22	
15	[22]	21	0	0	MAX
12	[23]	21	24	0	
15	[24]	23	0	0	SOMME
11	[25]	19	0	0	BOUCLE
2	[26]	18	31	27	
4	[27]	26	28	0	
6	[28]	27	29	0	SOMME
12	[29]	28	30	0	
16	[30]	29	0	0	0
3	[31]	26	32	0	
4	[32]	31	33	0	
6	[33]	32	34	0	MAX
12	[34]	33	35	0	
16	[35]	34	0	0	100

Reprise avec trace

(9) donne la liste des attributs après le tri topologique

```
      →□LC
∘∘∘ TEMPS ANALYSE SYNTAXIQ :0 4 45
EXEC[9] 3501 3401 3504 3404 3301 3201 3101 3001
        2901 3004 2904 2801 2701 2601 2504 3102 2602
        1802 1903 2104 2201 2401 2301 2404 2304 2204
        2101 2001 1901 1801 1404 1501 1701 1601 1704
        1604 1504 1401 1301 1201 1101 604 1004 904
        804 704 601 504 401 301 201 101 1102 202 102
EXEC[17] Δ16Δ1Δ1
EXEC[17] Δ12Δ1Δ1
EXEC[17] Δ16Δ1Δ4          trace de la succession de l'appel
EXEC[17] Δ12Δ1Δ4
EXEC[17] Δ6Δ1Δ1           des différentes fonctions
EXEC[17] Δ4Δ1Δ1
EXEC[17] Δ3Δ1Δ1           d'évaluation sémantique
EXEC[17] Δ16Δ1Δ1
EXEC[17] Δ12Δ1Δ1
EXEC[17] Δ16Δ1Δ4
EXEC[17] Δ12Δ1Δ4
EXEC[17] Δ6Δ1Δ1
EXEC[17] Δ4Δ1Δ1
EXEC[17] Δ2Δ1Δ1
EXEC[17] Δ11Δ1Δ4
EXEC[17] Δ3Δ1Δ2
EXEC[17] Δ2Δ1Δ2
EXEC[17] Δ2Δ1Δ2
EXEC[17] Δ2Δ3Δ3
EXEC[17] Δ13Δ1Δ4
EXEC[17] Δ15Δ1Δ1
EXEC[17] Δ15Δ1Δ1
EXEC[17] Δ12Δ1Δ1
EXEC[17] Δ15Δ1Δ4
EXEC[17] Δ12Δ1Δ4
EXEC[17] Δ15Δ1Δ4
EXEC[17] Δ13Δ1Δ1
EXEC[17] Δ6Δ1Δ1                    EXEC[17] Δ10Δ1Δ4
EXEC[17] Δ5Δ1Δ1                    EXEC[17] Δ15Δ1Δ4
EXEC[17] Δ2Δ1Δ1                    EXEC[17] Δ12Δ1Δ4
EXEC[17] Δ14Δ1Δ4                   EXEC[17] Δ16Δ1Δ4
EXEC[17] Δ16Δ1Δ1                   EXEC[17] Δ12Δ1Δ4
EXEC[17] Δ15Δ1Δ1                   EXEC[17] Δ10Δ1Δ1
EXEC[17] Δ12Δ1Δ1                   EXEC[17] Δ11Δ1Δ4
EXEC[17] Δ15Δ1Δ4                   EXEC[17] Δ7Δ1Δ1
EXEC[17] Δ12Δ1Δ4                   EXEC[17] Δ4Δ1Δ1
EXEC[17] Δ16Δ1Δ4                   EXEC[17] Δ2Δ1Δ1
EXEC[17] Δ14Δ1Δ1                   EXEC[17] Δ1Δ1Δ1
EXEC[17] Δ6Δ1Δ1                    EXEC[17] Δ2Δ1Δ2
EXEC[17] Δ4Δ1Δ1                    EXEC[17] Δ2Δ1Δ2
EXEC[17] Δ2Δ1Δ1                    EXEC[17] Δ1Δ1Δ2
```

```
.....PROGRAMME.....

*DEBUT DU PROG
PROG  START 0
 USING PROG,15
*AD. DES LABELS
     LA   2,I3
     ST   2,A13
*INSTRUCTIONS
    MVC   A1(4),A4
    MVC   A7(4),A12
I3    EQU *
     L   2,A7
     A   2,A1
     ST   2,B1
    MVC   A7(4),B1
     L   2,A1
     S   2,A19
     ST   2,B2
    MVC   A1(4),B2
     L   2,A1
     C   2,A12
     LA   2,0
     BNH   *+8
     LA   2,1
     LTR 2,2
     L   2,A13
     BCR   7,2
FIN   BR   14
*ZONE DE DONNEES
A4    DC   F'100'
A1    DS   F
A12    DC   F'0'
A7    DS   F
A13    DS   F
B1    DS   F
B2    DS   F
A19    DC   F'1'
B3    DS   F
     END

.....NOMBRE INSTRUCTIONS.....

5

ooo TEMPS ANALYSE SEMANTIQ :0 6 28
```

Résultat

Le temps réel est d'environ 6 20

La différence est due au stop, à la fonction ETAT et à la trace de EXEC.

```
    SUBMIT  PG

  38 RECORDS SAVED IN ASM
```

La fonction SUBMIT permet de mettre la matrice *PG* qui contient le programme dans un fichier sur disque TSIO.AAAAAFHL.ASM .

Ensuite une procédure KGMOVE permet de copier ce fichier sur un disque TSO sous le nom KGO.SUBMITASM

```
/*JOB=KGMOVE,ACCT=(CISI                    ),XEQ=S91E,ALL=PIF,CASE91
//          JOB      MSGLEVEL=(1,1),REGION=60K,EXCP=9998,TIME=(,36)
//TOTSO001 EXEC     PGM=IEHMOVE,REGION=60K
//DD3   DD  UNIT=2314,DISP=SHR,VOL=SER=APLSV3
//DD2       DD      UNIT=2314,DISP=SHR,VOL=SER=TSO001
//SYSPRINT DD       SYSOUT=A
//SYSIN     DD      *
 COPY DSNAME=TSIO.AAAAAFHL.ASM,FROM=2314=APLSV3,TO=2314=TSO001,
                RENAME=KG@.SUBMIT.ASM
/*
//SYSUT1    DD      UNIT=2314,VOL=SER=TSO001,DISP=SHR
//
```

Ensuite sous TSO on assemble et on fait l'édition de lien. Puis sous le contrôle du programme TEST on exécute le module, jusqu'à l'adresse FIN où l'on fait imprimer le contenu de SOMME qui contient la somme des 100 premiers nombres.

```
asm submit test list print(*) load(som)
```

```
 LEVEL=G       RELEASE=20SEP71       SYSTEM=MVT21
 ASSEMBLER OPTIONS=CS,ALGN,LIST,LOAD,TEST,BATCH,EXTEN,NOESD,
                  NOXREF,UPLIST,EXTIME=5,UTBUFF=3,INSTSET=1
```

```
   LOC  OBJECT CODE     ADDR1 ADDR2  STMT   SOURCE STATEMENT
                                        1 *DEBUT DU PROG
 000000                                 2 PROG  START 0
 000000                                 3  USING PROG,15
                                        4 *AD. DES LABELS
 000000 4120 F014             00014     5      LA  2,I3
 000004 5020 F068             00068     6      ST  2,A13
                                        7 *INSTRUCTIONS
 000008 D203 F05C F058 C005C 00058      8      MVC  A1(4),A4
 00000E D203 F064 F060 00004 00060      9      MVC  A7(4),A12
 000014                                10 I3    EQU *
 000014 5820 F064             00064    11      L  2,A7
 000018 5A20 F05C             0005C    12      A  2,A1
 00001C 5020 F06C             0006C    13      ST  2,B1
 000020 D203 F064 F06C 00064 0006C     14      MVC  A7(4),B1
 000026 5820 F05C             0005C    15      L  2,A1
 00002A 5B20 F074             00074    16      S  2,A19
 00002E 5020 F070             00070    17      ST  2,B2
 000032 D203 F05C F070 0005C 00070     18      MVC  A1(4),B2
 000038 5820 F05C             0005C    19      L  2,A1
 00003C 5920 F060             00060    20      C  2,A12
 000040 4120 0000             00000    21      LA  2,0
 000044 47D0 F04C             0004C    22      BNH  *+8
 000048 4120 0001             00001    23      LA  2,1
 00004C 1222                           24      LTR 2,2
 00004E 5820 F068             00068    25      L  2,A13
 000052 07F2                           26      BCR  7,2
 000054 07FE                           27 FIN   BR  14
                                       28 *ZONE DE DONNEES
 000056 0000
 000058 00000064                       29 A4    DC  F'100'
 00005C                                30 A1    DS  F
 000060 00000000                       31 A12    DC  F'0'
 000064                                32 A7    DS  F
 000068                                33 A13    DS  F
 00006C                                34 B1    DS  F
 000070                                35 B2    DS  F
 000074 00000001                       36 A19    DC  F'1'
 000078                                37 B3    DS  F
                                       38      END
```

```
link som load(som100) print(*) test

F88-LEVEL LINKAGE EDITOR OPTIONS SPECIFIED TEST
          DEFAULT OPTION(S) USED -  SIZE=(122880,38912)
           SYS00008 DEFAULT BLOCKING USED   1 - 1
****TEMPNAME  NOW ADDED TO DATA SET

READY
```

```
time
Session en cours  - Session=12mn 37,16s Presence=43,59s CPU consomme=03,36s
!!
READY
```

```
test som100
TEST
at fin (1 a7;go;end)
TEST
go
AT FIN                                          +/1100
A7  +5050                                       5050
PROGRAM UNDER TEST HAS TERMINATED NORMALLY+
READY
```

```
time
Session en cours  - Session=13mn 29,62s Presence=46,63s CPU consomme=03,62s
```

Voici un deuxième exemple :

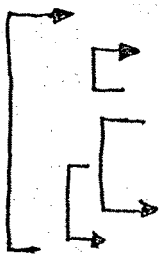
Calcul de somme alternée

$$\text{Résultat} \leftarrow \underbrace{1}_{I \rightarrow 1} - \underbrace{(2+3)}_{2} + \underbrace{(4+5+6)}_{3} - \underbrace{(7+8+9+10)}_{4} + \underbrace{(11+12+13+14+1}_{5}$$

jusqu'à I = 100

18 instructions du mini langage

107 instructions ASM générées



```
GO
: RESULT←0;I←0;U←0;TEST←1;
: BC1:I←I+1;SOMPAR←0;J←0;
: BC2:J←J+1;U←U+1;SOMPAR←SOMPAR+U;
: J<I→BC2;
: TEST=0→BC3;
: RESULT←RESULT+SOMPAR;TEST←0;
: TEST=0→S1;
: BC3:RESULT←RESULT-SOMPAR;TEST←1;
: S1:100>I→BC1  FIN
```

```
... TEMPS ANALYSE LEXICALE :0 6 44

ooo TEMPS ANALYSE SYNTAXIQ :0 11 17
```

```
*DEBUT DU PROG
PROG  START 0
 USING PROG,15
*AD. DES LABELS
     LA  2,I5
     ST  2,A15
     LA  2,I8
     ST  2,A25
     LA  2,I16
     ST  2,A28
     LA  2,I18
     ST  2,A31
*INSTRUCTIONS
     MVC  A1(4),A7
     MVC  A8(4),A7
     MVC  A9(4),A7
     MVC  A10(4),A14
I5    EQU *
     L  2,A8
     A  2,A14
     ST  2,B1
     MVC  A8(4),B1
     MVC  A18(4),A7
     MVC  A24(4),A7
I8    EQU *
     L  2,A24
     A  2,A14
     ST  2,B2
     MVC  A24(4),B2
     L  2,A9
     A  2,A14
     ST  2,B3
     MVC  A9(4),B3
     L  2,A18
     A  2,A9
     ST  2,B4
     MVC  A18(4),B4
     L  2,A24
     C  2,A8
     LA  2,0
     BNL  *+8
     LA  2,1
     LTR 2,2
     L  2,A25
     BCR  7,2
     L  2,A10
     C  2,A7
     LA  2,0     Résultat dans  A1
     BNE  *+8
     LA  2,1
     LTR 2,2
     L  2,A28
     BCR  7,2
     L  2,A1
     A  2,A18
     ST  2,B7
     MVC  A1(4),B7
     MVC  A10(4),A7
     L  2,A10
     C  2,A7
     LA  2,0
     BNE  *+8
     LA  2,1
     LTR 2,2
     L  2,A31
     BCR  7,2
I16   EQU *
     L  2,A1
     S  2,A18
     ST  2,B9
     MVC  A1(4),B9
     MVC  A10(4),A14
I18   EQU *
     L  2,A33
     C  2,A8
     LA  2,0
     BNH  *+8
     LA  2,1
     LTR 2,2
     L  2,A15
     BCR  7,2
```

```
*ZONE DE DONNEE
A7    DC  F'0'
A1    DS  F
A8    DS  F
A9    DS  F
A14   DC  F'1'
A10   DS  F
A15   DS  F
B1    DS  F
A18   DS  F
A24   DS  F
A25   DS  F
B2    DS  F
B3    DS  F
B4    DS  F
B5    DS  F
B6    DS  F
A28   DS  F
B7    DS  F
B8    DS  F
A31   DS  F
B9    DS  F
B10   DS  F
A33   DC  F'10
      END
```

```
   LOC  OBJECT CODE     ADDR1 ADDR2  STMT   SOURCE STATEMENT
                                        1 *DEBUT DU PROG
000000                                  2 PROG  START 0
000000                                  3  USING PROG,15
                                        4 *AD. DES LABELS
000000 4120 F038               00038    5      LA  2,I5
000004 5020 F148               00148    6      ST  2,A15
000008 4120 F056               00056    7      LA  2,I8
00000C 5020 F158               00158    8      ST  2,A25
000010 4120 F0F8               000F8    9      LA  2,I16
000014 5020 F170               00170   10      ST  2,A28
000018 4120 F110               00110   11      LA  2,I18
00001C 5020 F17C               0017C   12      ST  2,A31
                                       13 *INSTRUCTIONS
000020 D203 F134 F130 00134 00130      14      MVC  A1(4),A7
000026 D203 F138 F130 00138 00130      15      MVC  A8(4),A7
00002C D203 F13C F130 0013C 00130      16      MVC  A9(4),A7
000032 D203 F144 F140 00144 00140      17      MVC  A10(4),A14
000038                                 18 I5    EQU *
000038 5820 F138               00138   19      L  2,A8
00003C 5A20 F140               00140   20      A  2,A14
000040 5020 F14C               0014C   21      ST  2,B1
000044 D203 F138 F14C 00138 0014C      22      MVC  A8(4),B1
00004A D203 F150 F130 00150 00130      23      MVC  A18(4),A7
000050 D203 F154 F130 00154 00130      24      MVC  A24(4),A7
000056                                 25 I8    EQU *
000056 5820 F154               00154   26      L  2,A24
00005A 5A20 F140               00140   27      A  2,A14
00005E 5020 F15C               0015C   28      ST  2,B2
000062 D203 F154 F15C 00154 0015C      29      MVC  A24(4),B2
000068 5820 F13C               0013C   30      L  2,A9
00006C 5A20 F140               00140   31      A  2,A14
000070 5020 F160               00160   32      ST  2,B3
000074 D203 F13C F160 0013C 00160      33      MVC  A9(4),B3
00007A 5820 F150               00150   34      L  2,A18
00007E 5A20 F13C               0013C   35      A  2,A9
000082 5020 F164               00164   36      ST  2,B4
000086 D203 F150 F164 00150 00164      37      MVC  A18(4),B4
00008C 5820 F154               00154   38      L  2,A24
000090 5920 F138               00138   39      C  2,A8
000094 4120 0000               00000   40      LA  2,0
000098 47B0 F0A0               000A0   41      BNL  *+8
00009C 4120 0001               00001   42      LA  2,1
0000A0 1222                            43      LTR 2,2
0000A2 5820 F158               00158   44      L  2,A25
0000A6 0772                            45      BCR  7,2
0000A8 5820 F144               00144   46      L  2,A10
0000AC 5920 F130               00130   47      C  2,A7
0000B0 4120 0000               00000   48      LA  2,0
0000B4 4770 F0BC               000BC   49      BNE  *+8
0000B8 4120 0001               00001   50      LA  2,1
0000BC 1222                            51      LTR 2,2
0000BE 5820 F170               00170   52      L  2,A28
0000C2 0772                            53      BCR  7,2
0000C4 5820 F134               00134   54      L  2,A1
0000C8 5A20 F150               00150   55      A  2,A18
```

```
   LOC  OBJECT CODE     ADDR1 ADDR2  STMT    SOURCE STATEMENT
0000CC 5020 F174                00174    56      ST  2,B7
0000D0 D203 F134 F174 00134 00174    57      MVC  A1(4),B7
0000D6 D203 F144 F130 00144 00130    58      MVC  A10(4),A7
0000DC 5820 F144                00144    59      L  2,A10
0000E0 5920 F130                00130    60      C  2,A7
0000E4 4120 0000                00000    61      LA  2,0
0000E8 4770 F0F0                000F0    62      BNE  *+8
0000EC 4120 0001                00001    63      LA  2,1
0000F0 1222                              64      LTR 2,2
0000F2 5820 F17C                0017C    65      L  2,A31
0000F6 0772                              66      BCR  7,2
0000F8                                   67 I16    EQU *
0000F8 5820 F134                00134    68      L  2,A1
0000FC 5B20 F150                00150    69      S  2,A18
000100 5020 F180                00180    70      ST  2,B9
000104 D203 F134 F180 00134 00180    71      MVC  A1(4),B9
00010A D203 F144 F140 00144 00140    72      MVC  A10(4),A14
000110                                   73 I18    EQU *
000110 5820 F188                00188    74      L  2,A33
000114 5920 F138                00138    75      C  2,A8
000118 4120 0000                00000    76      LA  2,0
00011C 47D0 F124                00124    77      BNH  *+8
000120 4120 0001                00001    78      LA  2,1
000124 1222                              79      LTR 2,2
000126 5820 F148                00148    80      L  2,A15
00012A 0772                              81      BCR  7,2
00012C 07FE                              82 FIN   BR  14
                                         83 *ZONE DE DONNEES
00012E 0000
000130 00000000                          84 A7    DC  F'0'
000134                                   85 A1    DS  F
000138                                   86 A8    DS  F
00013C                                   87 A9    DS  F
000140 00000001                          88 A14    DC  F'1'
000144                                   89 A10    DS  F
000148                                   90 A15    DS  F
00014C                                   91 B1    DS  F
000150                                   92 A18    DS  F
000154                                   93 A24    DS  F
000158                                   94 A25    DS  F
00015C                                   95 B2    DS  F
000160                                   96 B3    DS  F
000164                                   97 B4    DS  F
000168                                   98 B5    DS  F
00016C                                   99 B6    DS  F
000170                                  100 A28    DS  F
000174                                  101 B7    DS  F
000178                                  102 B8    DS  F
00017C                                  103 A31    DS  F
000180                                  104 B9    DS  F
000184                                  105 B10    DS  F
000188 00000064                         106 A33    DC  F'100'
                                        107      END

NO STATEMENTS FLAGGED IN THIS ASSEMBLY
DYNAMIC ALLOCATION ERROR CODE 0000030C
READY
```

Résultat

```
go
AT  +D6
A1   +1
AT  +10A
A1   -4
AT  +D6
A1   +11
AT  +10A
A1   -23
AT  +D6
A1   +42
AT  +10A
A1   -69
AT  +D6
A1   +106
AT  +10A
A1   -154
AT  +D6
A1   +215
AT  +10A
A1   -290
AT  +D6
A1   +381
AT  +10A
A1   -489
AT  +D6
A1   +616
AT  +10A
A1   -763
AT  +D6
A1   +932
AT  +10A
A1   -1124
AT  +D6
A1   +1341
AT  +10A
A1   -1584
AT  +D6
A1   +1855
AT  +10A
A1   -2155
AT  +D6
A1   +2486
  .
  .
  .
AT  +10A
A1   -224664
AT  +D6
A1   +231721
AT  +10A
A1   -238924
AT  +D6
A1   +246275
AT  +10A
A1   -253775
PROGRAM UNDER TEST HAS TERMINATED NORMALLY+
TEST
end
READY
```

```
      GO
1
¯4
11
¯23
42
¯69
106
¯154
215
¯290
381
¯489
616
¯763
932
¯1124
1341
¯1584
1855
¯2155
2486
  .
  .
  .
¯224664
231721
¯238924
246275
¯253775
```

Fonction APL qui a permis la vérification

```
     ∇ GO
[1]    RESULT←0
[2]    I←0
[3]    N←0
[4]    TEST←1
[5]  BC1:I←I+1
[6]    SOMPAR←0
[7]    J←0
[8]  BC2:J←J+1
[9]    N←N+1
[10]   SOMPAR←SOMPAR+N
[11]   →BC2 IF J<I
[12]   →BC3 IF TEST=0
[13]   ⎕←RESULT←RESULT+SOMPAR
[14]   TEST←0
[15]   →S1
[16] BC3:⎕←RESULT←RESULT-SOMPAR
[17]   TEST←1
[18] S1:→BC1 IF 100>I
     ∇
```

TRAITEMENT DES ERREURS

C'est un des problèmes les plus difficiles car le moins bien formalisé. Les solutions adoptées dans ce domaine sont très souvent empiriques et très simples (message et fin de traitement). Dans notre système, nous nous sommes limités à incorporer l'architecture nécessaire à l'appel du superviseur d'erreur et au retour du contrôle. Les différents traitements particuliers à chaque type d'erreur sont à écrire par l'utilisateur. Des fonctions standard sont fournies pour indiquer les erreurs lexicales et syntaxiques (ERLX et ERSY).

Le superviseur d'erreur est la fonction ERREUR. Lors de son appel par un programme du système ou écrit par l'utilisateur, elle a deux arguments, le n° de l'action à entreprendre et le nom de la fonction interrompue. Il faut construire une matrice des actions ΔERR. Ces actions vont du simple message, à l'exécution d'un programme complexe qui tente de récupérer l'erreur. Quand une étape est erronée, il faut pouvoir sauter les étapes suivantes. C'est le rôle de la variabl de contrôle VAL. Lors de l'édition du message d'erreur, si le système est en mode TEST (variable TEST ← 1) on peut taper "?" pour examiner le problème. Pour cela, il faut bien connaître le système, le problème et le langage hôte (APL

```
                    ∇ERREUR[□]∇

     ∇ N ERREUR FCT;L
[1]    VAL←0
[2]    □←L←⍕ΔERR[N-8;]
[3]    →S1 IF~TEST
[4]    →0 IF '?'≠1↑(ρL)↓□
[5]    'ERREUR FONCTION ',FCT
[6]   B:□←L←'   QUE FAIRE ? :
[7]    →0 IF 0=ρL←(ρL)↓□
[8]    ⍕L
[9]    →B
[10] S1:SL,CR
[11]   →0
[12]   ERR
     ∇
```

mode test ?
mode débug si "?"

retour chariot et saut de ligne →

```
                    ΔERR

ERR
'BRANCHEMENT VERS LABEL INEXISTANT'
'CODE 11 LIBRE'
'CODE 12 LIBRE'
'CODE 13 LIBRE'
'CODE 14 LIBRE'
ERSY
'CODE 16 LIBRE'
'CODE 17 LIBRE'
ERSY
'CODE 19 LIBRE'
'CODE 20 LIBRE'
'CODE 21 LIBRE'
'CODE 22 LIBRE'
ERLX
```

matrice des actions
simple message →

fonction ersy →

```
                    ∇ERLX[□]∇

     ∇ Z←ERLX
[1]    Z←'ERREUR LEXICOGRAPHIQUE'
[2]    '        ',TEXT
[3]    '        ',((PT-1)ρ' '),'^'
     ∇
```

fonction standard
pour l'analyse
lexicographique et
syntaxique.

```
                    ∇ERSY[□]∇

     ∇ Z←ERSY
[1]    Z←'ERREUR DE SYNTAXE'
[2]    DTV[¯1↓TEXT[1;]]
[3]    ((G-1)ρ' '),'^'
     ∇
```

Voici une fonction utilitaire pour avoir l'arbre syntaxique sous une forme plus explicite :

N = 1 quand l'arbre est complet 0 sinon

```
     ∇ Z←ETAT N
[1]    ''
[2]    H←1↑ρΔARBRE
[3]    DVAL←''
[4]    I←0
[5]   B:→SQ IF H<I←I+1
[6]    DVAL←DVAL CAT ' ',LIRE ΔVAL[I;1]
[7]    →B
[8]   SQ:→SQ IF N
[9]    DVAL←⊖DVAL
[10] SQ:'REGLE   NOEUD  PERE   FILS      VALEUR'
[11]   (⍕(H,1)ρSAN[ΔARBRE[;1]]) CAC '       ' CAC(NUMER⍳H)
       CAC ' ' CAC(4 0 ⍕ 0 1 ↓ΔARBRE) CAC '       ' CAC DVAL
     ∇
```

On peut l'utiliser en conjonction avec EDIT (voir constructeur syntaxique) comme dans l'exemple 2.

EXEMPLE 1

erreur lexicographique.

Une variable ne peut pas commencer par Δ .

L'analyseur lexicographique a compris Δ RT ← 10
↑
symbole spécial non défini.

La solution est de le supprimer. On a alors RT ← 10 qui est valide.

AΔB ← 5 aurait nécessité la suppression de AΔ ou ΔB

car A Δ B ← 5
↘
A B ← 5

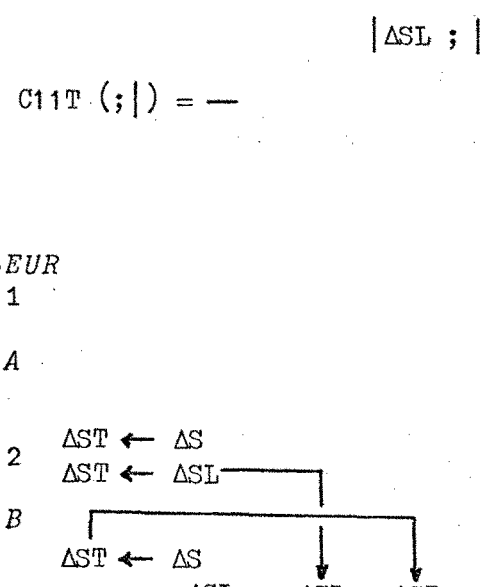
est alors faux syntaxiquement.

```
      TEST
1
      GO
   : ΔRT←10   FIN

           ΔRT←10
              ^
ERREUR LEXICOGRAPHIQUE?
ERREUR FONCTION SCANNER
   QUE FAIRE ? :  TEXT⍳'Δ'
5
   QUE FAIRE ? :  TEXT[5]←' '
   QUE FAIRE ? :  VAL←1
   QUE FAIRE ? :
... TEMPS ANALYSE LEXICALE :0 0 23
ooo TEMPS ANALYSE SYNTAXIQ :0 0 36
......PROGRAMME......

*DEBUT DU PROG
PROG  START 0
 USING PROG,15
*AD. DES LABELS
*INSTRUCTIONS
   MVC  A1(4),A3
FIN  BR  14
*ZONE DE DONNEES
A3   DC  F'10'
A1   DS  F
     END

......NOMBRE INSTRUCTIONS.....

1

ooo TEMPS ANALYSE SEMANTIQ :0 1 3
```

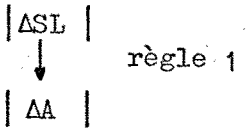
```
      GO
  : A←1;B←2; FIN
... TEMPS ANALYSE LEXICALE :0 0 31
V←C;V←C;
        ^
ERREUR DE SYNTAXE?
ERREUR FONCTION STAKING
   QUE FAIRE ? :  STAK
 10 21  5
  0  0  0
   QUE FAIRE ? :  S,G
3 9
   QUE FAIRE ? :  TEXT
  2  6  1  5  2  6  1  5 10
  1  0  2  0  3  0  4  0  0
   QUE FAIRE ? :  DICO[;1]←';'
5
   QUE FAIRE ? :  ETAT 0

REGLE   NOEUD  PERE   FILS      VALEUR
 16      [1]     2    0   0       1
 12      [2]     3    1   0
  6      [3]     4    2   0       A
  4      [4]     5    3   0
  3      [5]    10    4   0
 16      [6]     7    0   0       2
 12      [7]     8    6   0
  6      [8]     9    7   0       B
  4      [9]    10    8   0
  2      [10]    0    5   9
   QUE FAIRE ? :  ΔPROD[2 4;] EDIT 1

****  PRODUCTIONS     *******

  1      ΔSL          ::= ΔSL ; ΔST
  2      ΔST          ::= ΔS
   QUE FAIRE ? :  TEXT←1 1 1 1 1 1 1 0 1/TEXT
   QUE FAIRE ? :  STAK←1 1 0/STAK
   QUE FAIRE ? :  S←2
   QUE FAIRE ? :  G←8
   QUE FAIRE ? :  VAL←1
   QUE FAIRE ? :
ooo TEMPS ANALYSE SYNTAXIQ :0 1 48
......PROGRAMME......

*DEBUT DU PROG
PROG  START 0
 USING PROG,15
*AD. DES LABELS
*INSTRUCTIONS
   MVC  A1(4),A2
   MVC  A3(4),A4
FIN  BR  14
*ZONE DE DONNEES
A2   DC  F'1'
A1   DS  F
A4   DC  F'2'
A3   DS  F
     END

......NOMBRE INSTRUCTIONS.....

2


ooo TEMPS ANALYSE SEMANTIQ :0 1 54
```

EXEMPLE 2

erreur de syntaxe



on supprime le ';'



Le traitement effectué en mode débug peut être transformé en fonction de récupération de l'erreur. Voici comment :

```
     ∇ Z←ERTST
[1]    Z←'POINT VIRGULE INUTILE APRES LA DERNIERE INSTRUCTI
       ON'
[2]    ''
[3]    →ER IF~(¯1↑STAK[1;]=5)∧G=¯1↑ρTEXT
[4]    S←S-1
[5]    STAK← 0 ¯1 ↓STAK
[6]    VAL←1
[7]    →0
[8]   ER:Z←ERSY
     ∇
```

```
      TEST
0
      GO
  : A←1;B←2; FIN
... TEMPS ANALYSE LEXICALE :0 0 31

POINT VIRGULE INUTILE APRES LA DERNIERE INSTRUCTION


ooo TEMPS ANALYSE SYNTAXIQ :0 1 3
......PROGRAMME......

*DEBUT DU PROG
PROG  START 0
 USING PROG,15
*AD. DES LABELS
*INSTRUCTIONS
   MVC  A1(4),A2
   MVC  A3(4),A4
FIN  BR  14
*ZONE DE DONNEES
A2   DC  F'1'
A1   DS  F
A4   DC  F'2'
A3   DS  F
     END

......NOMBRE INSTRUCTIONS.....

2


ooo TEMPS ANALYSE SEMANTIQ :0 1 52
```

On peut aussi prévoir un traitement des erreurs sémantiques. Dans l'exemple du mini compilateur, il serait utile de tester si un branchement se fait vers une étiquette définie. Pour cela on crée par exemple deux variables globales supplémentaires LD et LU étiquette définie et étiquette utilisées.

La fonction BRNC fournit les étiquettes utilisées tandis que LABEL fournit les étiquettes définies. En fin de génération, lors de l'exécution de PROG on teste si LU ⊂ LD sinon il y a erreur.

```
      ∇BRNC[□]∇
    ∇ Z←A BRNC B;L
[1]   Z←'     L  2,',A
[2]   Z←Z ,' '     LTR 2,2'
[3]   Z←Z ,' '     L  2,',B
[4]   Z←Z ,' '     BCR  7,2'
[5]   LU←LU ,B
    ∇

      ∇LABEL[□]∇

    ∇ Z←A LABEL B;L
[1]   CSCT←CSCT ,' '     LA  2,I',⍕B
[2]   CSCT←CSCT ,' '     ST  2,',A
[3]   Z←(1,⍴Z)⍴Z←'I',(⍕B),'   EQU * '
[4]   LD←LD ,A
    ∇
```

```
   ∇ Z←PROG A
[1]   →S1 ⍱/(,COD LU)∈,COD LD
[2]   10 ERREUR 'LABELS'
[3]   →Z←0 ⍱VAL
[4]  S1:Z←INIT ,CSCT ,' '*INSTRUCTIONS' ,A ,
       FINAL ,DSCT ,' '    END'
   ∇
```

```
      TEST
0
      GO
  : A←1;A=2→NULPART; FIN
... TEMPS ANALYSE LEXICALE :0 0 39

POINT VIRGULE INUTILE APRES LA DERNIERE INSTRUCTION

ooo TEMPS ANALYSE SYNTAXIQ :0 1 22
·BRANCHEMENT VERS LABEL INEXISTANT

ooo TEMPS ANALYSE SEMANTIQ :0 2 8
```

Bien sûr les solutions présentées ne sont pas les meilleures, mais le but était de montrer par des exemples simples les possibilités et la souplesse du système. On peut même aller jusqu'à concevoir une génération automatique des fonctions de traitement des erreurs et des messages. On peut aussi, nous l'avons vu, de manière empirique améliorer le système (cas du dernier point virgule).