

ACTES  
DES JOURNÉES FRANCOPHONES  
SUR LA PRODUCTION ASSISTÉE DE LOGICIEL

GENÈVE, 10 & 11 JANVIER 1980

Organisées par les Universités de GENEVE et GRENOBLE  
sous l'égide de: SCACM, ASSPA, CNRS, AFCET, IRIA

*Editeurs: Bernard Amy & Christian Pellegrini*

# GENÉRATION AUTOMATIQUE D'ANALYSEURS SYNTAXIQUES AVEC RATTRAPAGE D'ERREURS

P. Boullier

## 1. - Introduction

Un certain nombre d'outils d'aide à la production automatique de traducteurs ont été développés. Ces outils (compilateurs de compilateurs, métacompileurs, générateurs de compilateurs, ou système d'écriture de traducteurs) produisent un traducteur (compilateur, interpréteur) à partir de la spécification d'un langage source et d'une machine objet. Les spécifications d'entrée de tels systèmes peuvent contenir :

- une description de la structure syntaxique et lexicale du langage source,
- une description de ce qu'il doit être généré pour chaque construction du langage source et
- une description de la machine objet.

Dans la plupart des cas la spécification de la sémantique est effectuée par un ensemble de programmes appelés actions sémantiques ou règles sémantiques exécutées lors de la reconnaissance syntaxique (traduction dirigée par la syntaxe). Cependant certains systèmes permettent, qu'une partie toute au moins, de la spécification d'un langage soit non-procédurale. Par exemple au lieu d'écrire un programme effectuant l'analyse syntaxique, l'utilisateur écrit une grammaire libre du contexte décrivant son langage et le métocompilateur convertit automatiquement cette grammaire en un programme d'analyse syntaxique.

Les principales aides fournies par les compilateurs de compilateurs existants sont :

- a) un constructeur d'analyseur lexical et un constructeur d'analyseur syntaxique, qui sont le cœur de tout système et cela pour deux raisons :
  - bien que l'analyse syntaxique ne représente qu'une petite portion de l'ensemble du compilateur, le fait d'avoir un cadre de travail fixe pour l'analyse syntaxique est d'une grande aide pour l'organisation du compilateur tout entier,

- les phases d'analyse (lexicale et syntaxique) sont actuellement les seules pour lesquelles il existe des notations (expressions régulières, grammaires libre du contexte) qui sont :
  - . suffisamment non procédurales pour réduire, cela de façon très significative, le travail du compilateur (personne chargée de réaliser le compilateur)
  - . suffisamment générales pour pouvoir être utilisées pour un langage quelconque
  - . suffisamment développées pour permettre la génération automatique d'implantations efficaces.

De tels constructeurs produisent une réalisation fiable et rapide et le produit obtenu est aisément maintenable et modifiable.

- b) un moyen pour "accrocher" des actions "sémantiques" écrites par l'utilisateur, actions qui seront appelées au bon moment par l'analyseur syntaxique généré automatiquement.

Dans les compilateurs de compilateurs on recherche actuellement des méthodes de description de la sémantique du langage source et de la machine objet présentant les mêmes avantages que les méthodes permettant de décrire les structures lexicales et syntaxiques du langage source.

Ce papier décrit SYNTAX la partie du méta-compilateur, écrit à l'IRIA-LABORIA, générant automatiquement des analyseurs lexico-syntaxiques à partir de leur description non-procédurale.

Le système SYNTAX (fig. 1) est constitué de plusieurs modules :

- INTRO module acceptant en entrée la description de la syntaxe du langage (grammaire libre du contexte).
- IOLCL module acceptant en entrée la description des unités lexicales du langage (expression régulières) et la liste des terminaux de la grammaire (fournie par INTRO) et produisant un automate d'états finis.

**Les constructeurs syntaxiques**

L'utilisateur a le choix entre 4 constructeurs syntaxiques différents, choix dépendant de la complexité (classe) de la grammaire décrivant son langage.

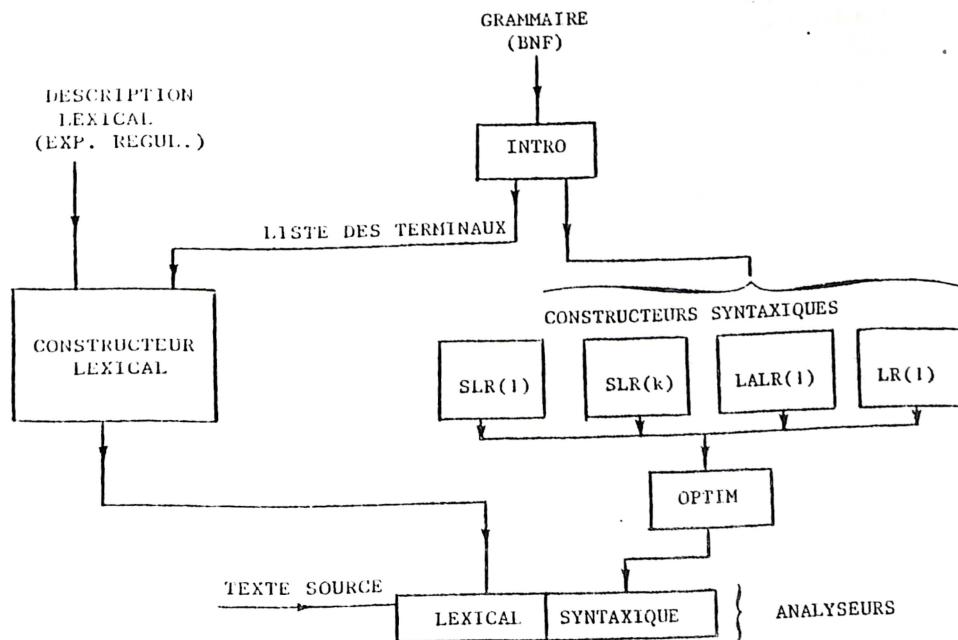
**OPTIM**

module commun aux 4 constructeurs syntaxiques et permettant d'optimiser l'automate à pile produit.

**Analyseurs syntaxique et lexical**

sont uniquement des interpréteurs des automates produits par les phases précédentes. Les interpréteurs incluent des modules permettant la correction et/ou la récupération automatique des erreurs.

Fig. 1 : SYNTAX



## 2. - Terminologie et notation

Les notions de symbole et de chaîne de symboles sont supposées connues. Un vocabulaire  $V$  est un ensemble de symboles.  $V^*$  dénote l'ensemble de toutes les chaînes de symboles qu'il est possible de construire sur  $V$ , y compris la chaîne vide qui est notée  $\epsilon$ .  $V^+$  dénote  $V^* - \{\epsilon\}$ . La longueur d'une chaîne  $a$  est notée  $|a|$ . Le premier symbole d'une chaîne non vide  $a$  est noté First ( $a$ ).

Si  $R$  est une relation,  $R^*$  dénote la fermeture reflexive transitive de  $R$  et  $R^+$  dénote la fermeture transitive.

Une grammaire libre du contexte est un quadruplet  $G = (T, N, S, P)$  où  $T$  est un ensemble fini de symboles terminaux,  $N$  est un ensemble fini de symboles non-terminaux tel que  $T \cap N = \emptyset$  (ensemble vide),  $S \in N$  est l'axiome et  $P$  est un sous ensemble fini de  $N \times V^*$  où  $x$  désigne le produit cartésien et  $V = T \cup N$  est appelé Vocabulaire (on dit aussi que  $T$  est le vocabulaire terminal et  $N$  le vocabulaire non-terminal). Chaque élément  $(A, a) \in P$ ,  $A \in N$ ,  $a \in V^*$  est appelé une production, on la note  $A \rightarrow a$ .  $A$  est appelé partie gauche et  $a$  partie droite.

Les notations suivantes sont utilisées :

$S, A, B, C, \dots$	$\in N$
$x, \dots$	$\in V$
$t, a, b, c, \dots$	$\in T$
$x, y, z, \dots$	$\in T^*$
$\alpha, \beta, \gamma, \delta, \dots$	$\in V^*$

La relation  $\Rightarrow$  s'énonce "produit directement par dérivation droite" et est définie sur  $V^*$  de la façon suivante

$\alpha A y \Rightarrow \alpha \beta y$  pour tout  $\alpha \in V^*$ ,  $y \in T^*$  et  $A \rightarrow \beta \in P$ .

Les fermetures transitives  $\xrightarrow{*}$  et  $\xrightarrow{+}$  s'énoncent "produit".

Si  $S \xrightarrow{*} \alpha$   $\alpha$  est appelé une forme sentencielles; si  $\alpha \in T^*$ ,  $\alpha$  est appelé une sentence. Le langage  $L(G)$  généré (décrit) par la grammaire  $G$  est l'ensemble de ses sentences, c'est-à-dire  $L(G) = \{x \mid x \in T^* \text{ et } S \xrightarrow{*} x\}$  soit  $G$  une grammaire et  $w = \alpha \beta \gamma$  une forme sentencielles, alors  $\beta$  est appelé

une phrase s'il existe un symbole non-terminal  $\alpha$  tel que :

$$S \xrightarrow{*} \alpha \alpha \gamma \text{ et } \alpha \xrightarrow{*} \beta$$

de plus  $\beta$  est appelé phrase simple si

$$S \xrightarrow{*} \alpha \alpha \gamma \text{ et } \alpha \Rightarrow \beta$$

le manche de toute forme sententielle est la phase simple la plus à gauche.

### 3. - INTRO

Les notations utilisées pour décrire la grammaire libre du contexte utilisées en entrée par INTRO sont très proches de la BNF.

La grammaire est une liste formée de productions. Chaque production se termine par le caractère ";" . Un symbole non-terminal commence par le caractère ":" et se termine par le caractère ">" .

La partie gauche et la partie droite d'une production sont séparées par le caractère "=".

Les terminaux de la grammaire sont écrits tels quels et se terminent au premier blanc rencontré, il y a cependant deux exceptions :

- si le premier caractère d'un symbole terminal est un métacaractère, on le fait précéder du caractère "#" exemple : le terminal ";" est représenté par "#"; car ; est le métacaractère marquant la fin d'une production.

- habituellement il existe deux types de terminaux les séparateurs, mots-clés,... qui lexicalement ont la même forme dans la grammaire, et dans le programme et les identifiants, nombres etc... dont le terminal au niveau syntaxique est uniquement un nom générique recouvrant un ensemble (généralement infini) de possibilité pour le programme. Par exemple le terminal "nombre-entier" d'une grammaire donnée recouvre toutes les possibilités des nombres entiers alors que "nombre-entier" n'est pas un nombre entier ("identifiant" est un identifiant). De tels terminaux "génériques" doivent obligatoirement avoir leur description au niveau lexical, on convient donc dans SYNTAX de les faire précéder du caractère "%".

Exemple

```
.
.
.
<STAT-LIST> = <STAT-LIST> <STAT> #; ;
<STAT-LIST> = ;
<STAT> = %ident := <EXPRESSION> ;
```

la partie droite de la deuxième production est vide.

Remarque

L'axiome doit être le premier non-terminal rencontré et ne doit pas apparaître en partie droite ;

INTRO lit la grammaire et la met sous forme interne. De plus INTRO effectue certains tests de cohérence (propreté) sur la grammaire :

- un non-terminal quelconque appartient à au moins une forme sententielle :

quelque soit  $A \in N$  l'on a  $S \xrightarrow{*} \alpha A y$

Cette condition assure que toute production est utile.

- un non terminal quelconque doit être productif (produit une chaîne terminale possiblement vide) :

quelque soit  $A \in N$  l'on a  $A \xrightarrow{+} y, y \in T^*$

- Aucun non-terminal ne dérive dans lui-même (condition suffisante d'ambiguité) :

quelque soit  $A \in N$  l'on n'a pas  $A \xrightarrow{+} A$ .

4. - lOCL, le constructeur lexical de SYNTAX

Il est tentant de supposer que les analyseurs lexicaux pour tous les langages sont presque les mêmes et diffèrent seulement sur quelques points particuliers (mots-clés, séparateurs,...). Beaucoup de compilateurs de compilateurs utilisent en fait des routines d'analyse lexicale fixes dans le compilateur généré, ces routines diffèrent seulement par la liste des mots-clés

reconnus, liste fournie par l'utilisateur. Cette approche peut se révéler impraticable si l'on désire reconnaître des unités lexicales non standard, par exemple des identificateurs comprenant des caractères autres que lettres ou chiffres. IOCL utilise une approche plus générale, sans pour cela augmenter le volume de la description nécessaire.

#### 4.1 - Rôle de l'analyseur lexical

l'analyseur lexical est la partie du compilateur qui est chargée de lire le programme source, caractère par caractère et de le traduire en une séquence de primitives appelées unités lexicales (mot-clé, identificateur, constante, séparateur,...).

L'analyseur lexical peut être considéré, comme une passe séparée du compilateur, plaçant dans un fichier la séquence des unités lexicales reconnues. Ce fichier servant d'entrée à l'analyseur syntaxique. Cette organisation peut être jugée préférable pour les très petites machines. Il est également possible de concevoir les analyseurs syntaxiques et lexicaux dans la même passe, l'analyseur lexical étant une (co) routine de l'analyseur syntaxique, appelée chaque fois que ce dernier a besoin d'une nouvelle unité. . lexical, bien entendu cette organisation supprime le fichier intermédiaire.

L'avantage du découpage usuel de l'analyse d'un programme source en deux phases, lexicale et syntaxique, est double.

- simplification de la conception globale du compilateur : conception séparée des deux analyseurs avec interface très simple (renseignements concernant l'unité lexicale reconnue)
  - les grammaires permettant de décrire les unités lexicales (grammaires régulières) ont des automates d'analyse plus simples et plus efficaces que les grammaires générales décrivant les langages de programmation.

#### 4.2. - Expressions régulières

C'est un symbolisme commode permettant de décrire les langages réguliers, langages auxquels appartiennent les éléments lexicaux. De plus ces expressions régulières peuvent se traduire automatiquement en automates d'états finis.

##### 4.2.1. - Définition

Définition récursive d'une expression régulière construite sur un vocabulaire  $\Sigma$

- 1)  $\epsilon$  est une expression régulière décrivant le langage  $\{\epsilon\}$
- 2) quelque soit  $a \in \Sigma$ ,  $a$  est une expression régulière décrivant le langage  $\{a\}$
- 3) Si  $R$  et  $S$  sont des expressions régulières décrivant les langages  $L_R$  et  $L_S$  alors  
 $R|S$  est une expression régulière décrivant le langage  $L_R \cup L_S$
- 4) Si  $R$  et  $S$  sont des expressions régulières décrivant le langage  $L_R$  et  $L_S$  alors  $RS$  est une expression régulière décrivant le langage  $L_R L_S$
- 5) Si  $R$  est une expression régulière décrivant le langage  $L_R$  alors  $R^*$  est une expression régulière décrivant le langage  $L_R^*$
- 6)  $R^+$  : abréviation pour  $RR^*$

##### Exemples

identifieur = lettre (lettre|chiffre)\*

Un identifieur est une chaîne commençant par une lettre et suivit d'un nombre quelconque, possiblement nul, de lettres ou de chiffres.

nombre-entier = chiffre<sup>+</sup>

chaîne = '(tout-sauf-apos|'')\*'

Une chaîne de caractères commence et se termine par une apostrophe, entre les deux on peut trouver un nombre quelconque de caractères quelconques, toutefois une apostrophe à l'intérieur de la chaîne doit être représentée par deux apostrophes consécutives.

Commentaire\_pl1 = /\* (tout-sauf-étoile|\*)+ tout sauf-étoile-sauf-slash)\*+\*/

#### 4.2.2. - Automate d'états finis non déterministes

Le passage d'une expression régulière à un automate d'états finis peut se faire de la façon suivante ; si l'on suppose que l'automate associé à un seul état initial  $i$  et un seul état final  $f$

Expression régulière

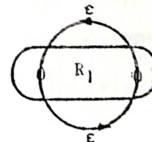
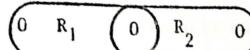
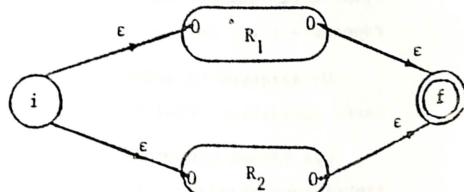
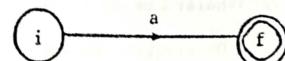
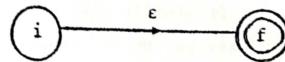
$\epsilon$

a

$R_1 | R_2$

$R_1^A$

Automate (diagramme de transition)



Dans le cas général, l'automate d'états finis ainsi produit est non déterministe (depuis un état donné il existe plusieurs transitions possibles sur un terminal donné) donc impropre à une analyse efficace. L'automate ainsi produit doit donc ensuite être rendu déterministe, ce qui est toujours possible (HOP69-ALL72).

La méthode utilisée dans 10CL(DER74) n'est pas classique et utilise les techniques LR (cf § 5) pour obtenir un automate d'états finis déterministe, directement à partir des expressions régulières.

#### 4.3. - Génération d'automates d'états finis en utilisant les techniques LR

On appelle item une expression régulière ayant une position distinguée repérée par un ":" (appelé marqueur LR) indiquant la progression de la reconnaissance de l'expression régulière.

Un marqueur LR devant une expression régulière indique le fait qu'aucun symbole du langage décrit (généré) par cette expression régulière n'a été reconnu - c'est le cas de l'état initial.

Un marqueur LR après une expression régulière indique le fait que cette expression régulière vient d'être reconnue.

Les règles gouvernant le déplacement du marqueur LR à travers les expressions régulières (EAR 70) sont les suivantes :

1) si un item est de la forme

$$\alpha \cdot (\omega_1 | \omega_2 \dots | \omega_n) \beta$$

le remplacer par n items

$$\alpha \cdot \omega_1 | \omega_2 \dots | \omega_n \beta$$

:

$$\alpha (\omega_1 | \dots | \omega_n) \beta$$

2) si un item est de la forme

$$\alpha (\omega_1 | \dots | \omega_j \cdot | \dots | \omega_n) \beta$$

le remplacer par :

$$\alpha (\omega_1 | \dots | \omega_j | \dots | \omega_n) \cdot \beta$$

3) si un item est de la forme

$$\alpha (\omega)^* \beta$$

ou

$$\alpha (\omega)^* \beta$$

le remplacer par deux items

$$\alpha (\omega)^* \beta$$

$$\alpha (\omega)^* \beta$$

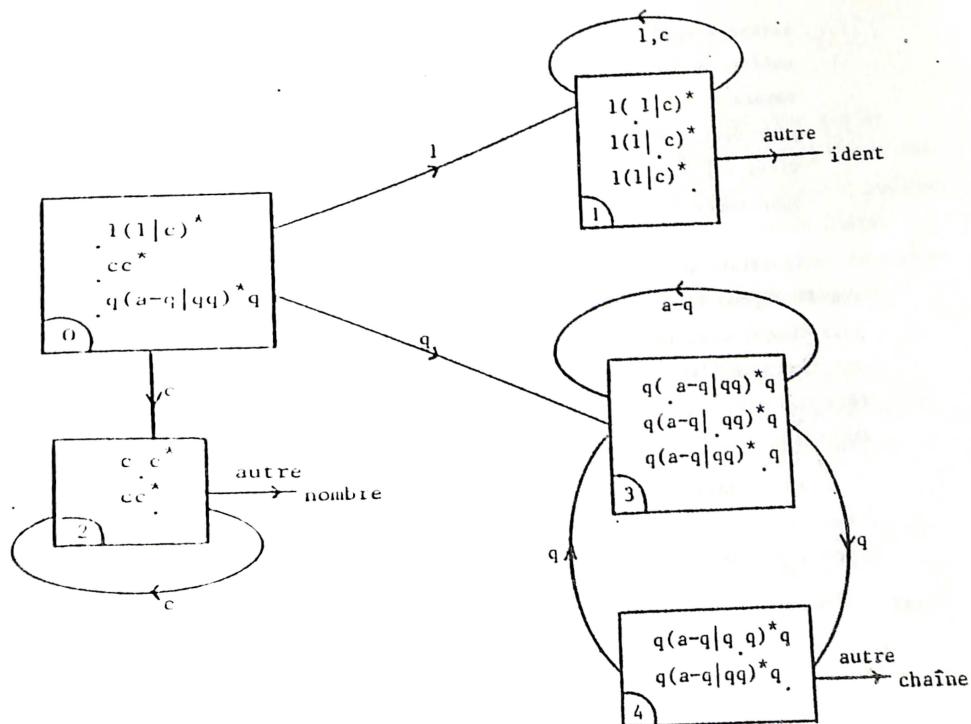
### Exemple

$$\text{ident} = 1(1|c)^*$$

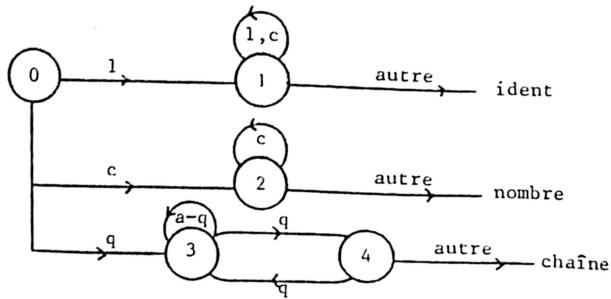
$$\text{nombre} = c\ c^*$$

$$\text{chaîne} = q(a-q|qq)^*q$$

$a-q$  signifie un caractère quelconque sauf une apostrophe.



automate



4.4. - Exemple utilisant IOCL

On définit les classes simples en regroupant tous les caractères jouant le même rôle au point de vue lexical.

Un caractère donné ne peut appartenir qu'à une seule classe simple.

lettre = /ABC... XYZ/  
chiffre = /01... 89/  
espace = / /  
pour\_cent = /%/  
tiret = /\_-/  
apostrophe = /"/

Il existe une classe simple prédéfinie EOL qui contient le caractère fin de ligne. Il est créé automatiquement par le système une classe simple, pour chaque caractère défini au niveau syntaxique (entrée de INTRO) et non redéfini par l'utilisateur au niveau lexical (ex. : +, \*, (, ...)). Tout caractère non défini (par l'utilisateur ou par le système) est un caractère interdit.

L'utilisateur peut définir alors des classes composées de caractères à partir (union d'ensemble : opérateur + et différence ensemble : opérateur -) de classes simples ou de classes composées précédemment définies.

Il existe une classe composée prédéfinie ANY qui contient tous les caractères définis. Un caractère peut appartenir à plusieurs classes composées.

```
lettre_ou_chiffre = lettre + chiffre ;
tout_sauf_eol = ANY - EOL ;
tout_sauf_apos_et_eol = tout_sauf_eol - apostrophe ;
```

L'utilisateur a la possibilité de définir alors des abréviations, ce sont des expressions régulières, nommées par l'utilisateur, chaque occurrence ultérieure du nom de l'expression régulière sera remplacée par l'expression régulière correspondante.

Le vocabulaire  $\Sigma$  des expressions régulières est :

$$\Sigma = \{\text{classes simples}\} \cup \{\text{classes composées}\} \cup \{\text{abréviations}\}$$

Les opérateurs de fermeture transitive ( $*$  et  $^+$ ) des expressions régulières sont notés en IOCL par les opérateurs préfixés  $\$$  et  $\%$ .

ainsi lettre $^*$  s'écrit  $\$$ lettre  
et chiffre $^+$  s'écrit  $\%$ chiffre.

Certains caractères reconnus au niveau lexical, n'ont plus aucune utilité dans la suite du processus de traduction. Ainsi le fait important pour l'analyseur syntaxique est de savoir que l'analyseur lexical a reconnu le symbole d'affectation, la chaîne " $:=$ " ; elle, n'a pas besoin d'être conservée. Par contre, lors de la reconnaissance d'un identifieur la chaîne de caractères le constituant doit être conservée pour des traitements ultérieurs (table des symboles).

L'utilisateur a donc la possibilité de spécifier si un caractère doit être ou non conservé par l'analyseur lexical. Si un caractère appartenant à une classe (simple ou composée) donnée doit être supprimé, ou le fait précéder du signe moins " $-$ " ;

L'utilisateur définit ensuite les commentaires de son langage. Ce sont des chaînes, reconnues par l'analyseur lexical et qui en général sont totalement éliminées par ce dernier, de plus l'analyseur syntaxique n'a pas conscience de leur présence.

%-EOL|%-espace|-pour\_cent \$-tout\_sauf\_eol -EOL ;

Dans notre langage, par exemple, les fins de lignes, les espaces et les chaînes comprises entre % et la fin de ligne sont des commentaires, tous les commentaires sont supprimés (signe - devant chaque occurrence d'une classe).

Finalement l'utilisateur défini les expressions régulières vérifiées par les terminaux génériques (précédés du caractère %) de sa grammaire.

%identifieur = lettre \$ (tiret lettre\_ou\_chiffre|lettre\_ou\_chiffre) ;

un identifieur commence par une lettre et est suivi d'un nombre quelconque de lettres ou de chiffres éventuellement précédés d'un tiret.

%nombre\_entier = %chiffre ;

%chaîne = -apostrophe \$(tout\_sauf\_apos\_et\_eol|-apostrophe apostrophe)-apostrophe ;

Une chaîne est une séquence des caractères quelconques entourée par des apostrophes. Pour représenter une apostrophe à l'intérieur d'une chaîne, on la double, de plus une chaîne ne peut contenir le caractère fin de ligne. Les apostrophes entourant une chaîne et la première apostrophe de la représentation interne d'une apostrophe étant inutiles pour la suite de la traduction on les supprime.

chaîne reconnue

"a""b"

""

chaîne conservée

a"b

(chaîne vide)

Les terminaux génériques sont les seuls qui doivent obligatoirement être définis au niveau lexical, tous les autres sont traités automatiquement par le système de la façon suivante :

Pour un terminal donné le système regarde s'il est reconnu par une expression régulière existante :

non : alors il génère automatiquement l'expression régulière le reconnaissant par exemple le terminal ":= n'est reconnu ni par %identifiant, ni par %nombre-entier ni par %chaîne le système génère donc l'expression régulière -: == reconnaissant := mais ne conserve pas la chaîne de caractères le constituant.

oui : par exemple le mot clé BEGIN est reconnu par l'expression régulière %identifiant, le système construit alors une table de hash-code contenant tous les terminaux reconnus par %identifiant. Lors de la reconnaissance par l'analyseur lexical d'un identifiant, ce dernier devra chercher dans sa table pour un mot-clé éventuel, si la recherche est positive alors le code du mot-clé reconnu est rendu à l'analyseur syntaxique sinon c'est le code d'identifiant qui lui est rendu.

L'ensemble des terminaux (généralement les mots-clé) figurant dans la table de hash-code étant connu statiquement (lors de la construction) IOCL construit une fonction de hash-code, dépendant de cet ensemble, et telle qu'elle assure la reconnaissance (ou la non reconnaissance) d'un mot-clé en une seule comparaison (SPR77), de plus le taux de remplissage de cette table est généralement proche de 100%.

#### Remarque

Lorsque l'on écrit qu'un identifiant est reconnu par l'expression régulière lettre (lettre|chiffre)\* la signification est la suivante : tant que le caractère sous la tête de lecture est une lettre, ou un chiffre, cette lettre ou ce chiffre appartient à l'identifiant que l'on est en train de reconnaître : ainsi la chaîne formée de la lettre a et du chiffre 1 est reconnue comme étant l'identifiant al et non l'identifiant a suivi du nombre entier 1. On cherche donc à reconnaître la plus longue chaîne vérifiée par une expression régulière donnée. Dans IOCL l'utilisateur a la possibilité de spécifier explicitement cette priorité et en particulier de donner priorité

à la reconnaissance sur la lecture pour cela il peut utiliser le métacaractère "." dans une expression régulière. Ce point à la signification suivante : si lors de la construction de l'automate, dans un item un marqueur LR coïncide avec le "." alors tous les items de l'état considéré dont le marqueur LR ne coïncide pas avec le "." sont supprimés.

Exemple :

Un commentaire de notre langage aurait pu s'écrire (en conservant son texte)

`pour_cent $ANY EOL.`

la signification intuitive étant la suivante :

lorsqu'une fin de ligne est rencontrée, il s'agit de la fin du commentaire (ne pas oublier que ANY contient EOL)  
alors que: `pour_cent $ANY EOL`  
aurait absorbé tout le programme jusqu'à la fin (la fin de ligne prise par défaut - priorité à la lecture - étant celle de ANY et non le EOL de la fin de l'expression régulière).

Les commentaires PL/I peuvent par exemple se décrire par :

`commentaire_PL/I = /*$ANY*/.`

(le premier "\*/" rencontré marque la fin du commentaire  
cette description doit être comparée avec celle figurant à la fin du § 4.2.1.).

## 5. - Constructeurs syntaxiques

Les bases théoriques sur lesquelles s'appuient les constructeurs syntaxiques de SYNTAX, ont été décrites par Knuth [KNU 65]. Ces constructeurs acceptent en entrée des grammaires de la famille LR et les analyseurs syntaxiques générés s'appellent des analyseurs LR.

Ces analyseurs sont appelés LR parce qu'ils lisent le texte source de la gauche vers la droite (Left-to-right) et construisent la séquence inverse de la séquence de dérivations droites (Right)-propriété caractéristique des analyseurs ascendants.

Les grammaires et les analyseurs LR ont un certain nombre de propriétés qui font de la méthode LR une méthode de choix pour l'analyse syntaxique des langages de programmation à savoir :

- les langages LR (langages ayant une grammaire LR les décrivant) constituent la classe la plus large de langages qui peuvent être analysés de façon déterministe en une passe de gauche à droite. Il est donc possible de construire des analyseurs qui reconnaissent pratiquement toutes les constructions des langages de programmation pour lesquels il est possible d'écrire une grammaire libre du contexte.
- les analyseurs LR peuvent être réalisés avec le même degré d'efficacité que d'autres méthodes, pourtant moins puissantes (méthodes de précédence, contexte-borné, méthodes LL)
- les analyseurs LR peuvent détecter les erreurs de syntaxe le plus tôt qu'il est possible de le faire lors d'une lecture gauche droite du texte source (propriété du préfixe-correct).

Informellement une grammaire est LR( $k$ ) (translatable from Left-to-right in using Right reduction with  $k$  symbols of look-ahead) si, dans toute forme sententielle, le manche peut être déterminé en utilisant la phrase réductible potentielle, tout son contexte gauche et  $k$  symboles sur sa droite.

Les constructeurs syntaxiques de SYNTAX acceptent en entrée, par ordre de puissance croissante des grammaires du type SLR(1) (Simple LR(1)), LALR(1) (Look-Ahead LR(1)), LR(1) ainsi que des grammaires SLR( $k$ ).

Jusque dans les années 70, les techniques d'analyse LR n'ont pas été aussi largement utilisées que d'autres méthodes, pourtant moins attractives sur le plan théorique. Cela est dû principalement à deux raisons :

- les premières présentations de la méthode ont laissé supposé que la théorie sous-jacente était difficilement accessible.

- et surtout, une implantation directe de la méthode de Knuth produit des automates d'analyse d'une taille "extravagante", et l'approche LR de l'analyse n'a pu s'envisager que lorsque des méthodes d'optimisation adéquates furent découvertes.

Maintenant, le LR est devenu une méthode de choix dans un très grand nombre de situations.

La raison pour laquelle on s'intéresse tout particulièrement aux grammaires de la famille LR ayant un look-ahead de 1 (SLR(1), LALR(1) et LR(1)) est double.

- on a prouvé qu'il existe une grammaire LR(1) équivalente à une grammaire LR(k) donnée, donc reconnaissant le même langage.
- très souvent, dans la pratique, lorsqu'une grammaire n'est pas LR(1) c'est qu'elle est ambiguë donc non LR(k).

## 5.1. - Automate LR(0)

### 5.1.1. - Construction

#### Items

D'une façon analogue à ce qui a été fait au § 4.3. nous définissons un item comme étant une production de la grammaire avec une position distinguée dans sa partie droite, notée par un point "." et appelé le marqueur LR. Ce marqueur LR indique la progression de la reconnaissance de la partie droite de la production considérée : la sous-chaîne située à sa gauche a été reconnue, la sous-chaîne située à sa droite n'a pas encore été reconnue. Un item avec un marqueur LR sur sa droite correspond à une réduction possible (la partie droite de la production considérée vient d'être reconnue, est-ce bien le manche ?), il est appelé item complet.

### étais

Un état est une collection d'informations indiquant le progrès de l'analyse, il est représenté par un ensemble d'items.

L'analyse LR est basée sur l'observation que pour chaque grammaire LR seulement un nombre fini d'états ont besoin d'être distingués pour permettre une analyse et que toutes les transitions (déterministes si la grammaire est LR) peuvent être calculées en avance.

### Exemple

Construction de l'automate LR(0) pour une grammaire des expressions arithmétiques simplifiée (fig. 2).

G1 :

- 0: S → E #
- 1: E → E + T
- 2: E → T
- 3: T → (E)
- 4: T → a

Une grammaire est LR(0) si dans tout état comportant un item complet, cet état est formé de ce seul item ; la grammaire G1 est LR(0).

Par contre si l'on considère la grammaire G'1.

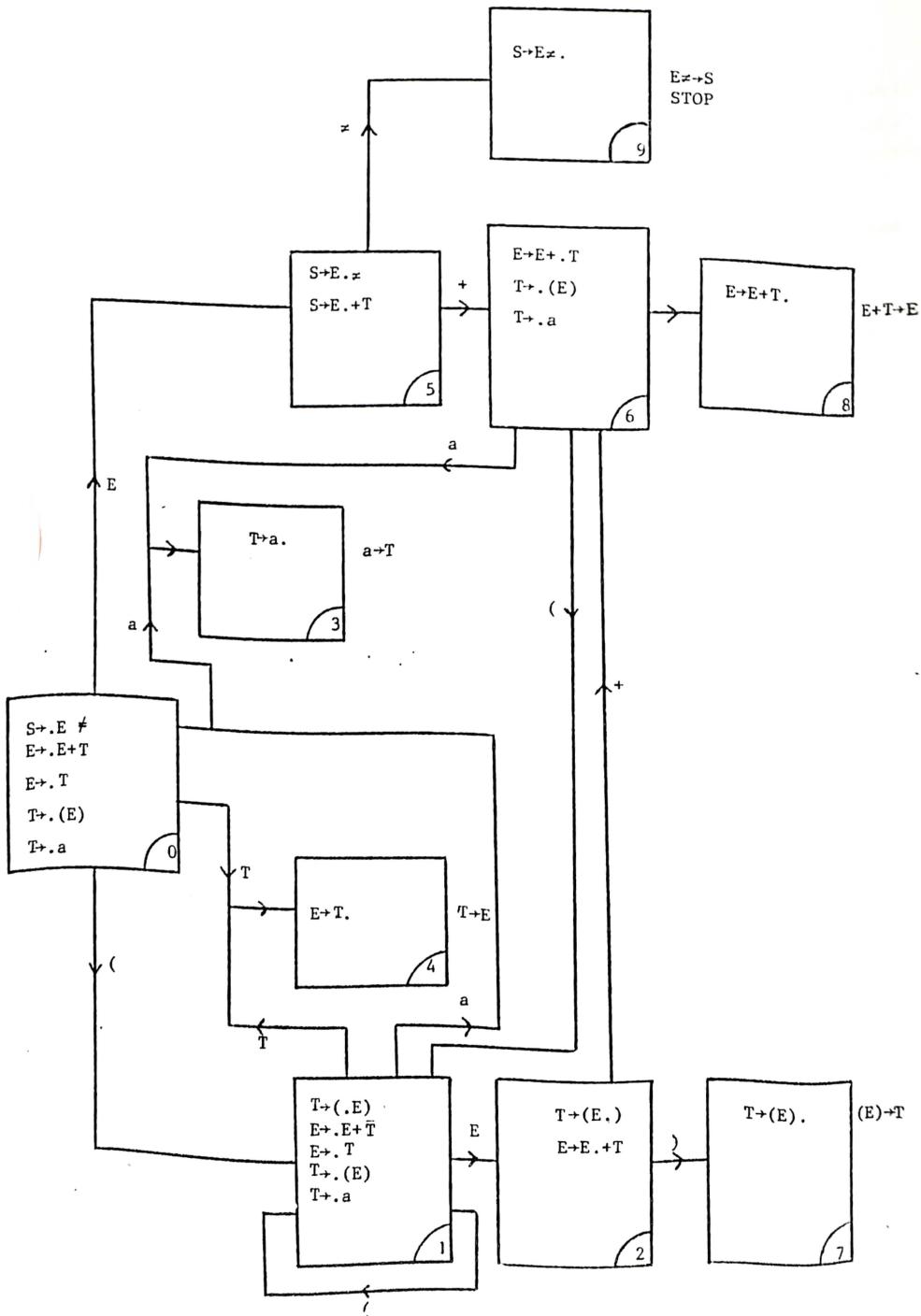
G'1 :

- 0: S → E
- 1: E → E + T
- 2: E → T
- 3: T → (E)
- 4: T → a

dans laquelle le marqueur de fin "#" de l'expression arithmétique a été supprimé ; l'état 5 devient :

$$\begin{aligned}S &= E \\E &= E_+ + T\end{aligned}$$

Fig. 2 : Automate LR(0) de la grammaire G1



qui est un état inadéquat, nous avons alors besoin de connaître un symbole en avance (look-ahead) pour pouvoir prendre une décision, à savoir : si le symbole suivant est + on passe dans l'état 6, si le symbole suivant est fin de fichier on passe dans l'état 9 (réduction vers l'axiome et fin d'analyse) sinon détection d'une erreur.

Les méthodes de construction SLR, LALR et LR dépendent de la façon dont sont construits ces ensembles de symboles en look-ahead. Pour plus de détails l'on pourra consulter [DeR 69], [AHO 72], [AHO 77].

#### 5.1.2. - Analyse syntaxique utilisant l'automate LR(0).

L'analyse d'un texte source utilisant l'automate produit s'effectue très simplement de la façon suivante :

- Partant de l'état initial 0 on effectue les transitions d'état à état, transitions gouvernées par le texte source, en mémorisant dans une pile - la pile des états - les états par lesquels on est passés.

Lorsque l'on effectue une réduction, vers un certain non terminal A, on dépile la pile des états de la longueur de la partie droite de la réduction considérée et l'on effectue depuis l'état en sommet de pile la transition sur A.

Définissons une configuration de l'automate d'analyse comme étant un triplet (pile-des-états, texte-source, réductions-effectuées) où pile des états est représenté par une séquence d'états avec le sommet à droite, texte-source est la portion du texte source non encore lu et réductions-effectuées est la liste des réductions effectuées jusqu'à présent (cette liste permet par exemple de construire l'arbre d'analyse).

Exemple                  analyse de a + (a)

configuration initiale : (0, a+(a)\*, ε)

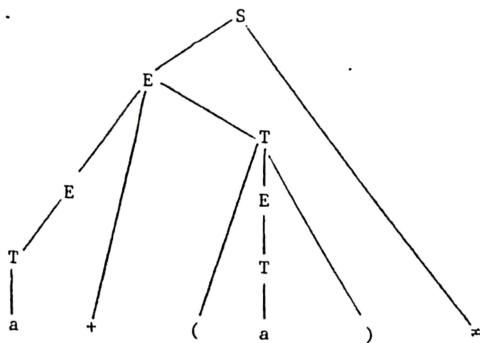
arbre vide

texte source

état initial

(0, a+(a) , ε)	→ (03, +(a)x, ε)
	→ (04, +(a)x, 4)
	→ (05, +(a)x, 42)
	→ (056, (a)x, 42)
	→ (0561, a)x, 42)
	→ (05613,)x, 42)
	→ (05614,)x, 424)
	→ (05612,)x, 4242)
	→ (056127,x, 4242)
	→ (0568,x, 42423)
	→ (05,x, 424231)
	→ (059,ε, 424231)
	→ (0,ε, 4242310)

L'arbre d'analyse correspondant est :



6. - OPTIM

L'automate d'analyse produit par les constructeurs de la famille LR est généralement représenté sous forme matricielle  $M$ . L'action à effectuer dans l'état  $s$  par transition sur le symbole  $x$  est trouvé en  $M(s,x)$ . Mais une implantation directe sous forme matricielle est impensable : typiquement pour un langage de programmation de taille moyenne l'on manipule de 100 à 200 symboles (terminaux et non terminaux) et le nombre d'états différents est de l'ordre de plusieurs centaines.

Or la matrice obtenue est en général assez creuse, c'est-à-dire que la plupart des actions correspondant à un élément  $M(s,x)$  correspond à la détection d'une erreur.

La plupart des optimisations effectuées tendent à réduire la taille de cette matrice et sont de deux ordres

- optimisations propres à l'analyse LR
- techniques de représentation des matrices creuses.

Les automates produits par les constructeurs SLR, LALR ou LR sont optimisés par le module unique OPTIM, nous allons voir les principales optimisations effectuées sur l'exemple de l'automate de la grammaire G'1 produit par le constructeur LR(1).

Un item de l'automate LR(1) est une production de la grammaire munie de son marqueur LR à laquelle on a associé un ensemble de chaînes (ensemble de look-ahead) de longueur 1 (de longueur  $k$  pour le LR( $k$ )) représentant l'ensemble des contextes droits pouvant suivre le non-terminal en partie gauche de la production considérée.

Deux états de l'automate LR(1) sont égaux ssi ils comportent les mêmes items associés au même ensemble de look-ahead. Par exemple l'état initial de la grammaire G'1 est construit de la façon suivante :

I'on part de  $S \star E, \{\epsilon\}$

Le marqueur LR devant  $E$  exprime le fait que l'on est dans l'état initial donc que l'on n'a encore rien reconnu. L'ensemble de look-ahead  $\{\epsilon\}$  exprime le fait qu'une sentence est suivie de la chaîne vide (fin de fichier dans la pratique).

Ayant le marqueur LR devant E indique que l'on s'apprête à reconnaître possiblement toutes les phrases générées par E, on positionne donc le marqueur LR devant toutes les parties droites des productions où le non terminal E intervient en partie gauche soit  $E \rightarrow E + T$  et  $E \rightarrow T$  les ensembles de look-ahead associés sont  $\{\epsilon\}$ , ensemble de chaînes pouvant suivre le non-terminal E ayant produit ces deux items l'on a donc :

$E \rightarrow E + T, \{\epsilon\}$   
 $E \rightarrow T, \{\epsilon\}$

puis l'on recommence (fermeture transitive de la relation First). L'on a un marqueur LR devant E dans  $E \rightarrow E + T$ , l'on s'apprête donc possiblement à reconnaître toutes les phrases générées par E suivies de + on ajoute donc (s'ils n'existent pas déjà) à l'état courant les items  $E \rightarrow E + T$  et  $E \rightarrow T$  avec leur contexte  $\{+\}$ .

Lorsqu'aucun autre item, ou contexte ne peut être ajouté, l'état est complet. Pour l'état initial on obtient donc :

$S \rightarrow E, \{\epsilon\}$   
 $E \rightarrow E + T, \{\epsilon, +\}$   
 $E \rightarrow T, \{\epsilon, +\}$   
 $T \rightarrow (E), \{\epsilon, +\}$   
 $T \rightarrow a, \{\epsilon, +\}$

L'automate LR(1) pour la grammaire  $G'$  est obtenu par la méthode décrite précédemment mais chaque état est partitionné en deux sous-ensembles

- l'état proprement dit qui est constitué de l'ensemble des items ayant le marqueur LR devant un non-terminal (cet ensemble peut être vide)

$S \rightarrow E, \{\epsilon\}$   
 $E \rightarrow E + T, \{\epsilon, +\}$   
 $E \rightarrow T, \{\epsilon, +\}$

- son complémentaire, appelé étiquette formé des items ayant le marqueur LR devant un terminal ou en fin de partie droite

$T \rightarrow (E), \{\epsilon, +\}$   
 $T \rightarrow a, \{\epsilon, +\}$

Ce partitionnement permet une première optimisation. Le nombre d'états et le nombre d'étiquettes ainsi obtenu est inférieur (ou égal) au

nombre d'états obtenus par la construction canonique.

On peut également constater que les transitions sur des symboles terminaux concernent uniquement les étiquettes, alors que les transitions sur des non-terminaux concernent uniquement les états (ce sont les seules informations qu'il est nécessaire d'empiler lors de l'analyse).

L'automate LR(1) pour la grammaire G'1 est représenté fig. 3, l'état initial est représenté par le couple  $S_1, E_1$ . Depuis un état ou une étiquette s la transition sur un symbole x vers  $S_i, E_j$  ( $s \xrightarrow{x} S_i, E_j$ ) s'énonce : étant en s par transition sur x on empile l'état  $S_i$  et l'on va à l'étiquette  $E_j$ .

$$E_1 : T \xrightarrow{*} (E), (\epsilon, +)$$

$$T \xrightarrow{*} a, (\epsilon, +)$$

$$\begin{matrix} E_1 \xrightarrow{a} E_2 \\ | \\ E_1 \xrightarrow{*} S_2, E_3 \end{matrix}$$

$$\begin{matrix} S_1 : S \xrightarrow{*} E, \{\epsilon\} \\ E \xrightarrow{*} E + T, \{\epsilon, +\} \\ E \xrightarrow{*} T, \{\epsilon, +\} \end{matrix}$$

$$\begin{matrix} S_1 \xrightarrow{E} E_5 \\ S_1 \xrightarrow{T} E_6 \end{matrix}$$

$$E_2 : T \xrightarrow{*} a, (\epsilon, +)$$

$$\begin{matrix} S_2 : T \xrightarrow{*} (E), \{\epsilon, +\} \\ E \xrightarrow{*} E + T, \{\}, +\} \\ E \xrightarrow{*} T, \{\}, +\} \end{matrix}$$

$$E_3 : T \xrightarrow{*} (E), \{\}, +\}$$

$$T \xrightarrow{*} a, \{\}, +\}$$

$$E_3 \xrightarrow{a} E_4$$

$$E_3 \xrightarrow{*} S_3, E_3$$

$$E_4 : T \xrightarrow{*} a, \{\}, +\}$$

$$\begin{matrix} S_2 \xrightarrow{E} E_7 \\ S_2 \xrightarrow{T} E_8 \end{matrix}$$

$$\begin{matrix} S_3 : T \xrightarrow{*} (E), \{\}, +\} \\ E \xrightarrow{*} E + T, \{\}, +\} \\ E \xrightarrow{*} T, \{\}, +\} \end{matrix}$$

$$E_5 : S \xrightarrow{*} E, \{\epsilon\}$$

$$S \xrightarrow{*} E + T, \{\epsilon, +\}$$

$$E_5 \xrightarrow{*} S_4, E_1$$

$$\begin{matrix} S_3 \xrightarrow{E} E_{10} \\ S_3 \xrightarrow{T} E_8 \end{matrix}$$

$E_6 : E \rightarrow T \cdot, \{\epsilon, +\}$

$S_4 : E \rightarrow E^+ \cdot T, \{\epsilon, +\}$

$S_4 \xrightarrow{T} \phi, E_{11}$

$E_7 : T \rightarrow (E \cdot), \{\epsilon, +\}$

$E \rightarrow E \cdot + T, \{(), +\}$

$E_7 \xrightarrow{\epsilon} \phi, E_9$

$S_5 : E \rightarrow E^+ \cdot T, \{(), +\}$

$S_5 \xrightarrow{T} \phi, E_{12}$

$E_7 \xrightarrow{+} S_5, E_3$

$E_8 : E \rightarrow T \cdot, \{(), +\}$

$E_9 : T \rightarrow (E \cdot), \{\epsilon, +\}$

$E_{10} : T \rightarrow (E \cdot), \{(), +\}$

$E \rightarrow E \cdot + T, \{(), +\}$

)

$E_{10} \xrightarrow{\epsilon} \phi, E_{13}$

$E_{10} \xrightarrow{+} S_5, E_3$

$E_{11} : E \rightarrow E^+ T \cdot, \{\epsilon, +\}$

$E_{12} : E \rightarrow E^+ T \cdot, \{(), +\}$

$E_{13} : T \rightarrow (E \cdot), \{(), +\}$

Fig. 3 : Automate LR(1) pour la grammaire G'1

On trouve fig. 4 la représentation matricielle de l'automate précédent subdivisé en 3 matrices

Action, Goto, Reduc

Soit t le prochain symbole terminal dans la chaîne d'entrée

si Action(Ei, t)=S : on lit le terminal suivant dans la chaîne d'entrée, et

si Goto(Ei, t) = Sj, Ek alors on empile Sj et l'on va à  
l'étiquette Ek (So représente l'état vide)

Si  $\text{Action}(E_i, t) = p$  : cela signifie que  $t$  est un suivant possible de la  $i$ ème production soit  $A \rightarrow a$ , alors le  $N^p$  est ajouté au fichier de sortie contenant la liste des réductions effectuées, la pile des états est écrétée de  $|a|$ , soit  $S$  le nouveau sommet de pile si  $\text{Reduc}(S, A) = S_j$ ,  $E_k$  alors l'on empile  $S_j$  et l'on va à l'étiquette  $E_k$ .

si  $\text{Action}(E_i, t) = A$  alors  $t = \epsilon$ , fin d'analyse

si  $\text{Action}(E_i, t) = X$  : erreur de syntaxe

Fig. 4 : Représentation matricielle de l'automate LR(1) de G<sup>1</sup>.

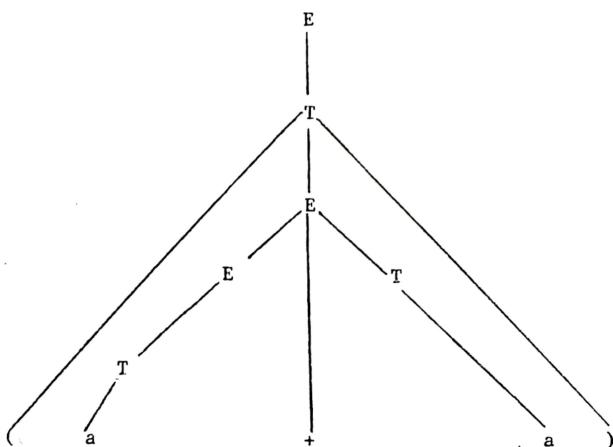
Action	a	+	(	)	$\epsilon$	Goto	a	+	(	)
$E_1$	S	X	S	X	X		$S_0, E_2$	X	$S_2, E_3$	X
$E_2$	X	4	X	X	4		X	X	X	X
$E_3$	S	X	S	X	X		$S_0, E_4$	X	$S_3, E_3$	X
$E_4$	X	4	X	4	X		X	X	X	X
$E_5$	X	5	X	X	A		X	$S_4, E_1$	X	X
$E_6$	X	2	X	X	2		X	X	X	X
$E_7$	X	S	X	S	X		X	$S_5, E_3$	X	$S_0, E_9$
$E_8$	X	2	X	2	X		X	X	X	X
$E_9$	X	3	X	X	3		X	$S_5, E_3$	X	$S_0, E_{13}$
$E_{10}$	X	S	X	S	X		X	X	X	X
$E_{11}$	X	1	X	X	1		X	X	X	X
$E_{12}$	X	1	X	1	X		X	X	X	X
$E_{13}$	X	3	X	3	X		X	X	X	X

Reduc	E	T
$S_1$	$S_0, E_5$	$S_0, E_6$
$S_2$	$S_0, E_7$	$S_0, E_8$
$S_3$	$S_0, E_{10}$	$S_0, E_8$
$S_4$	X	$S_0, E_{11}$
$S_5$	X	$S_0, E_{12}$

Exemple analyse de (a+a)

$(S_1, E_1(a+a), \epsilon) \rightarrow (S_1 S_2, E_3, a+a), \epsilon$   
 $\rightarrow (S_1 S_2 S_0, E_4, +a), \epsilon$   
 $\rightarrow (S_1 S_2 S_0, E_8, +a), 4$   
 $\rightarrow (S_1 S_2 S_0, E_7, +a), 42$   
 $\rightarrow (S_1 S_2 S_0 S_5, E_3, a), 42$   
 $\rightarrow (S_1 S_2 S_0 S_5 S_0, E_4, ), 42$   
 $\rightarrow (S_1 S_2 S_0 S_5 S_0, E_{12}, ), 424$   
 $\rightarrow (S_1 S_2 S_0, E_7, ), 4241$   
 $\rightarrow (S_1 S_2 S_0 S_0, E_9, \epsilon, 4241)$   
 $\rightarrow (S_1 S_0, E_6, \epsilon, 42413)$   
 $\rightarrow (S_1 S_0, E_5, \epsilon, 424132)$   
 $\rightarrow \text{Stop}$

Arbre d'analyse :



Propriété 1 : On montre que les champs erreurs des matrices Goto et Reduc ne sont jamais atteint, cela quelque soit le texte source fournit à l'analyseur.

#### 6.1. - Extension des contextes de réduction

Propriété 2 : On montre que sur toute étiquette  $E_i$  telle que  $Action(E_i, t) = p$  il est possible de remplacer  $Action(E_i, a) = X$  par  $Action(E_i, a) = p$  c'est-à-dire qu'au lieu de détecter une erreur sur  $Action(E_i, a)$  on effectuera la réduction N°p. La propriété du préfixe-correcte n'est pas modifiée par ce changement, l'erreur sera détectée lors de la tentative de lecture de a. L'inconvénient de cette modification intervient seulement au niveau du rattrapage d'erreur ("défaire" les actions sémantiques indument exécutées), mais l'on verra que par une modification minime de l'analyseur il est possible d'échapper complètement à cet inconvénient. Cette modification, connue sous le nom d'extension des contextes de réduction conduit aux matrices de la fig. 5.

#### 6.2. - Fusion des états équivalents

L'on s'aperçoit alors que certaines lignes des matrices Action, Goto, deviennent égales :  $E_2$  et  $E_4$ ,  $E_6$  et  $E_8$ ,  $E_9$  et  $E_{13}$ ,  $E_{11}$  et  $E_{12}$  et que d'autres comme  $E_1$  et  $E_3$  le sont presque :

$E_1$  et  $E_3$  sont égaux si  $E_2$  est égal à  $E_4$  et si  $S_2$  est égal à  $S_3$ . Or  $E_2$  est égal à  $E_4$ . De plus  $S_2$  et  $S_3$  sont égaux si  $E_7$  est égal à  $E_{10}$ . Or  $E_7$  et  $E_{10}$  sont égaux si  $E_9$  est égal à  $E_{13}$  or  $E_9$  et  $E_{13}$  sont égaux

Donc en définitive l'on a

$$E_1=E_3, E_2=E_4, E_6=E_8, E_7=E_{10}, E_9=E_{13}, E_{11}=E_{12} \quad \text{et} \quad S_2=S_3, S_4=S_5$$

Cette optimisation par fusion des états équivalents conduit aux matrices de la figure 6.

Action.	a	+	(	)	$\epsilon$	Goto	a	+	(	)
E <sub>1</sub>	S	X	S	X	X					
E <sub>2</sub>	4	4	4	4	4		S <sub>0</sub> , E <sub>2</sub>	X	S <sub>2</sub> , S <sub>3</sub>	X
E <sub>3</sub>	S	X	S	X	X		X	X	X	X
E <sub>4</sub>	4	4	4	4	4		S <sub>0</sub> , E <sub>4</sub>	X	S <sub>3</sub> , E <sub>3</sub>	X
E <sub>5</sub>	X	S	X	X	A		X	X	X	X
E <sub>6</sub>	2	2	2	2	2		X	X	X	X
E <sub>7</sub>	X	S	X	S	X		X	S <sub>5</sub> , E <sub>3</sub>	X	S <sub>0</sub> , E <sub>9</sub>
E <sub>8</sub>	2	2	2	2	2		X	X	X	X
E <sub>9</sub>	3	3	3	3	3		X	X	X	X
E <sub>10</sub>	X	S	X	S	X		X	S <sub>5</sub> , E <sub>3</sub>	X	S <sub>0</sub> , E <sub>13</sub>
E <sub>11</sub>	1	1	1	1	1		X	X	X	X
E <sub>12</sub>	1	1	1	1	1		X	X	X	X
E <sub>13</sub>	3	3	3	3	3		X	X	X	X

Reduc	E	T
S <sub>1</sub>	S <sub>0</sub> , E <sub>5</sub>	S <sub>0</sub> , E <sub>6</sub>
S <sub>2</sub>	S <sub>0</sub> , E <sub>7</sub>	S <sub>0</sub> , E <sub>8</sub>
S <sub>3</sub>	S <sub>0</sub> , E <sub>10</sub>	S <sub>0</sub> , E <sub>8</sub>
S <sub>4</sub>	X	S <sub>0</sub> , E <sub>11</sub>
S <sub>5</sub>	X	S <sub>0</sub> , E <sub>12</sub>

Fig. 5 : Extension des contextes de réduction

	a	+	(	)	$\epsilon$		a	+	(	)
E <sub>1</sub>	S	X	S	X	X	S <sub>0</sub> , E <sub>2</sub>	X	S <sub>0</sub> , E <sub>1</sub>	X	
E <sub>2</sub>	4	4	4	4	4	X	X	X	X	
E <sub>5</sub>	X	S	X	X	A	X	S <sub>4</sub> , E <sub>1</sub>	X	X	
E <sub>6</sub>	2	2	2	2	2	X	X	X	X	
E <sub>7</sub>	X	S	X	S	X	X	S <sub>4</sub> , E <sub>1</sub>	X	S <sub>0</sub> , E <sub>9</sub>	
E <sub>9</sub>	3	3	3	3	3	X	X	X	X	
E <sub>11</sub>	1	1	1	1	1	X	X	X	X	

	E	T
S <sub>1</sub>	S <sub>0</sub> , E <sub>5</sub>	S <sub>0</sub> , E <sub>6</sub>
S <sub>2</sub>	S <sub>0</sub> , E <sub>7</sub>	S <sub>0</sub> , E <sub>6</sub>
S <sub>4</sub>	X	S <sub>0</sub> , E <sub>11</sub>

Fig. 6 : Fusion des états équivalents

### 6.3. - Suppression des réductions simples

Dans une grammaire, une production simple est une production de la forme  $A+B$  avec  $A,B\in N$ .

Ce type de production, n'a en général aucune sémantique qui lui est attachée, il est donc inutile (et même nuisible du point de vue de l'efficacité de l'analyse) pour l'analyseur d'effectuer de telle réduction. Le but de l'optimisation présentée est d'essayer de supprimer de telle réduction. Le gain obtenu peut être assez considérable, une description des expressions du PL/I nécessite 13 niveaux de productions simples pour décrire la priorité des opérateurs les uns par rapport aux autres, ce qui veut dire que, pour reconnaître qu'un nombre seul est une expression il faut effectuer 13 réductions simples, alors que si elles sont supprimées l'on effectuera directement la réduction de nombre+<expression>.

On montre qu'il est toujours possible de supprimer les réductions simples, mais cette optimisation totale n'est en général, pas compatible avec les optimisations précédentes et de plus peut augmenter la taille de l'automate généré.

Dans OPTIM on ne supprime que les réductions simples auxquelles il n'y a pas de sémantique d'attachée, compatibles avec les optimisations précédentes et diminuant la taille de l'automate généré.

Dans notre exemple la seule production simple est la 2 :  $E \rightarrow T$ . La réduction 2 intervient dans la ligne  $E_6$  or la ligne  $E_6$  n'est atteinte que par  $\text{Reduc}(S_1, T) = S_0, E_6$  et  $\text{Reduc}(S_2, T) = S_0, E_6$  donc après réduction vers le non-terminal  $T$  si l'état  $S_1$  ou  $S_2$  se trouve en sommet de pile. Or l'action effectuée sur la réduction est : dépiler de 1 (production simple) donc supprimer le  $S_0$  qui venait d'être empilé, le nouveau sommet de pile est donc  $S_1$  ou  $S_2$  et l'on va effectuer  $\text{Reduc}(S_1, E) = S_0, E_5$  ou  $\text{Reduc}(S_2, E) = S_0, E_7$  actions que l'on aurait pu effectuer directement sur  $\text{Reduc}(S_1, T)$  ou  $\text{Reduc}(S_2, T)$ .

On supprime donc la ligne  $E_6$  et la matrice Reduc devient :

	E	T
$S_1$	$S_0, E_5$	$S_0, E_5$
$S_2$	$S_0, E_7$	$S_0, E_7$
$S_4$	X	$S_0, E_{11}$

Or d'après la propriété 1 on constate que les deux colonnes E et T sont identiques, il est donc possible de les fusionner en une seule. Ce qui veut dire que du point de vue de l'analyse il est équivalent d'effectuer une réduction vers E ou vers T. C'est une des rares optimisation, qui améliore à la fois la vitesse de l'analyseur et la taille de l'automate d'analyse.

Les matrices obtenues se trouvent figure 7.

	a	+	(	)	$\epsilon$	a	+	(	)
$E_1$	S	X	S	X	X	$S_0, E_2$	X	$S_2, E_1$	X
$E_2$	4	4	4	4	4	X	X	X	X
$E_5$	X	S	X	X	A	X	$S_4, E_1$	X	X
$E_7$	X	S	X	S	X	X	$S_4, E_1$	X	$S_0, E_9$
$E_9$	3	3	3	3	3	X	X	X	X
$E_{11}$	1	1	1	1	1	X	X	X	X

	T
	E
$S_1$	$S_0, E_5$
$S_2$	$S_0, E_7$
$S_4$	$S_0, E_{11}$

fig. 7 : Suppression des réductions simples

#### 6.4. - Génération des productions de Floyd-Evans

Contrairement à ce que montre l'exemple jouet précédent la matrice obtenue est très creuse (la plupart des éléments de Action (E,t) détectent une erreur).

On a choisi dans SYNTAX de représenter ces matrices par des séquences de productions de Floyd-Evans. Les productions de Floyd-Evans sont les instructions d'un langage permettant de programmer des automates à piles déterministes.

Une production de Floyd-Evans comporte 5 champs

test	scan	reduce	stack	goto
------	------	--------	-------	------

et utilise la pile des états et le symbole terminal sous la tête de lecture.

Si le champ test est présent, la production correspondante n'est exécutée que si le symbole en main est celui du champ test sinon on essaie d'exécuter la production suivante.

Si le champ test est absent la production est exécutée inconditionnellement.

Si le champ scan est présent (symbolisé par "\*") on lit le symbole terminal suivant

Si le champ reduce est absent on empile le contenu du champ stack et l'on va exécuter la production se trouvant à l'adresse contenu dans le champ goto.

Si le champ reduce contient p, on effectue la réduction N°p, on dépile du contenu du champ stack (n défilages sont symbolisés par n"|"'), soit S le nouveau sommet de pile.

On recherche alors dans la table de branchement qu'elle est l'adresse de la production de Floyd-Evans associée avec  $S_0$  et le non-terminal contenu dans le champ goto (symbolisé par goto  $t$  suivi du nom du non terminal).

On obtient le programme (en production de Floyd-Evans) de la fig. 8.

La production  $E_0$ , se contente de lire le premier terminal du texte source, d'empiler  $S_1$  et d'aller exécuter la production se trouvant en  $E_1$ .

Les étiquettes des séquences de productions de Floyd-Evans correspondent aux lignes des matrices Action et Goto.

La matrice Reduc a produit

- une table de branchement
- des productions de Floyd-Evans étiquetées "Ti:"

#### 6.5. - Compaction en TEST-SCAN-REDUCE

On voit que toutes les séquences ne comportant qu'une seule production effectuant une réduction sans test ( $E_2, E_9, E_{11}$ ) peuvent être incorporées dans les productions les référant.

Par exemple

$T_3 :$	$S_0$	Goto $E_{11}$
$E_{11} :$	1	Goto $tE$

peut être réduit en :

$T_3 :$	1	Goto $tE$
---------	---	-----------

(on dépile une fois de moins pour tenir compte de  $S_0$  qui n'a pas été empilé)  
On obtient ainsi la fig. 9.

$E_0 :$	*	$S_j$	Goto $E_j$
$E_1 :$	a	*	$S_0$ Goto $E_2$
	(	*	$S_2$ Goto $E_1$
		4	erreur
$E_2 :$			Goto tE
$E_5 :$	+	*	$S_4$ Goto $E_j$
	$\epsilon$		STOP
			erreur
$E_7 :$	+	*	$S_4$ Goto $E_1$
	)	*	$S_0$ Goto $E_9$
			erreur
$E_9 :$	3		Goto tE
$E_{11} :$	1		Goto tE
$T_1 :$		$S_0$	Goto $E_5$
$T_2 :$		$S_0$	Goto $E_7$
$T_3 :$		$S_0$	Goto $E_{11}$

TABLE DE BRANCHEMENT

$S_j : (E, T_j)$

$S_2 : (E, T_2)$

$S_4 : (E, T_3)$

Fig. 8 : FLOYD-EVANS GENERE A PARTIR DES MATRICES

$E_0 :$	*		$S_1$	Goto $E_1$
$E_1 :$	a	*	4	Goto tE
	(	*		$S_2$ Goto $E_1$
				erreur
$E_5 :$	+	*	$S_4$	Goto $E_1$
	$\epsilon$			STOP
				erreur
$E_7 :$	+	*	$S_4$	Goto $E_1$
	)	*	3	Goto tE
				erreur
$T_1 :$			$S_0$	Goto $E_5$
$T_2 :$			$S_0$	Goto $E_7$
$T_3 :$		1		Goto tE

TABLE DE BRANCHEMENT

$S_1 : (E, T_1)$

$S_2 : (E, T_2)$

$S_4 : (E, T_3)$

Fig. 9 : COMPACTION EN : TEST-SCAN-REDUCE

#### 6.6. - Optimisation de la table de branchement

- Si, dans la table de branchement, un non-terminal donné A à la même adresse de branchement a dans le Floyd-Evans, cela quelque soit l'état S considéré, cela signifie que cette adresse a est connue statiquement (lors de la construction) on peut donc remplacer toutes les occurrences de Goto tA par un branchement direct Goto a. Bien entendu toutes les occurrences du couple (A,a) dans la table de branchement sont supprimées.
- si, dans la table de branchement, pour un état S donné toutes les adresses de branchement dans le Floyd-Evans sont les mêmes, soit a, cela quelque soit le non-terminal considéré, il est possible d'empiler l'adresse a à la place de l'état S et de supprimer l'état S de la table de branchement.  
On obtient la figure 10, la table de branchement a complètement disparue.

Après ces optimisations, lors de l'analyse la recherche d'une adresse de branchement après réduction se fait dans l'ordre

- branchement direct
- branchement par le sommet de pile
- processus normal.

#### 6.7. - Réarrangement optimal des séquences

On constate assez souvent que des séquences entières de production de Floyd-Evans sont des sous-ensembles d'autres séquences, comment réarranger ces productions pour en avoir le nombre minimal. Ce problème [ICH70] a une solution particulièrement simple: considérons la matrice

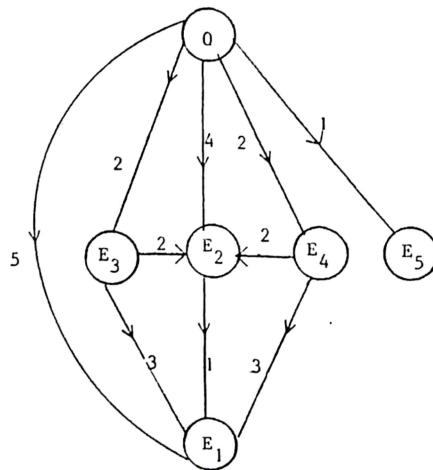
	a <sub>1</sub>	a <sub>2</sub>	a <sub>3</sub>	a <sub>4</sub>	a <sub>5</sub>	a <sub>6</sub>
E <sub>1</sub>	S <sub>1</sub>		S <sub>3</sub>	S <sub>4</sub>	S <sub>5</sub>	S <sub>6</sub>
E <sub>2</sub>	S <sub>1</sub>		S <sub>3</sub>	S <sub>4</sub>		S <sub>6</sub>
E <sub>3</sub>	S <sub>1</sub>			S <sub>4</sub>		
E <sub>4</sub>	S <sub>1</sub>					S <sub>6</sub>
E <sub>5</sub>			S <sub>2</sub>			

$E_0 :$	*		$T_1$	$\text{Goto } E_j$
$E_1 :$	a	*	4	$\text{Goto } tE$
	(	*		$\text{Goto } E_1$
	)	*		erreur
$E_5 :$	t	*	$T_3$	$\text{Goto } E_1$
	c			STOP
				erreur
$E_7 :$	+	*	$T_3$	$\text{Goto } E_1$
	)	*	3	$\text{Goto } tE$
				erreur
$T_1 :$			$S_0$	$\text{Goto } E_5$
$T_2 :$			$S_0$	$\text{Goto } E_7$
$T_3 :$		1		$\text{Goto } tE$

Fig. 10 : OPTIMISATION DE LA TABLE DE BRANCHEMENT

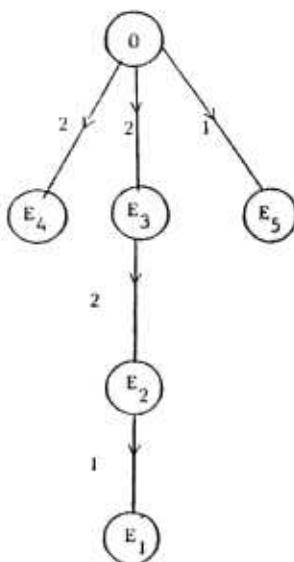
Sous précautions, cette matrice nécessite 14 productions pour être représentée.

Construisons le graphe orienté acyclique avec une racine (notée 0 représentant une ligne vide). Ce graphe a autant de noeuds que la matrice a de ligne et de plus un arc va du noeud  $E_i$  au noeud  $E_j$  ssi  $E_i \subset E_j$  de plus cet arc est étiqueté par  $|E_j - E_i|$



Tout arbre maximal construit sur ce graphe représente une possibilité de génération correcte. Le problème est donc de trouver l'arbre maximal de coût maximal.

Pour chaque noeud (excepté la racine) il suffit de conserver une (l') arête entrante étiquetée par la valeur minimale par exemple :



et l'on génère à partir des feuilles vers la racine :

$$E_4 : S_1$$

$$S_b$$

$$E_1 : S_5$$

$$E_2 : S_3$$

$$S_b$$

$$E_3 : S_1$$

$$S_4$$

$$E_5 : S_2$$

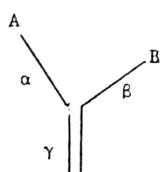
#### 6.8. - Utilisation des sous-séquences communes

Mais, le plus souvent, deux séquences n'ont en commun qu'un sous-ensemble. De plus, dans l'exemple précédent nous avons supposé que l'ordre des instructions *à l'intérieur d'une séquence* était quelconque (on pouvait les réarranger à notre guise) et la dernière instruction d'une séquence de productions de

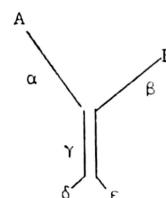
Floyd-Evans est soit erreur, soit une réduction sans champ test, production devant obligatoirement être la dernière exécutée (si toutes les précédentes ont échouées).

Pour deux séquences A et B ayant une sous-séquence commune  $\gamma$  on a donc les deux cas suivants (peut être après réarrangement interne des séquences)

cas 1



cas 2



dans ce cas le Floyd-Evans générée est le suivant :

A : sous-séquence  $\alpha$

AB : sous-séquence commune  $\gamma$

B : sous-séquence  $\beta$   
goto AB

A : sous-séquence  $\alpha$   
call AB

production  $\delta$

B : sous-séquence  $\beta$   
call AB

production  $\epsilon$

AB : sous-séquence commune  $\gamma$   
return

Il semblerait que dans ce cas la solution optimale (nombre optimal de productions générées) soit un problème NP-complet. OPTIM utilise une modification de l'algorithme d'Ichbiah et Morse et quelques heuristiques semblant donner de bons résultats.

$E_0 :$	*			$T_1$	Goto $E_1$
$E_1 :$	a	*	4		Goto $tE$
	(	*		$T_2$	Goto $E_1$
					erreur
$E_5 :$	*				STOP
					Goto $E_5 E_7$
$E_7 :$	)	*	3		Goto $tE$
$E_5 E_7 :$	+	*		$T_3$	Goto $E_1$
					erreur
$T_1 :$				$S_0$	Goto $E_5$
$T_2 :$				$S_0$	Goto $E_7$
$T_3 :$			1		Goto $tE$

Fig. 11 : Réutilisation des sous-séquences communes

Cette technique de réutilisation des sous-séquences communes est également utilisée pour optimiser la table de branchement.

On trouvera ci-dessous un tableau donnant les temps de construction (constructeur SLR(1)) et d'optimisation sur IRIS 80 ainsi que les tailles des automates produits, cela pour un certain nombre de langages. La longueur d'une grammaire est la somme des longueurs des parties droites de ses productions.

LANGAGE	NOMBRE DE REGLES	LONGUEUR	TEMPS DE CONSTR. (mn)	TEMPS DE L'OPTIM. (mn)	NB DE PRODUCTIONS GENERES			TABLE DE BRANCHEMENT		
					Sans Optim	Avec Optim	Gain (%)	Sans Optim	Avec Optim	Gain (%)
SMALL	48	137	0.04	0.07	156	113	27	118	23	80
SNOBOL4	51	165	0.06	0.08	158	119	24	127	32	74
Express	63	199	0.04	0.09	222	137	38	73	12	83
PAL	79	261	0.16	0.21	306	193	36	646	62	90
LET	92	281	0.11	0.10	367	204	44	392	64	83
EULER	120	347	0.28	0.24	509	222	56	1015	72	92
ALGOL60	147	429	0.17	0.30	481	318	33	694	93	86
PL360	151	446	0.11	0.23	515	297	42	315	42	86
LIS	182	563	0.38	0.73	986	464	52	935	142	84
SIMULA67	265	812	0.96	1.02	1093	577	47	1814	180	90
PL/I	354	1162	0.62	1.70	1617	758	53	1831	161	91

## 7. - Traitement des erreurs

Un analyseur (lexical ou syntaxique) donné doit nécessairement, après chaque détection d'erreur, pouvoir continuer son travail donc se "récupérer" de l'erreur détectée. Il est donc naturel, dans le cadre d'un compilateur de compilateur de concevoir un modèle de récupération d'erreur générée automatiquement.

Le traitement des erreurs du système SYNTAX [BOU77] se décompose en deux parties

- une tentative de correction locale du programme source erroné
- si la tentative précédente échoue, récupération au niveau global de l'erreur détectée.

Nous allons illustrer la méthode par un exemple soit G2 la grammaire d'un petit langage à structure de blocs. Chaque bloc est formé de deux parties,

- déclaration d'une liste de variables entières
- liste d'instructions séparées par des points-virgules ; une instruction est soit vide, soit une affectation, soit un autre bloc. Une expression est supposée être un symbole terminal ;

G2 :

1: <P>→<B>	7: <LS>+<LS>; <S>
2: <B>→BEGIN <LD> <LS>END	8: <LS>+<S>
3: <LD>+	9: <S>+
4: <LD>→INTEGER <LV>;	10: <S>+<B>
5: <LV>→ID	11: <S>+ID=EXP
6: <LV>→<LV>, ID	

L'automate d'analyse figure 12 peut être construit pour G2 si l'on suppose que le symbole "z" termine chaque sentence de  $\ell(G2)$ .

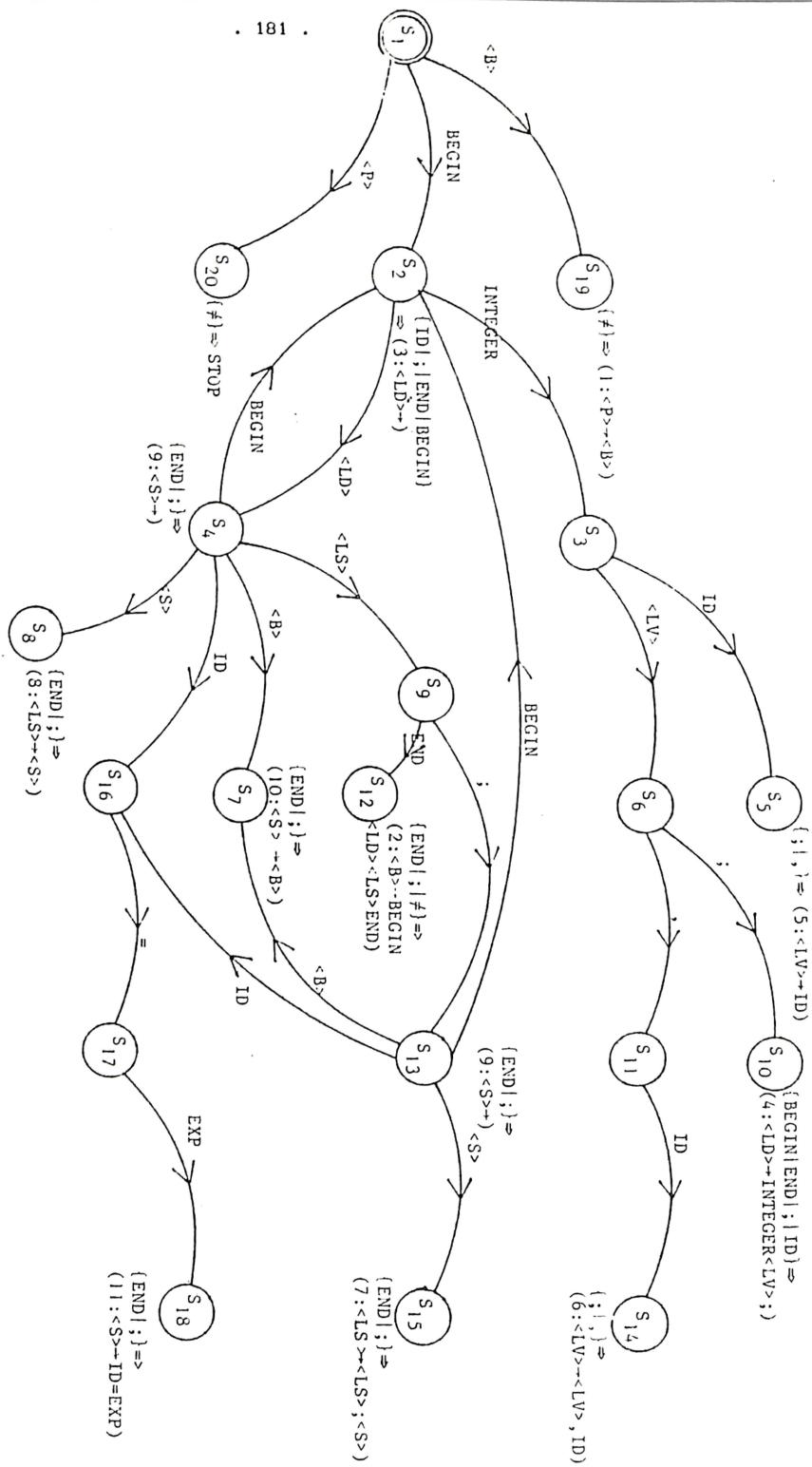


Fig.12 : Automate d'Analyse SLR(1) pour la grammaire G2

Soit le programme

```
BEGIN  
INTEGER I  
I=EXP  
END#
```

dans lequel le point-virgule séparant la partie déclaration de la partie instruction a été oublié.

La séquence de configurations prise par l'automate d'analyse est :

```
(S1, "BEGIN INTEGER I I=EXP END#", ε)  
→ (S1S2, "INTEGER I I=EXP END#", ε)  
→ (S1S2S3, "I I=EXP END#", ε)  
→ (S1S2S3S5, "I=EXP END#", ε)  
→ ERREUR(seuls les symboles ";" et "\", peuvent suivre l'état S5  
et provoquer la réduction n°5)
```

#### 7.1. - Correction locale

Cette approche basée sur les idées de LAFRANCE [LAF71] est valide pour tous les analyseurs ayant la propriété du préfixe correcte.

Supposons qu'une erreur de syntaxe soit détectée dans l'état S et soit  $\alpha\beta$  le texte source où  $\alpha$  est le préfixe valide déjà analysé et  $\beta=a_0 a_1 a_2 a_3 \dots$  est le suffixe non encore analysé,  $a_0$  étant le symbole terminal sur lequel l'erreur a été détectée ( $\text{Action}(S, a_0) = \text{erreur}$ ).

La méthode de correction locale s'énonce comme suit :

- on calcule l'ensemble V de toutes les chaînes terminales de longueur n tel que quelque soit la chaîne x de V la chaîne  $\alpha x$  soit un préfixe valide d'une sentence du langage considéré (pour faire ce calcul l'automate est utilisé en génération et non pas en reconnaissance).
- on choisit dans V une chaîne qui va remplacer le début de  $\beta$  de telle façon que le texte source modifié "ressemble" au texte source initial. Cette ressemblance est décrite par l'utilisateur du système qui spécifie une liste de modèles de priorité décroissante.

Considérons par exemple les trois modèles suivants :

1:	X	0	1	2
2:	X	1	2	3
3:	1	2	3	4



Priorité décroissante

Chaque élément de cette liste est un modèle de chaîne où une occurrence d'un X a la signification d'un symbole terminal quelconque et l'occurrence d'un nombre i signifie  $a_i$ , le ième terminal du suffixe  $\beta$ .

Supposons que la chaîne  $b a_1 a_2 a_3$  soit dans V, cette chaîne satisfait le modèle n°2, qui est interprété de la façon suivante :

le symbole  $a_0$  est erroné, il peut être remplacé par le symbole b. La chaîne d'entrée " $a_0 a_1 a_2 a_3$ " sera donc remplacée par " $b a_1 a_2 a_3$ " à moins qu'il n'y ait dans V une chaîne qui satisfasse le premier modèle (" $c a_0 a_1 a_2$ " par exemple) de plus forte priorité.

Dans l'exemple précédent l'ensemble V de toutes les chaînes de longueur 3 qui sont des suivants valides de

"BEGIN INTEGER I" est :

```
V={,ID,|,ID;|;ID=|;BEGIN INTEGER|;BEGIN ID|;BEGIN BEGIN|;BEGIN;|; BEGIN END|
    ;;ID|;;BEGIN|;;|;;END|;END≠}
```

Si l'on suppose que la liste des modèles est :

1:	X	0	I
2:	X	I	2
3:	I	2	3

et que la chaîne en entrée lors de la détection d'erreur est :

```
I = EXP END
index 0 I 2 3
```

alors la seule chaîne de V qui satisfasse le texte source est ";ID=" en utilisant le modèle n°1.

La correction effectuée par le système est donc d'insérer un point-virgule. Le programme devient donc :

BEGIN INTEGER I ; I EXP END \*

Les modèles peuvent évidemment être plus élaborés :

	<u>modèles</u>				<u>interprétation</u>
1:	X	0	1	2	Oubli d'un symbole
2:	X	1	2	3	Un symbole erroné
3:	1	2	3	4	Un symbole en trop
4:	1	0	2	3	Intervention de deux symboles
5:	X	X	0	1	Oubli de deux symboles
6:	X	X	1	2	Un symbole utilisé à la place de deux autres.
	•				•
	•				•

Le plus grand nombre dans la liste des modèles (ici 4 dans le modèle n°3) donne le nombre maximal de symboles au delà du symbole en erreur que le module de correction doit connaître. La longueur du plus long modèle (ici 5 dans le modèle n°5) est la longueur n des chaînes de V.

La liste des modèles de correction est fournie par l'utilisateur du système, bien que SYNTAX en fournisse une par défaut. On peut penser que plus la longueur n des chaines de V est grande (plus le contexte droit sur lequel s'appuie une correction est grand) meilleure sera la correction, ce n'est pas vrai dans le cas où un grand nombre d'erreurs se produit dans une petite portion de texte car le contexte droit, partiellement erroné, ne peut être utilisé pour la correction. De plus il ne faut pas perdre de vue que le temps de calcul de V est exponentiel en n. Il semble qu'un bon compromis entre la rapidité et la précision de la correction soit atteint pour une valeur de n voisine de 4.

Le même principe de correction est utilisé dans l'analyseur lexical ; là les modèles permettant l'insertion, le remplacement ou la suppression d'un caractère semblent convenir.

Si aucune correction locale, gouvernée par les modèles fournis, ne peut produire une chaîne valide, l'on tente une action plus globale permettant la reprise de l'analyse, cette récupération peut modifier la pile des états.

### 7.2. - Récupération globale

Elle est basée sur des idées émises par James [JAM72] et met en jeu les propriétés de l'analyse LR. Son principe est le suivant : autour du point où l'erreur a été détectée on essaye de déterminer la plus petite portion de texte source, appelée segment en erreur, qui puisse être remplacée par une phrase. C'est à dire que l'on cherche s'il existe une chaîne, dérivée d'un non-terminal, qui remplaçant le segment en erreur redonnerait la consistance syntaxique. Seule l'existence d'une telle phrase (et non une chaîne explicite) est d'importance ici. La découverte d'un tel segment en erreur nécessite la reconnaissance de sa frontière gauche et de sa frontière droite.

#### 7.2.1. - Recherche de la frontière gauche

La frontière gauche du segment en erreur est soit juste devant le symbole terminal en erreur, soit plus probablement dans la partie du texte qui a déjà été analysée et qui, en général, n'est plus accessible. Or dans l'analyse LR la connaissance du texte source déjà analysé est résumé dans la pile des états : chaque occurrence d'un état  $S$  en pile, état depuis lequel il existe une transition valide sur un non terminal donné, peut être considéré comme l'état initial d'un automate reconnaissant une phrase quelconque générée par ce non-terminal. Donc, en termes d'analyse LR, la frontière gauche d'une phrase coïncide avec l'occurrence d'un état  $S$  en pile et le symbole non-terminal vers lequel cette phrase peut être réduite, appartient à l'ensemble des non-terminaux (s'il existe) ayant une transition valide depuis cet état  $S$ . À chaque état correspond un certain nombre de phrases potentielles dont on doit rechercher la frontière droite.

### 7.2.2. - Recherche de la frontière droite

On appelle sous-pile d'une pile P, la pile P' obtenu depuis P par dépilage (longueur P' ≤ longueur P)

Soit S' l'état en sommet de toute sous-pile P' de la pile des états P. Si depuis S' il existe au moins une transition sur un non terminal A on calcule l'ensemble  $W_p^A$ , de toutes les chaînes terminales x de longueur p qui sont des suivants valides de la pile P' après transition sur A.

La recherche de la frontière droite d'une phrase se fait en supprimant dans le texte source les terminaux jusqu'à ce que la chaîne en entrée coïncide sur une longueur de p avec une des chaînes appartenant à l'un des  $W_p^A$ , par exemple.

L'on suppose alors que le segment en erreur est une écriture erronée d'une phrase de A (on ignore laquelle). L'analyse repart avec la pile P' par transition sur A. Le mécanisme de récupération assure qu'aucune erreur ne sera détectée sur une longueur de p symboles.

#### Exemple

Soit le texte erroné

```
BEGIN  
INTEGER I;  
INTEGER J;  
I-EXP...
```

dans lequel la partie déclaration aurait dû être une simple liste d'identifiants (INTEGER I,J;)

La liste des configurations de l'automate est la suivante :

- ( $S_1$ , "BEGIN INTEGER I; INTEGER J; I=EXP...",  $\epsilon$ )
  - ( $S_1 S_2$ , "INTEGER I; INTEGER J; I=EXP...",  $\epsilon$ )
  - ( $S_1 S_2 S_3$ , "I; INTEGER J; I=EXP...",  $\epsilon$ )
  - ( $S_1 S_2 S_3 S_5$ , "; INTEGER J; I=EXP...",  $\epsilon$ )
  - ( $S_1 S_2 S_3 S_6$ , "; INTEGER J; I=EXP...", 5)
  - ( $S_1 S_2 S_3 S_6 S_{10}$ , "; INTEGER J; I=EXP...", 5)
  - ERREUR

Si la liste des modèles est la suivante :

1:	X	0	1
2:	X	1	2
3:	1	2	3

aucune correction n'est possible (le modèle 2 3 4 aurait supprimé INTEGER J) une récupération globale est alors tentée.

Les états  $S_{10}$  et  $S_6$  n'ont pas de transition sur non-terminal, ils ne sont pas significatifs. Cependant l'état  $S_3$  a une transition sur <LV>  
l'état  $S_2$  a une transition sur <LD>  
l'état  $S_1$  a une transition sur <B> et <P>

Prenons par exemple p=2 pour la recherche de la frontière droite. Nous avons

$$W_{S_1 S_2 S_3}^{<LV>} = \{ , ID | ; ID | ; END | ; BEGIN | ; ; \}$$

$$W_{S_1 S_2}^{<LD>} = \{ BEGIN BEGIN | BEGIN END | BEGIN INTEGER | BEGIN; | BEGIN ID | ID= | END= | ; BEGIN | ; | ; ID | ; END \}$$

$$W_{S_1}^{<B>} = W_{S_1}^{<P>} = \{ \# \}$$

L'on voit que ni "INTEGER J" ni "J;" n'appartient à l'un de ces ensembles. Par contre ";" est un élément de  $W_{S_1 S_2 S_3}^{<LV>}$  et un élément de  $W_{S_1 S_2}^{<LD>}$ . La recherche du plus petit segment en erreur nous conduit à prendre ";" dans  $W_{S_1 S_2 S_3}^{<LV>}$ .

L'analyse repart donc dans la configuration :

$(S_1 S_2 S_3 S_6, "; I=EXP...", 5)$

et l'action prise par le mécanisme de récupération globale a été de considérer que "I;INTEGER J" était une phrase générée par <LV> donc une liste de variable dans la partie déclaration.

L'implémentation actuelle tient compte du fait que dans la plupart des langages de programmation il existe des terminaux clés (";", "END", "THEN", "ELSE") qui terminent (séparent) les phrases de haut-niveau, et sur lesquels l'utilisateur peut pouvoir se récupérer.

L'utilisateur de SYNTAX fournit donc avec une liste des modèles de correction des erreurs, une liste de terminaux clés et la longueur  $p$  des chaînes de  $W$ .

La récupération globale procède comme suit :

Lorsque la correction a échouée, le texte source est lu (sans analyse) jusqu'à rencontrer un terminal appartenant à l'ensemble des terminaux-clés. On regarde alors si la chaîne source de longueur  $p$  commençant par ce terminal peut être un suivant possible de l'un des états de la pile.

- si OUI l'analyse repart de cet endroit avec la sous-pile comme pile des états
- si non on recherche dans le texte source le prochain terminal appartenant à l'ensemble des terminaux-clés et l'on recommence.

De bons résultats sont généralement obtenus avec une valeur  $p=2$ .

#### 7.3. - Correction des fautes d'orthographe

Dans le paragraphe sur la correction des erreurs, rien n'a été dit sur le cas où plusieurs chaînes satisfont un modèle donné, en fait SYNTAX en choisie une au hasard (la première construite) or, dans certains cas il serait agréable d'avoir un certain contrôle sur ce choix, c'est en particulier le cas lorsque se produisent des fautes d'orthographe sur les mots-clés.

Considérons par exemple le langage où les mots-clés PROCEDURE et FUNCTION sont les seuls pouvant intervenir à un endroit donné, une correction "aveugle" de PROCEDURE pourrait se faire en FUNCTION.

SYNTAX a un mécanisme de correction des fautes d'orthographe sur les mots-clés [MOR 70], il permet de corriger les fautes simples tombant dans l'une des quatre catégories suivantes :

- omission d'un caractère
- ajout d'un caractère

- remplacement d'un caractère
- interversion de deux caractères

d'après MORGAN ce mécanisme simple permet de corriger 80 % des fautes d'orthographe.

Il faut pourtant noter, qu'en général, ce module ne produit sa pleine puissance que s'il est associé avec un mécanisme de "correction en retard". En effet reprenons l'exemple de la grammaire  $G_2$  une faute dans l'écriture de INTEGER (ENTEGER par exemple) transforme le mot-clé en un identifiant, identifiant qui est syntaxiquement valide à cet endroit là (partie gauche d'affectation) la faute n'étant détectée que sur le symbole suivant. Il paraît donc souhaitable que les modules de correction (fautes d'orthographe et modèles) aient la possibilité de "remonter" dans le texte source. SYNTAX à la possibilité de remonter d'un cran en arrière, donc d'effectuer des corrections impliquant le symbole précédent le symbole sur lequel l'erreur a été détectée.

Les traitements sémantiques effectués en parallèle avec l'analyse syntaxique doivent donc être retardés pour ne pas avoir à les "défaire" lors des corrections "en arrière". Ce problème de "défaire" les actions sémantiques, se pose de toutes les façons dans les analyseurs LR dans lesquels l'optimisation "extension des contextes de réduction" (cf § 6.1) est effectuée. Or cette optimisation est presque toujours faite. Ce mécanisme de retard des actions sémantiques sur la syntaxe doit donc être, de toutes les façons, réalisé.

Ce modèle de correction et de récupération des erreurs a été comparé à son avantage à la fois avec les résultats fournis par des compilateurs industriels mais aussi avec les résultats fournis par d'autres méthodes automatiques [BOU 77].

### 8. Conclusions

Le système SYNTAX utilisé dans un contexte de recherche (IRIA-LABORIA, Université de Toulouse, Université Paris VI) a prouvé sa validité par le fait que de plus en plus d'industriels l'utilisent pour l'écriture de traducteurs:

CII-HB Simula 67

CAP-SOGETI LOGICIEL CPL/I

TITN CERES, HARMONIE

CISI 4X

Toute la partie frontale du premier traducteur prototype d'ADA a été écrite en utilisant la version MULTICS de SYNTAX.

Références

AHO 72

Aho, A.V., Ullman, J.D;  
The theory of parsing, Translation and Compiling;  
Vol 1,2: Parsing, Prentice-Hall, Englewood Cliffs, N.J. (1972)

AHO 77

Aho, A.V., Ullman, J.D;  
Principles of Compiler Design;  
Addison-Wesley, Reading, Mass. (1977)

BOU 77

Boullier, P.J. ;  
Automatic syntactic error recovery for LR-parsers ;  
Proceedings of the 5th annual IRI conference  
Publié par l'IRIA-Guidel (France) (May 1977)

DeR 69

De Remer, F.L.;  
Practical translators for LR(k) languages ;  
Ph.D. thesis, Dept of Electrical Engineering, M.I.T.,  
Cambridge, Mass., (1969)

DeR 74

De Remer, F.L. ;  
Advanced course on compiler construction ;  
Tech. University of Munich Germany.  
March 4 to 15, (1974)

EAR 70

Earley, J. ;  
An efficient context-free parsing algorithm ;  
Comm. ACM 13 p94-102 (1970)

HOP 69

Hopcroft, J.E., Ullman, J.D;  
Formal languages and their relation to automata ;  
Reading (Mass.):Addison-Wesley (1969)

ICH 70

Tchibiah, J.D., and Morse, S.P.;

A technique for generating almost optimal Floyd-Evans productions  
for precedence grammars;  
Comm. ACM 13:8 p.501-508, (1970)

JAM 72

James, L.R.;

A syntax directed error recovery method;  
Master's thesis, Technical report CSR6-13, Computer  
Systems research group, University of Toronto, Canada, (May 1972)

KNU 65

Knuth, D.E.;

On the translation of languages from left to right;  
Information and Control 8 p.607-639 (1965)

LAF 71

Lafrance, J.E.;

Syntax directed error recovery for compilers;  
Ph.D. Thesis, Illiac IV document 249, Computer Science Department,  
University of Illinois, Urbana, Illinois, (June 1971)

MOR 70

Morgan, H.L.;

Spelling correction in systems programs;  
Comm. ACM 13:2 p.90-94 (1970)

SPR 77

Sprignoli, Renzo;

Perfect hashing functions: a single probe retrieving method for static sets;  
Comm. ACM 20 p.841-850 (Nov 1977)