Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Essai prospectif d'amélioration d'un macro générateur à noms distribués

Ville, A.

Award date:
1975

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 04. May. 2026

300F

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX - NAMUR
INSTITUT D'INFORMATIQUE

1974 - 1975

ESSAI PROSPECTIF D'AMELIORATION D'UN MACROGENERATEUR A NOMS DISTRIBUES

A. VILLE

Mémoire présenté en vue de l'obtention
du grade de Licencié et Maître en Informatique

Nous exprimons à Monsieur C. Cherton notre gratitude pour l'attention et l'aide qu'il nous a apportées tout au long de ce travail.

Nous exprimons également notre reconnaissance au Dr Whitby-Strevens pour l'intérêt qu'il a porté à ce mémoire.

Nous remercions les membres du Département d'Informatique de l'Université de Warwick pour l'accueil et la disponibilité dont ils ont fait preuve pendant notre stage.

Nous tenons à remercier vivement tous les membres de l'Institut d'Informatique des Facultés Notre-Dame de la Paix de Namur dont le dévouement, à travers les cours de trois années, a permis la réalisation de ce mémoire.

Namur, le 18 juillet 1975

Sommaire

1. Introduction

2. Proposition d'amélioration d'un macrogénérateur et sa réalisation

3. Conclusions et projets d'extensions.

Avant-propos

Les connaissances acquises sur la macrogénération, suite au précédent mémoire, ont mis en évidence la lenteur des macrogénérateurs.

Notre objectif de base est d'étudier les causes de cette lenteur afin de voir dans quelle mesure elle est inhérente à leur principe même ou si, au contraire, il ne serait pas possible d'y remédier.

Après une brève introduction à ce problème général, trop vaste pour un simple travail de fin d'études, nous aborderons un de ses aspects particuliers : la reconnaissance de macroréférences par noms distribués (Template Matching).

Le corps de notre mémoire consistera en une proposition d'extension à un macrogénérateur existant (Stage 2) permettant d'utiliser un algorithme de Template Matching plus efficient, tout en conservant sa souplesse d'utilisation, voire même en l'augmentant.

Dans le texte, nous avons adopté les conventions suivantes : les appellations "Macrogénérateur à buts multiples" et "Template Matching" seront remplacées par les abréviations "GPMG" (General Purpose Macro Generator) et "TM".

1. Introduction

1.1 Principe de la macrogénération

1.2 Evolution de la conception de la macrogénération

1.2.1 Macroassembleurs

1.2.2 Macrogénérateurs à buts multiples

1.2.3 Macrogénérateurs à buts spécifiques et extensions aux macrogénérateurs existants

1.3 Avantages et inconvénients des macrogénérateurs

1.1 Principe de la macrogénération

Un macrogénérateur prend en considération un texte source et produit un texte objet. Cette transformation s'effectue sur un principe de REMPLACEMENT.

Rappelons que le macromécanisme dirigeant la génération consiste à reconnaître une macroréférence dans le texte source analysé, à identifier la macrodéfinition invoquée (la procédure de transformation), à passer les arguments et à évaluer le texte de remplacement.

Il n'y a donc génération que s'il y a macroréférence et le texte généré remplace généralement cette macroréférence dans le texte objet.

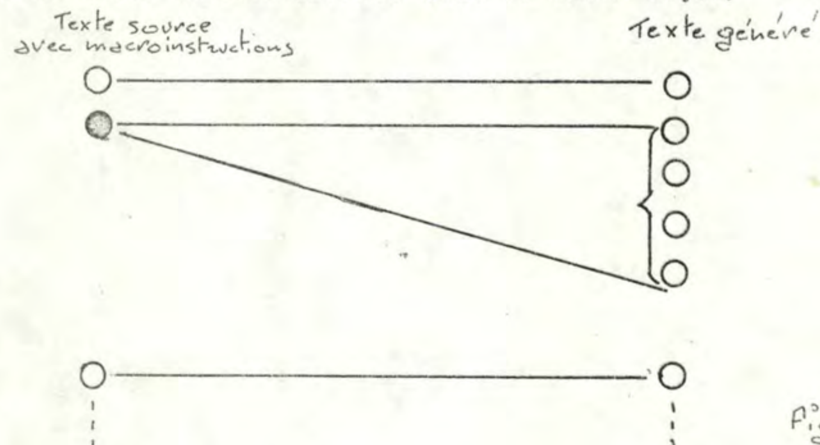


Fig 1.1

1.2 Evolution de la conception de la macrogénération

1.2.1 Macroassembleurs

Ils furent les premiers à utiliser un principe de remplacement.

Le macromécanisme consistait en une simple règle de recopie textuelle après passage d'arguments. Cette règle permettait une manipulation plus aisée des fonctions systèmes (I/O ...), mais la macrodéfinition restait l'apanage du système. Il s'agissait d'une extension prédéfinie du répertoire d'instructions du langage d'assemblage.

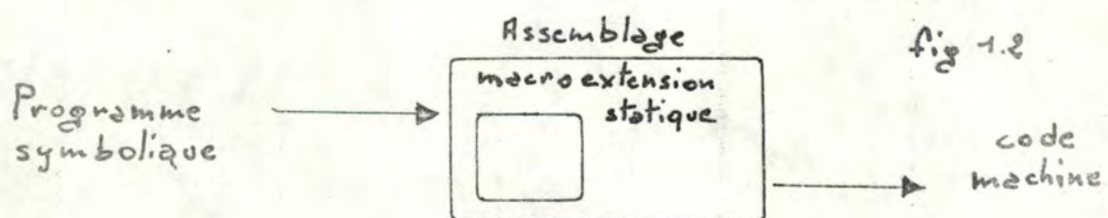


fig 1.2

Très vite, on prit en considération le "macro-time", l'utilisateur pu définir ses propres macros - la règle de recopie étant toujours de rigueur.

1.2.2 Macrogénérateurs à buts multiples

Plus tard, le principe de remplacement fut isolé de l'assemblage, ce qui donna les premiers macrogénérateurs à buts multiples (GPMG).

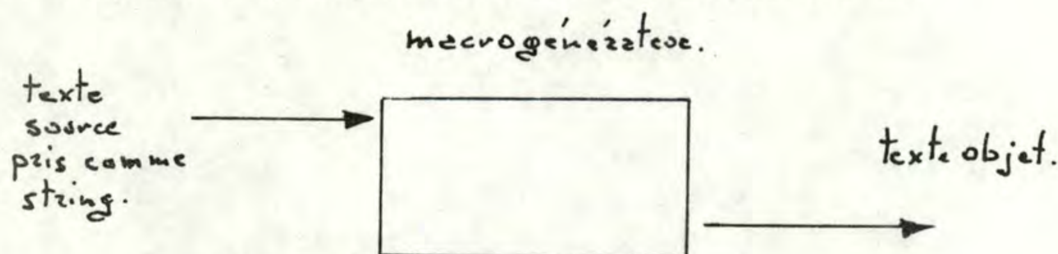


fig 1.3 GPMG.

1.2.3 Macrogénérateurs à buts spécifiques et extensions aux macrogénérateurs existants

Parallèlement au développement des GPMG apparaissait la notion de compilateur et de langages de haut niveau, riches en nouveaux principes.

Il en découla que le "macro-time" bénéficia généralement des notions de variables, de tests et de branchements, de calculs et d'itérations des langages de haut niveau.

Quant aux langages de haut niveau, on leur adjoignit le principe de remplacement créant ainsi la gamme des macrogénérateurs à buts spécifiques (Special Purpose MacroGenerator)

1.3 Avantages et inconvénients des macrogénérateurs

Sur le plan vitesse de traitement, on constate que les macrogénérateurs sont considérablement plus lourds que les compilateurs, à un point tel que ce défaut peut paraître rédhibitoire pour la majorité des applications.

On peut cependant se poser la question de savoir si ce défaut est inhérent aux principes des macrogénérateurs. Sans pouvoir encore répondre à cette question, nous pensons qu'il est possible d'améliorer considérablement les performances sans nuire aux avantages offerts à l'utilisateur.

Une première analyse nous a conduits aux remarques suivantes :

a) la macrogénération est simple et élégante dans son principe. Celui-ci présente une certaine généralité : supposant peu de choses, il possède néanmoins un domaine théorique étendu d'application.
Cependant, dans certains macrogénérateurs, la volonté de n'utiliser que le principe de remplacement à l'exclusion de tout autre, conduit à la fois à un manque de souplesse et d'efficacité.

Nous rappelons à cet effet le long détour nécessaire pour générer un simple IF en GPM à l'aide de ce principe.

IF X ≡ Y then V ELSE W

(X et Y étant des strings quelconques)

Pour effectuer une telle instruction " at macro-time", la macroréférence et les macrodéfinitions suivantes doivent être présentées :

#X, #DEF, X, <V> ; #DEF, Y, <W>; ;

Rappelons qu'en GPM, le préfixe " # " indique le début d'une macroréférence; suivent le nom identifiant la macrodéfinition appelée et, enfin, l'ensemble des paramètres séparés par une virgule.

Les macrodéfinitions présentent le format suivant :

#DEF, nom - de - macro, <corps - de - macro>;

Le délimiteur ";" indiquant la fin d'une macroréférence ou d'une macrodéfinition.

Rappelons que les paramètres sont évalués avant génération, ce qui va permettre l'exécution du "I F" :

1. Le préfixe " #X" est reconnu et constitue un appel de macrodéfinition.

2. L'évaluation du premier paramètre implique l'acceptation d'une nouvelle définition de nom "X" et de corps "V".

3. Le second paramètre indique une macrodéfinition de nom "Y" et de corps "W".

4. Une fois l'indicateur de fin de macroréférence rencontré, la génération commence. On parcourt la table des noms en débutant par les plus récentes macrodéfinitions.

Si X est lexicographiquement identique à Y, W est donc généré; sinon , ne pouvant plus être identique qu'à lui-même, V est considéré.

5. Les macrodéfinitions passées comme paramètre sont alors supprimées des tables et le macrogénérateur poursuit l'analyse du source à la ligne suivante.

Pour "exécuter" un simple "IF" par le principe de remplacement, il fallut faire appel à la définition dynamique de macros qui est toujours d'un emploi très peu naturel.

b) Comme les macrogénérateurs sont interprétatifs en ce sens que, lors d'une génération, non seulement le texte source est analysé pour y détecter les appels de macros, mais le corps des macrodéfinitions est lui aussi interprété à chaque appel, on conçoit aisément à quelles pertes de performances ce mécanisme peut conduire, surtout si la macrogénération peut être effectuée récursivement - ou même itérativement.

C'est dire aussi que l'adjonction de certains principes, propres aux langages de haut niveau, à celui de la macrogénération fut pour une part dans la perte en efficacité des macrogénérateurs.

c) Dans le but d'augmenter la souplesse d'utilisation, certains systèmes acceptent que les macroréférences de l'utilisateur puissent prendre des formes très diverses. Cela conduit à l'emploi d'algorithmes de reconnaissance très généraux donc très lents.

Dans la plupart des langages de haut niveau, au contraire, la rigidité de la syntaxe permet l'emploi d'algorithmes et de techniques d'implémentation très spécifiques (tables de précédence, par exemple), donc plus rapides.

Puisque ce sont, néanmoins, les macrogénérateurs offrant la plus grande liberté quant aux formes syntaxiques des appels de macros qui présentent le plus d'avantages à l'utilisateur au niveau de la souplesse (par exemple, ML/1 et Stage 2), nous tâcherons de tenir compte de cette remarque.

d) Enfin, l'objectif de construire des GPMG a le plus souvent conduit les implémenteurs à considérer le texte source comme une simple suite de caractères. Cette façon de faire prive le macrogénérateur d'informations qui pourraient lui être précieuses pour son efficacité.

Le point (a), cité ci-dessus, est d'ores et déjà assez nettement dépassé par beaucoup de macrogénérateurs qui offrent à l'utilisateur la possibilité d'utiliser d'autres mécanismes que la simple expansion proprement dite.

Quant aux autres points que nous avons soulevés - et bien d'autres sans doute - , il nous semble qu'une étude approfondie reste à faire. Dans le cadre d'un mémoire, nous avons bien été obligés de nous limiter.

Partant d'un subset du GPMG, conçu par Monsieur B. Pirotte (1) nous avons abordé un aspect particulier des points (c) et (d). Ces deux points sont effectivement liés, dans un tel macrogénérateur, dans la mesure où ne fournir au macrogénérateur, aucune indication concernant les intentions du programmeur (en considérant le texte source comme une simple suite de caractères) est d'autant plus nuisible aux performances que les structures syntaxiques possibles sont plus complexes.

Notre travail se limitera donc à une expérience : partant d'un GPMG existant et offrant une grande souplesse dans la syntaxe des appels de macrodéfinition (noms distribués ou Templates), tenter d'y adjoindre un outil permettant à l'utilisateur de fournir au macrogénérateur des indications sur ses intentions, de façon à lui faciliter l'analyse. Et ce, tout en ne diminuant pas la souplesse d'emploi, voire même en cherchant à l'améliorer.

---

(1) Nous avons pris cette implémentation comme base de travail puisque, d'une part, nous désirions implémenter, au départ, le macrogénérateur étendu sur le modular 1 de l'Université de Warwick (projet qui dut être abandonné par manque de place) et dès lors, nous étions tenus de l'écrire en BCPL; et que d'autre part, ne nous intéressant pas aux problèmes fondamentaux d'une implémentation, nous avons évité de travailler sur l'implémentation de Stage 2 conçue par Poole et Waite, car elle fonctionnait par Boot Strap.

Nous reporterons fréquemment le lecteur à la définition de métalangage Stage 2 qu'il trouvera dans :

"The Stage 2 macroprocessor user reference manual" de MMr P.C. Poole et W.M. Waite.

2. Proposition d'amélioration d'un macrogénérateur et sa réalisation

2.1 Inefficience de l'algorithme de T.M.

2.2 Macrogénérateur étendu

2.3 Manuel de référence du macrogénérateur Stage 2 étendu

2.4 Implémentation

2.1 Inefficience de l'algorithme de T.M.

2.1.1. Macroréférences pures (sans paramètre)

2.1.2. Passage de paramètres

2.1.3. Conclusion

2.1 Inefficience de l'algorithme de T.M.

Nous allons présenter en un court exemple le fonctionnement de l'algorithme de T.M.

Lors de l'analyse du texte source, le macrogénérateur possède un répertoire de templates.

Supposons le répertoire et le texte source suivant

Source

Ligne 19 - ...
20 - ...
21 - STOP.
22 - STORE.
23 - PRINT ABCDE IN HEXA.
24 - PRINT INABCDEFGH.
25 - ...

Répertoire

Template 9 - ...
10 - ...
11 - PRINT' IN DEC.
12 - PRINT' IN HEXA.
13 - STOP.
14 - ...
15 - ...

"'" indique la présence d'1 paramètre formel
et "." indique la fin d'une ligne source

Le but de l'algorithme est de détecter les macroréférences, en extraire les paramètres actuels et communiquer ceux-ci aux routines d'expansion. Aussi, chaque ligne est considérée comme une référence possible et comparée lexicographiquement aux templates du répertoire.

2.1.1 Macroréférences pures (sans paramètres)

La ligne 21 concordera avec le cliché 13. Par contre, pour la ligne 22, l'ensemble du répertoire est parcouru en pure perte.

Dans un cas, comme dans l'autre, un certain nombre de templates, selon la représentation interne du répertoire, sont proposés inutilement.(1) Ces tests se terminent souvent par la discordance du premier caractère. Mais un préfixe peut être commun à la ligne source et au template (préfixe "STO", dans "STORE" et "STOP"), ce qui contraint l'algorithme à poursuivre.

---

(1) En Stage 2, les templates sont triés selon leurs préfixes communs.

2.1.2 Passage de paramètres

Si la macroréférence contient des paramètres, le macrogénérateur doit créer les relations :

Paramètre formel $_i$ ⟶ Paramètre actuel $_i$

Testée à l'aide du template 11, la ligne 23 présente le préfixe désiré "PRINT". L'algorithme est informé de l'existence d'un paramètre par la présence de l'indicateur " ' " dans le template.

Le paramètre actuel est supposé de longueur "O" et le suffixe "IN DEC" est proposé avec certains aléas qui retardent l'abandon du template 11 : un préfixe "IN", dans le suffixe "IN DEC", peut être reconnu dans le paramètre actuel "INABCDEFGH" mais non pas l'entièreté du suffixe désiré.

La longueur supposée du paramètre est incrémentée à chaque test négatif. Dans notre cas, l'algorithme sera amené à poser une longueur de 9 et reconnaîtra à cet instant l'indicateur de fin de ligne. Comme le caractère sous-test dans le template ne lui correspond pas, n'ayant plus de caractères disponibles, l'algorithme reprend la ligne avec l'élément suivant du répertoire et obtient finalement un succès total.

En ce qui concerne la dernière ligne source (24) tous les éléments du répertoire sont testés, avec, si nécessaire, les hypothèses de longueur - en pure perte.

Remarquons que si le template contenait plusieurs : " ' " (1), l'algorithme travaillerait selon les règles suivantes :

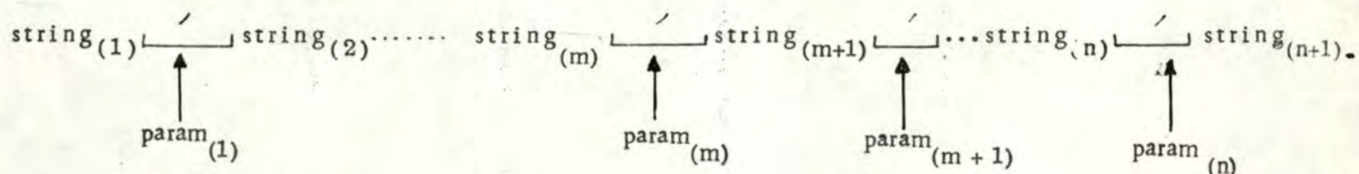
Règle 1 : Le premier paramètre actuel est supposé de longueur "O" et la règle 2 lui est appliquée.

Règle 2 : - Si l'incrémentation de la longueur supposée du paramètre $_{(m)}$ $(1 \leq m \leq n)$ permet la reconnaissance dans la ligne source du string $_{(m+1)}$ le séparant du paramètre $_{(m+1)}$, ce dernier reçoit une longueur "O" et l'on reprend la règle 2.

- Sinon, l'indicateur " . " est atteint dans la ligne source et le template en cours est abandonné.

---

(1) Nous adoptons pour les templates la forme conventionnelle :



Donc, pour chaque paramètre, le même travail laborieux reprend au détriment de la rapidité du macrogénérateur. (Remarque : certaines ambiguïtés peuvent même se présenter. Par exemple, il est généralement impossible à l'utilisateur de passer, pour un paramètre $\text{actuel}_{(m)}$, le $\text{string}_{(m+1)}$ qui le sépare du paramètre $\text{actuel}_{(m+1)}$ suivant :

Soit la macroréférence : ABBEFC.

et le template : A' B' C.

Le premier paramètre formel sera supposé vide et le second paramètre formel reprendra les deux paramètres actuels concaténés.
Pour l'algorithme la ligne se découpe comme ceci :

A | | B | B E F | C.

et pour l'utilisateur :

A | B | B | E F | C.)

2.1.3 Conclusions

Lors d'une macroréférence l'algorithme ne peut sélectionner directement le bon template, mais surtout, il ne dispose pas d'une information suffisante en ce qui concerne la délimitation des paramètres actuels.

Sous un certain point de vue, un template n'est rien d'autre que la définition d'un contexte permettant le passage des paramètres actuels et dans lequel l'indicateur de paramètres formels est une information pauvre que l'on pourrait associer à un prédicat d'existence; ou en terme de BNF, à un "non-terminal" auquel nulle règle de production n'accorderait une structure autre que celle de strings de longueur indéterminée et de composants quelconques.

Une série de tests pourtant simples, se multiplient annihilant l'efficacité de l'algorithme.

2.2 Macrogénérateur étendu

Nous venons de montrer l'inefficacité inhérente à l'algorithme de TM. Finalement, cette inefficacité est conséquente aux deux principes sous-jacents à l'algorithme.

1. Le texte source est perçu comme un ensemble de strings.
2. Les macroréférences se présentent sous l'une des formes définies par les noms distribués.

Dans un compilateur, les tests sont plus complexes mais moins nombreux : il existe un ensemble de règles permettant de sélectionner très tôt, dans l'analyse d'une instruction, un nombre restreint de possibilités.

La différence réside donc au niveau de l'environnement dont celui-ci dispose: table de précédence d'opérateurs, description grammaticale, déclaration,...

2.2.1 Passage des paramètres

Les paramètres actuels jouent souvent un rôle particulier lors de l'analyse d'un problème : ils ne sont généralement pas perçus par l'utilisateur comme de simples strings, mais comme des objets appartenant à certaines classes, comme des êtres d'un certain type.

Dès lors, ne peut-on pas permettre à l'utilisateur de décrire ses classes de façon ~~qu'il~~ à ce que le macro générateur en tire profit pour limiter les tests inutiles ?

Cette idée nous a conduits à imaginer des classes de paramètres actuels.

Nous proposons de remplacer, dans son template, un identificateur de paramètre formel par une référence à une classe. A cette classe serait associé l'ensemble des paramètres actuels pouvant apparaître pour ce paramètre formel précis dans ce template.

Nous devons donc fournir à l'utilisateur la possibilité.

1. de définir le contenu des classes de paramètres actuels
2. d'indiquer dans la définition d'un template les classes auxquelles doit appartenir chacun de ces paramètres.
3. de déclarer des classes de macros.
4. de référencier la classe de macrodéfinitions à considérer pour l'analyse d'une ligne.

fin §2.2.2

2.2.2. Correspondance Macroréférence - Template

Une macrodéfinition est généralement invoquée dans un certain contexte: en d'autres termes, en fonction de la logique sous-jacente à un texte source ou généré, l'utilisateur sait que seules certaines macrodéfinitions sont susceptibles d'être appelées.

Ce qui nous conduit à considérer que les macrodéfinitions pourraient être rangées en classes.

Nous obtiendrions ainsi un répertoire partitionné :

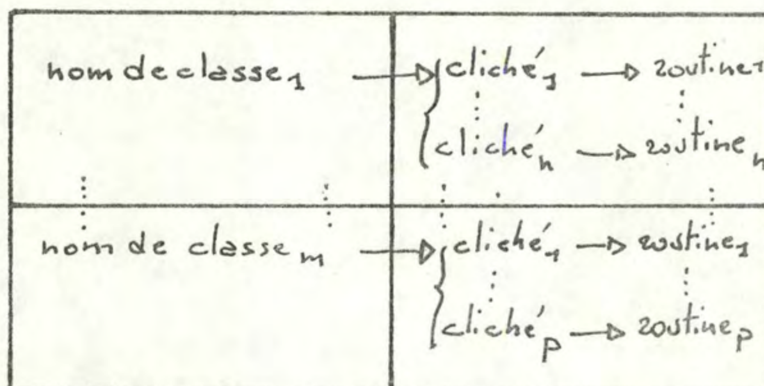


Fig 2.1 Répertoire partitionné

Au niveau des lignes du texte, nous pourrions indiquer, non seulement si nous désirons, ou non, la considération de la ligne en tant que macroréférence possible, mais encore pourrions-nous lui associer une classe de templates susceptible de correspondre à la ligne source.

L'analyse porterait uniquement sur le sous-ensemble indiqué.

2.2.3. Ces moyens sont fournis à l'aide de ce que nous appelons le métalangage.

Nous allons présenter au cours des chapitres suivants :

1) La syntaxe du métalangage utilisé dans notre implémentation et la justification de la forme adoptée.

2) Quelles extensions de ce métalangage seraient susceptibles d'être ajoutées afin d'augmenter l'efficacité du TM et la souplesse d'utilisation.

3) Présenter l'algorithme de TM répondant aux nouvelles possibilités d'analyse.

La figure 2.2 résume le flot d'information dans un macrogénérateur étendu.

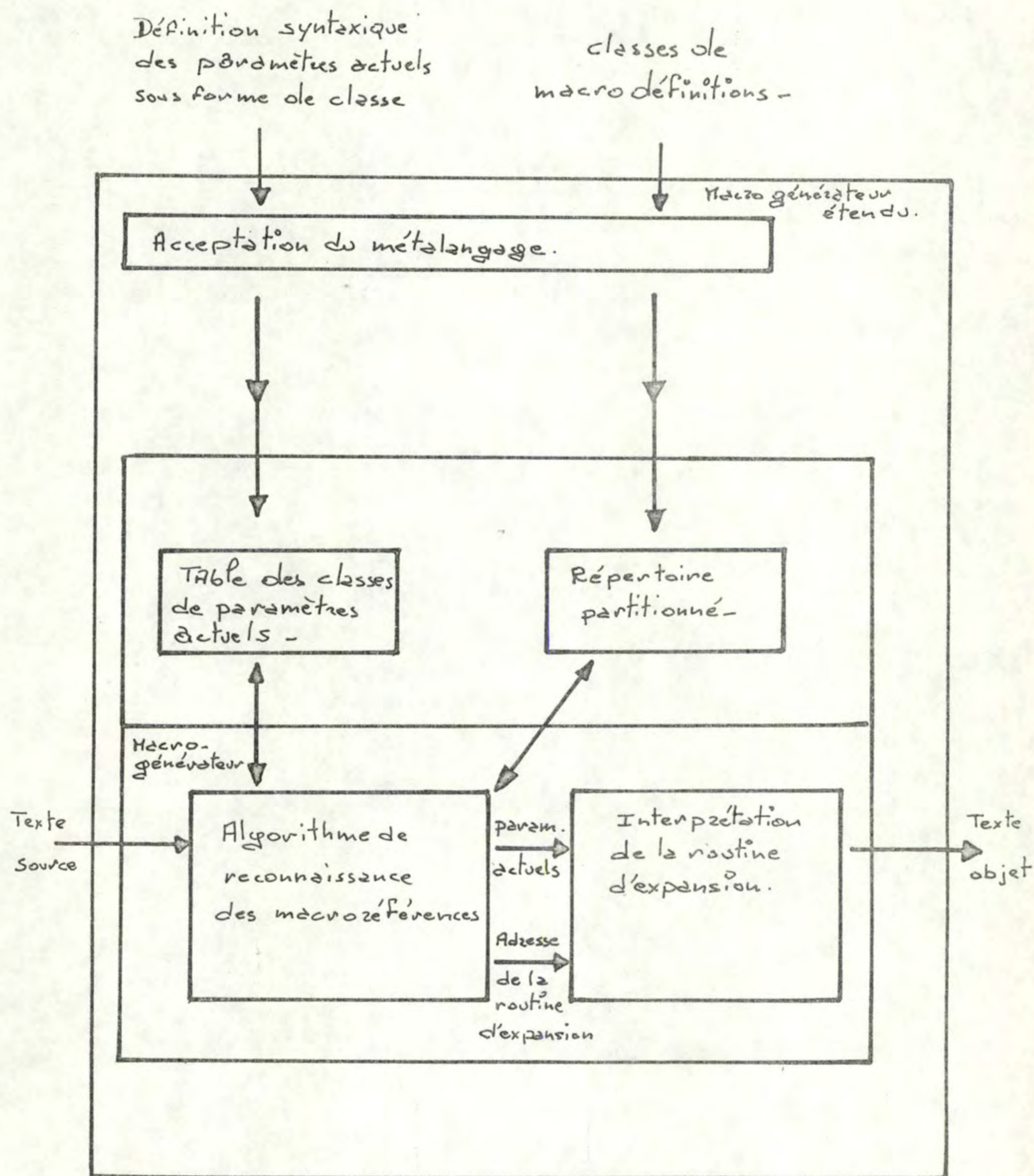


fig 2.2 flot d'information

2.3 Manuel de référence du macrogénérateur Stage 2 étendu.

2.3.1. Choix du macrogénérateur

2.3.2. Description de la syntaxe du métalangage

A) Ligne des caractères de contrôle

B) Métadéclaration des classes

1) Classes de paramètre

a) Production

b) Parenthérisation du BNF

2) Classes de macrodéfinitions

a) Déclaration explicite de nom de classes

b) Déclaration explicite imbriquée

c) Déclaration implicite de nom de classes

d) Syntaxe d'un Template

e) Définition du corps d'une macrodéfinition

C) Métainstructions

1) Directives au TM

2) Directives à l'algorithme de génération

D) Exemple

E) Classes de macrodéfinitions de base

2.3.3. Souplesse d'utilisation

2.3.4. Orientation syntaxique du métalangage et sa justification.

2.3 Manuel de référence du macrogénérateur Stage 2 étendu

2.3.1 Choix du macrogénérateur

Notre métalangage n'a été créé que pour des macroréférences identifiables par noms distribués. Dès lors, deux macrogénérateurs pourraient convenir : ML/1 et Stage 2.

- L'en-tête d'une macrodéfinition en ML/1 est compatible avec ce que nous appelons template, puisqu'elle spécifie un préfixe et des délimiteurs qui doivent être reconnus dans la ligne source afin de permettre la détermination des paramètres actuels.

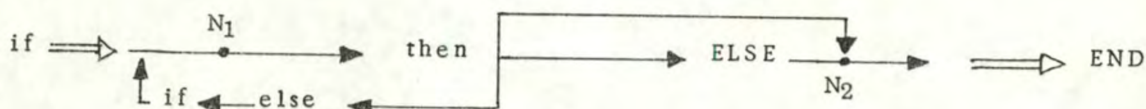
- En outre, le préfixe correspond à ce que nous appelions un "nom de classe" de macrodéfinition.

Si nous présentons la macrodéfinition :

$$\underline{\text{Macdef}}\ \text{if}\ \underline{N_1}\ \text{then} \left\{ \begin{array}{l} \underline{\text{Opt}}\ \text{else}\ \underline{\text{with}}\ \text{if}\ \underline{N_1} \\ \underline{\text{or}} \left\{ \begin{array}{l} \underline{\text{Opt}}\ \text{else}\ \underline{N_2} \\ \underline{\text{or}}\ \underline{N_2}\ \underline{\text{END}} \end{array} \right\} \underline{\text{ALL}} \end{array} \right\} \underline{\text{ALL}}$$

$$\text{As} \left\{ \text{routine d'expansion} \right\}$$

elle précise le préfixe "if" et le graphe de délimiteur :



Dans cette macrodéfinition, "if" est un nom de classe car, une fois reconnu, le graphe associé permet plusieurs formes syntaxiques pour la partie restante de la ligne source.
Ainsi :

if ... Then ...,
if ... Then ... else ...,
if ... Then ... else if ... Then,

sont acceptés comme macroréférence à la macrodéfinition donnée.

Conséquemment, ML/1 pourrait être envisagé comme macrogénérateur de base aux extensions que nous proposons.

Néanmoins, la détermination du graphe n'est pas toujours chose aisée et le préfixe, étant une partie intégrante d'un template, constitue une limitation à la généralité des formes que peuvent prendre les macroréférences et aux possibilités d'utilisation des classes de macrodéfinition.

Stage 2 ne souffre pas de telles limitations. Parmi les macrogénérateurs exposés lors du précédent mémoire, il présente, dès que l'on s'est habitué à la syntaxe des conversions de paramètres et des fonctions systèmes, la plus grande souplesse d'utilisation et la meilleure lisibilité.

2.3.2 Description de la syntaxe du métalangage

A) Ligne des caractères de contrôle

La première ligne proposée aux macrogénérateurs étendus définit les caractères de contrôle.
Elle assure son indépendance par rapport à la machine et au texte source. Ces caractères ne font pas partie intégrante du métalangage, mais sont nécessaires à la reconnaissance de ses éléments par le système.

Il y a deux formats possibles :
- le premier permet l'utilisation du Stage 2 classique.
- le second, celle du macrogénérateur étendu

Voici ces formats, reprenant le choix des caractères utilisés dans ce mémoire :

Format 1

$$\not{\$}_S \ ' \ \not{\$}_T \ e \ 0 \sqcup (+ - * /) * N$$

(Cette ligne peut être précédée par un nombre quelconque de blancs. Le string "* N" représentera la fin physique d'une ligne (carriage return, end of blok))

Format 2

$$\not{\$}_S \ ' \ \not{\$}_T \ e \ 0 \sqcup (+ - * /) , < > [\ ] \ \text{comment} \ * \ N$$

(Le string "comment" représente une zone pouvant contenir un string quelconque, habituellement des commentaires)

Une ligne de contrôle contient donc, à partir d'une position quelconque, dix sept caractères, choisis par l'utilisateur, dont la signification est donnée par le tableau 1 :

Ordre	Fonction	Représentation utilisée
1-	indicateur de fin de ligne source	$\not{\$}_S$
2-	indicateur de paramètre formel	'
3-	indicateur de fin de ligne construite	$\not{\$}_T$
4-	caractère dit "escape"	e
5-	chiffre O (les autres sont pris dans l'ordre croissant du code interne)	O
6-	caractère de remplissage ("padding")	␣ ou ƀ (blanc = caractère)
7-	parenthèse gauche (Fonction système eF_1, expressions et paramètres)	(
8-	opérateur d'addition (Fonction système eF_6 et expression)	+
9-	opérateur de soustraction (Fonction système eF_6 et expression)	-
10-	opérateur de multiplication (expression)	*
11-	opérateur de division (expression)	/
12-	parenthèse droite	)
13-	séparateur "or" d'alternatives	'
14-	séparateur gauche de nom de classe de paramètres actuels	<
15-	séparateur droit de nom de classe de paramètres actuels	>
16-	séparateur gauche des noms de classes de macrodéfinitions	⊏
17-	séparateur droit des noms de classes de macrodéfinitions	⊐

Tableau 1

Les caractères " $\not{b}_S$ " et " $\not{b}_T$ " permettent de définir la fin logique des lignes (par opposition à leur fin physique) du texte source et du texte généré.

Le caractère " ' " sert à indiquer la position des ~~identifications~~ paramètres formels dans les templates en Stage 2 classique.

Nous avons déjà présenté ces trois types d'indicateurs ; par contre, les cinq derniers séparateurs servent principalement pour la syntaxe du métalangage.

Nous avions introduit deux formats de ligne de contrôle afin de pouvoir utiliser le système comme un macrogénérateur Stage 2 classique, pour être à même de procéder à une étude de performance.

Cette dualité se retrouvera encore dans ce manuel, mais, dans la suite, nous ne discuterons que le cas du format 2.

B) Métadéclaration des classes

1 Classes de paramètres

Le langage utilisé pour décrire les classes de paramètres actuels est similaire à la forme de Backus - Naur (BNF).

Les non terminaux et les terminaux de la BNF sont utilisés et redéfinis comme noms de classe et paramètres actuels, respectivement. Chaque production est utilisée pour associer un nom de classe à son contenu.

Le BNF, utilisable avec l'implémentation telle qu'elle est à l'heure actuelle, est cependant limité.

a) Productions

Elles ont la forme bien connue :

LHS : = RHS $\not{b}_S$

- La partie gauche contient le nom de la classe entourée des séparateurs " < " et " > " :

<class-name> : = RHS $\not{b}_S$

Le nom de classe est un string composé de six caractères maximum, parmi les soixante quatre possibles (cf. annexe 2), à l'exception de " > ". Un nom de classe est toujours inscrit de cette manière, quelque soit l'endroit où il apparaît.

- La partie droite spécifie le contenu de la classe.

Elle prend la forme :

Alternative$_1$, Alternative$_2$, ..., Alternative$_n$ $\not{b}_S$ où chaque alternative définit un élément distinct de cette classe.

La restriction sur le BNF est de n'admettre qu'une "énumération simple". C'est-à-dire que la concaténation de terminaux et de non-terminaux n'est pas permise au sein d'une même alternative. Comme cette énumération peut être plus ou moins longue, il est permis de l'écrire sur plusieurs lignes physiques (par l'emploi de "* N" en un endroit quelconque de la partie droite).

Exemple : <Caract> : = <ALPHA>, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 $\not\$_S$

<ALPHA> : = A, B, C, D, E, F, G, H, I, J, K, L, M, N * N
O, P, Q, R, S, T, U, V, W, X, Y, Z $\not\$_S$

b) <u>Parenthérisation du BNF</u>

Pour être plus aisément acceptable par le système et plus lisible pour l'utilisateur, l'ensemble du BNF est contenu dans un seul bloc défini comme suit :

PHRASES COMMENT * N
< > : = ... $\not\$_S$ * N
< > : = ... $\not\$_S$ * N
: :
< > : = ... $\not\$_S$ * N
$\not\$_S$ $\not\$_S$ * N

(Remarque : "Phrases" est synonyme de "production")

2) Classes de macrodéfinitions

a) Déclaration explicite d'un nom de classe

L'écriture des macrodéfinitions est semblable à celle utilisée pour Stage 2. La différence se situe au niveau de la "parenthérisation". Les classes et leurs sous-classes sont délimitées par une structure syntaxique de bloc. A chaque bloc est associé un nom de classe et les macrodéfinitions qu'il contient forment les éléments de sa classe.

```
Format class  [class-name]  comment  * N
      :
$T  $T  comment  * N
```

(Format est synonyme de macrodéfinition).

Le nom de classe est soumis aux mêmes règles de construction que le nom de classe de paramètres actuels. Une seule différence : le caractère "]" est, cette fois interdit et non plus " > ". Le nombre limite de macrodéfinitions lexicographiquement inclues dans la classe est de cent.

b) Déclaration explicite imbriquée

Pour indiquer qu'une classe contient une sous-classe, il suffit d'imbriquer leur bloc respectif :

```
┌─ Format class  [cl 1]  comment  * N
│  :
│ ┌Format class  [cl 1 - 1]       * N
│ └ $T  $T                        * N
│  :
│ ┌ Format class  [cl 1.2]        * N
│ │  :
│ │┌Format class  [cl 1.2.1]      * N
│ │└ $T  $T                       * N
│ │  :
│ └ $T  $T                        * N
│  :
└─ $T  $T                         * N
```

Le niveau maximum d'imbrication est de cinq. Dans l'exemple, la numérotation utilisée comme nom de classe indique l'imbrication. Le contenu d'une classe imbriquée fait également partie du contenu de la classe de niveau directement supérieur.

c) Déclaration implicite de nom de classe

Un nombre de classes est réservé au macrogénérateur : [$$$$$$] . Il permet de travailler avec un nom de classe indéterminé (cf métainstructions de directives au TM).

Cette classe fictive n'est pas habituellement déclarée. Le macrogénérateur l'a créée automatiquement et considère qu'elle contient toutes les autres classes.

Il n'est qu'un seul cas explicite de déclaration de la classe [$$$$$$] , par le bloc :

Format set comment *N
:
$\$_T$ $\$_T$ comment *N

Sa seule utilité est de définir par sa classe fictive la totalité des macrodéfinitions d'un problème traité en Stage 2 classique. Car l'utilisateur n'a plus alors la possibilité de déclarer ni classes de macrodéfinitions, ni classes de paramètres actuels.

d) Syntaxe d'un template

La différence entre nos templates et ceux de Stage 2 classique, réside en ce que les indicateurs de paramètres " ' " sont remplacés par une indication précisant à quelle classe doit appartenir le paramètre correspondant : " <class-name> " pour les paramètres formels simples, ou " [Format - class - name] " pour les paramètres formels constituant des macroréférences.

Ceci détermine les formats possibles :

- le format pour l'utilisation de Stage 2 comme sous-système :

$string_1$ ' $string_2$ ' $string_3$ ... ' $string_9$ ' $string_{10}$ $\$_S$ comment *N

- le format pour l'utilisation de Stage 2 étendu devient :

$string_1$ { <class-name> | [class-name] } $string_2$ { <class-name> | [class-name] } ...

... $string_9$ { <class-name> | [class-name] } $string_{10}$ $\$$S comment *N

Inutile de préciser que ~~ces~~ [tout] nom~~s~~ de classe ~~doivent avoir été déclarés au préalable~~ [référence doit finalement être déclaré] et que les caractères réservés au sein d'un template sont : $\not\$_S$, ' , <, [, *N

e) Définition du corps d'une macrodéfinition

La routine d'expansion est constituée à l'aide des strings et des métainstructions nécessaires pour diriger la construction d'une ou de plusieures lignes de texte. Chaque ligne est terminée par :

--- --- $\not\$_T$ comment * N

(ou) --- --- * N

Outre " $\not\$_T$ " et " * N", le caractère d'escape (" e ") annonçant la présence d'une métainstruction est réservé.

~~Une définition sémantique~~ [Un corps de macro définition] se termine par une ligne restreinte à :

$\not\$_T$ comment * N

(ou) $\not\$_T$ $\not\$_T$ comment * N (1)

(1)(Lorsque la macrodéfinition est la dernière d'une classe)

C) Métainstruction.

1) Directives au TM

La métainstruction :

e C [class-name]

écrite à la fin d'une ligne du source ou du généré spécifie la classe des templates pouvant correspondre à cette ligne.

Si l'utilisateur ne désire pas proposer la ligne comme macroréférence possible, il peut utiliser les métainstructions :

- ~~E~~ [e] S (pour les lignes du texte source)
- eF_1 (pour les lignes du texte ~~des définitions sémantiques~~ [des corps de macro définition]) écritent à la fin de la ligne. Ceci soulage le macrogénérateur de recherches inutiles.

2) Directives à l'algorithme de génération

Ces métainstructions sont celles de Stage 2. Nous reportons le lecteur au manuel de Poole et Waite.

Rappelons simplement que les "conversions de paramètres" forment un langage de manipulation de string et de gestion des valeurs associées aux paramètres actuels.

Les fonctions - systèmes assurent, elles, des opérations de branchement, d'itération, d'assignation et de calcul d'expressions.

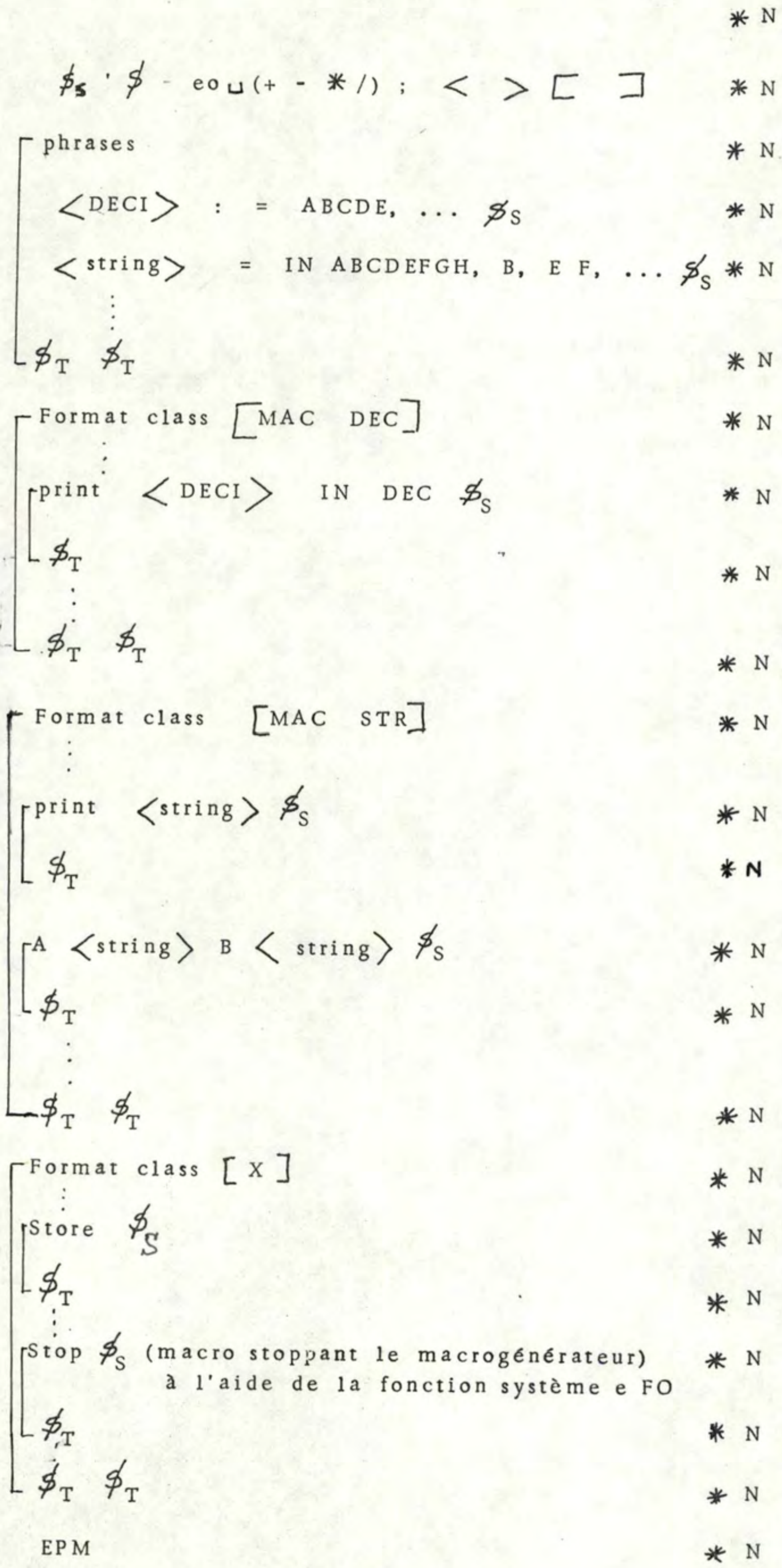
Nous avons repris la syntaxe et la sémantique de ces instructions dans les tableaux un et deux de l'annexe 2.

Nous avons décrit ainsi l'entièreté du métalangage. La partie déclarative est séparée du texte source proprement dit par l'en-tête.

EPM comment *N (End of Primary Material)

D) Exemple

L'exemple présenté au § 2.1 peut s'écrire schématiquement :



print IN ABCDEFGH IN DEC ec [MAC DEC] $S

print ABC DE e C [MAC STR] $\not{\$}_S$

ABBEF eC [MAC STR] $\not{\$}_S$

Store e C [X] $\not{\$}_S$

Stop e C [X] $\not{\$}_S$

- Pour une macroréférence, le nombre de templates possibles est plus réduit.
- L'algorithme dispose des paramètres actuels pouvant apparaître pour chaque paramètre formel dans chaque template; les hypothèses de longueurs sont, à présent, inutiles.
- Le passage des paramètres pour la macroréférence "A' B' C $\not{\$}_S$" s'effectue correctement.

Les points soulevés dans le § 2.1 semblent, à première vue, avoir trouvé leur solution .

Les notions de base étant décrites, nous pouvons examiner l'implémentation du macrogénérateur Stage 2 et en tirer des conclusions plus concrètes.

Nous renvoyons à l'annexe 1, le lecteur désireux de comparer syntaxiquement un même programme destiné au macrogénérateur Stage 2 dépouillé et au macrogénérateur Stage 2 étendu.

E) Remarque : classes de macrodéfinitions de base

Nous avons déjà souligné que la forme syntaxique des conversions de paramètres et des fonctions-systèmes demande une certaine pratique, et ce pour deux raisons :

1. Elles s'écrivent généralement à l'aide de chiffres qui rendent la signification de la métainstruction.
2. Il existe une numérotation par laquelle on spécifie au macrogénérateur quel paramètre il doit considérer. Cette numérotation est basée sur l'ordre d'apparition des paramètres formels dans un template.

Implicitement, les fonctions-systèmes travaillent, en général, sur les paramètres un à trois. Or, au niveau du texte source, nous ne pouvons pas toujours décider de la syntaxe de l'utilisation des fonctions-systèmes.

A titre indicatif, nous proposons la classe des macrodéfinitions " [BASIC] " permettant l'utilisation des fonctions-systèmes par le biais de macros.(v.annexe 1)

2.3.3 Souplesse d'utilisation

1. A la lumière du manuel de référence, il nous semble que la souplesse inhérente au macrogénérateur Stage 2 serait préservée, si nous acceptions la concaténation. (Nous reviendrons sur ce point au paragraphe 2.3.4).

2. Une des caractéristiques essentielles des macrogénérateurs est de permettre à l'utilisateur une approche modulaire des problèmes. Il peut, en effet, à chaque niveau de résolution dégager les fonctions (macros) qui lui sont nécessaires et les considérer comme élémentaires, c'est-à-dire supposées déjà définies. Chacune de ces fonctions ne se définira explicitement qu'au niveau suivant, assurant ainsi une grande transparence à l'ensemble.

Cette démarche conduira généralement l'utilisateur à considérer des classes, des types d'objets. Notre système permet, grâce aux métadéclarations, d'indiquer ces classes au macrogénérateur et par les métainstructions et les templates, d'en préciser leur utilisation.

Le métalangage a donc pour but de préciser, dans une certaine mesure, les intentions du programmeur.

La souplesse d'utilisation des noms de classes est d'abord fonction de leur signification pour l'utilisateur.

A première vue, il y a, nous semble-t-il, quatre principes que l'on peut dégager quant à l'utilisation du métalangage.

Principe 1 : Un type sera associé à une classe de paramètres si ce type ne demande pas de génération et, dans le cas contraire, à une classe de macros.

Principe 2 : Quant aux indicateurs de paramètres, ils présentent alors l'avantage de servir de mnémoniques, rendant cette fois la signification du paramètre.

En ce qui concerne les noms de classes de paramètres dans le contexte spécifique d'un template, ils précisent mieux l'idée du traitement dont ils seront l'objet.

Principe 3 : Si des macrodéfinitions sont d'utilisation standard (macros utilitaires à types de paramètres très généraux), leurs noms de classes rappelleront le type de généré auquel on peut s'attendre pour ces macrosdéfinitions, quelque soit l'endroit où il est utilisé (Les macros "de base" en sont un bon exemple).

Principe 4 : Dans le cas de macros à buts spécifiques (type(s) de paramètres précis), on peut également utiliser un nom de classe, rappellant le(s) type(s) des paramètres sur lesquels elles travaillent. (Par exemple, une fonction de calcul d'expressions booléennes serait contenue dans la classe [Bool])

Si nous désirons étendre un langage d'assemblage avec l'aide d'expressions entières, il semble naturel de penser aux types (ou mnémoniques) suivant : "Variables, Strings, caractères alphabétiques, caractères numériques; assignations, expressions, facteurs, termes, opérateurs "...

Si l'on applique le principe 1, parmi les types cités, seuls l'affectation et les opérateurs demandent une génération. On pourrait donc écrire le texte suivant (en appliquant les principes 2 et 4), sans que les corps même des macrodéfinitions soient nécessaires à la compréhension.

Format class [stat] $\not{\$}_T$

<VAR> = [ExpR] $\not{\$}_S$

$\not{\$}_T$ $\not{\$}_T$

Format class [expr] $\not{\$}_T$

[FACT] + [expr] $\not{\$}_S$

[FACT] - [expr] $\not{\$}_S$

[FACT] $\not{\$}_S$

$\not{\$}_T$ $\not{\$}_T$

Format class [FACT] $\not{\$}_T$

([expr]) $\not{\$}_S$

<VAR> * [FACT] $\not{\$}_S$

<VAR> $\not{\$}_S$

$\not{\$}_T$ $\not{\$}_T$

Loin d'être une contrainte additionnelle pour l'utilisateur, cette façon de faire conduira généralement à une facilité supplémentaire pour celui-ci.

La seconde partie de la remarque ci-dessous montrera que l'utilisation de classes nous permet de résoudre le problème de conversion complexe - réel et des tests pour l'application de cette conversion. De fait, une seule macro effectue ce travail et est appelée implicitement chaque fois que nécessaire.

Remarque :

a. Dans l'exemple ci-dessus, le template "$\langle VAR \rangle \not{\$}_S$" est pratiquement inutilisable en Stage 2 classique car un tel template s'écrirait : " ' $\not{\$}_T$ ". C'est-à-dire qu'une ligne quelconque du texte source, ne répondant pas à l'un des autres templates, serait acceptée comme une simple variable.

b. Une remarque similaire peut s'appliquer à la reconnaissance d'un template réduit à un appel de macro passée comme paramètre.

Supposons que nous disposions de deux classes de macros dont l'une est de type réel et l'autre, de type complexe. On pourrait imbriquer la classe réelle dans la classe complexe :

Format class [COMPLEX] $\not{\$}_T$

CALCUL des racines de l'équation du second degré : $\langle EXPR \rangle = 0 \not{\$}_S$

Format class [REAL] $\not{\$}_T$

$\langle VAR \rangle$: = [COMPLEX] $\not{\$}_S$

$\not{\$}_T$ $\not{\$}_T$

⋮

$\not{\$}_T$ $\not{\$}_T$

Dans la structure ci-dessus, le résultat de la macro de calcul des racines d'une équation du second degré est complexe, en toute généralité. La macro d'assignation à une variable réelle porte, en toute logique, sur des résultats réels, de même que toutes les macros de cette classe. Nous sommes donc en présence d'un problème de conversion de type.

La solution est très simple quand on prend en considération la remarque précédente.

Format class [COMPLEX] $\not{\$}_T$

Calcul des racines de <EXPR> = O $\not{\$}_S$

$\not{\$}_T$ $\not{\$}_T$

:

Format class [REAL] $\not{\$}_T$

<VAR> : = [Real] $\not{\$}_S$

:

[COMPLEX] $\not{\$}_S$

$\not{\$}_T$ $\not{\$}_T$

La conversion d'un type complexe à un type réel est assurée par la nouvelle macrodéfinition " [COMPLEX] $\not{\$}_S$".

Lors du TM de la ligne source, le template "<VAR> : = [REAL] $\not{\$}_S$" est évalué. Pour le paramètre" [REAL]" , le template " [COMPLEX] $\not{\$}_S$" va cette fois concorder avec la partie de la ligne source demandant le calcul de racines.

Le macrogénérateur assurera ainsi les différentes fonctions (par le biais de la macro de transformation).

3. L'approche modulaire permise par un macrogénérateur présente la caractéristique suivante : lorsque l'on s'attache à la description explicite d'un module, plus son niveau est profond, plus on se préocupe de problèmes propres à la macrogénération et non plus, par opposition, aux problèmes soulevés par l'application qui sont plutôt traités à l'occasion des premiers niveaux.

Ceci a deux conséquences :

a) la sémantique des noms de classes suit la même évolution, c'est-à-dire que, par exemple, on aboutit finalement à des classes du genre des macrodéfinitions de base proposées précédemment;

b) Si l'on change de machine ou, plus exactement, de macrogénérateur étendu, la logique des classes devrait demeurer. Seules les métadéfinitions seraient sans doute à modifier quant à leur syntaxe et les métainstructions à transposer en fonction du nouveau macrogénérateur sous-jacent.

4. Au point où en sont les études réalisées sur les possibilités de démonstration de la fiabilité des programmes (et spécialement si l'on considère la méthode des assertions inductives), on constate que la plus sûre garantie de la validité d'un programme résiderait encore dans la logique de conception sous-jacente à la rédaction de ce programme.

D'une part, le macrogénérateur étendu aide l'utilisateur dans sa conception, en favorisant une vue à la fois globale et modulaire; et d'autre part, le métalangage l'amène à approfondir la logique de sa découpe par l'affectation de types.

La qualité que nous espérons de cette approche et sa mémorisation aisée par l'usage de mnémoniques (noms de classes) devraient être la meilleure garantie de la fiabilité d'un travail, de l'absence d'erreur par omission.

2.3.4 Orientation syntaxique des métadéfinitions et leurs justifications

1. Classes de paramètres actuels

Pour définir le contenu d'un ensemble de strings, il y a deux modes possibles :

- par énumération
- par la définition d'une structure syntaxique commune aux éléments de l'ensemble.

Nous avons choisi l'énumération. Une structure plus élaborée exige la conception d'un analyseur syntaxique plus évolué. Tel n'était pas le but fondamental de ce mémoire qui ne constitue qu'une expérience. De fait, notre but est de tester l'effet d'un principe sur l'efficience d'une classe de macrogénérateur et non de fournir dès maintenant un outil opérationnel.

L'énumération simple ne permet pas de préserver l'indépendance totale vis à vis du texte source. Elle n'est réalisable que si l'on peut relever dans ce texte les strings passés comme paramètre.

A titre d'exemple, supposons que nous désirions transformer des programmes FORTRAN, contenant des spécifications non standard par rapport au compilateur que nous possédons; c'est-à-dire que le texte source sera volumineux, sinon, il serait plus simple d'effectuer cette transformation manuellement.

En extraire les paramètres actuels de chaque classe sera long, impossible à priori, et source d'erreurs. D'autant plus que certains paramètres peuvent être générés au sein d'une macrodéfinition et utilisés par une macroréférence interne.

Ceci tend déjà à prouver que récursion et concaténation seraient le minimum auquel on pourrait s'attendre pour la description d'une classe de paramètres.

Supposons une macrodéfinition utilitaire :

EDIT $\langle$ STRING $\rangle$ $\not\$_S$

qui est valable pour tous strings. Elle ne serait utilisable que si l'on a déclaré tous les strings possibles. La concaténation serait d'une grande facilité (**)

(**) Remarquons qu'imposer à un utilisateur la redéfinition du contenu des différentes classes de paramètres, à chaque proposition d'un texte source au sein d'une même application ("Application" doit être pris ici comme "types de traitement") ne l'amène pas à remanier la structure de ses métadéclarations (noms de classes, interrelation des classes, template et corps de macros) puisqu'elles sont spécifiques à l'application.

L'énumération présente l'avantage de définir un ensemble de strings "concrets". C'est-à-dire que ses éléments sont créés physiquement dès l'acceptation du métalangage, ce qui facilite la logique de l'implémentation et permet un débuggage plus aisé pour tous.

Nous avons choisi un BNF pour définir ces énumérations car nous tenions à garder une certaine ouverture.

Les extensions possibles du BNF (concaténation, récursivité, but not, intersection) seront présentées dans le chapitre concernant l'implémentation. Leur acceptation ou leur rejet sera fonction de l'accroissement en souplesse et de la perte en efficacité qu'elles occasionnent.

Désormais, nous utiliserons le terme "déclaration" lorsque nous ferons référence à l'énumération et "définition" lorsque nous ferons référence à une structure de plus haut niveau.

2. Classes de macrodéfinitions

Le but fondamental de nos classes de macrodéfinition (et de notre mémoire) est d'accélérer le TM. Ce qui explique :

- que nous ayons conservé la forme des macrodéfinitions de Stage 2

- et d'autre part que nous ayons partitionné le répertoire à l'aide d'une structure syntaxique de blocs qui rendait finalement l'image de la structure interne.

Pensant que cet outil était suffisamment maniable, nous ne nous sommes pas préoccupés d'ajonctions (intersection, complémentarité, ...) autres que celles déjà permises :

- union d'ensembles disjoints
- subdivision d'ensembles

Nous pensons même qu'il serait préférable que deux ensembles ne puissent partiellement se chevaucher afin d'assurer au type une certaine homogénéité.

2.4 Implémentation

2.4.1. Plan de travail

2.4.2. Gestion des classes de paramètres actuels.

2.4.2.1 Structure de la table des symbôles et représentation d'une classe

A) Table des symbôles

B) Représentation d'une classe

2.4.2.2 Possibilités d'extensions offerte par la structure de la table des symbôles

1) Concaténation

2) Récursivité

3) But Not

4) Intersection

2.4.2.3 Critique de la représentation des données

1) Key Words

a) Représentation interne

b) Construction

c) Recherche dans la table des symbôles à l'aide des Key Words

2) Autres représentations

a) Table des paramètres actuels triés

b) Correspondance par graphe (grammaire énumérative)

c) Correspondance par graphe (grammaire régulière)

2.4.3 Gestion des classes de macrodéfinitions

1) Structure des données en Stage 2

2) Structure donnée aux classes de macrodéfinitions.

2.4.4 Template Matching modifié

1) Vue générale sur l'algorithme

2) Sélection d'arbre

3) Gestion de la chaîne de paramètres et de copy-stack.

4) Recherche en table des symbôles

5) Recherche en arbre

2.4.5. Organigramme du template Matching

2.4.6. Génération.

2.4.1 Plan de travail

Nous présentons ci-après le schéma du macrogénérateur tel qu'il est implémenté.

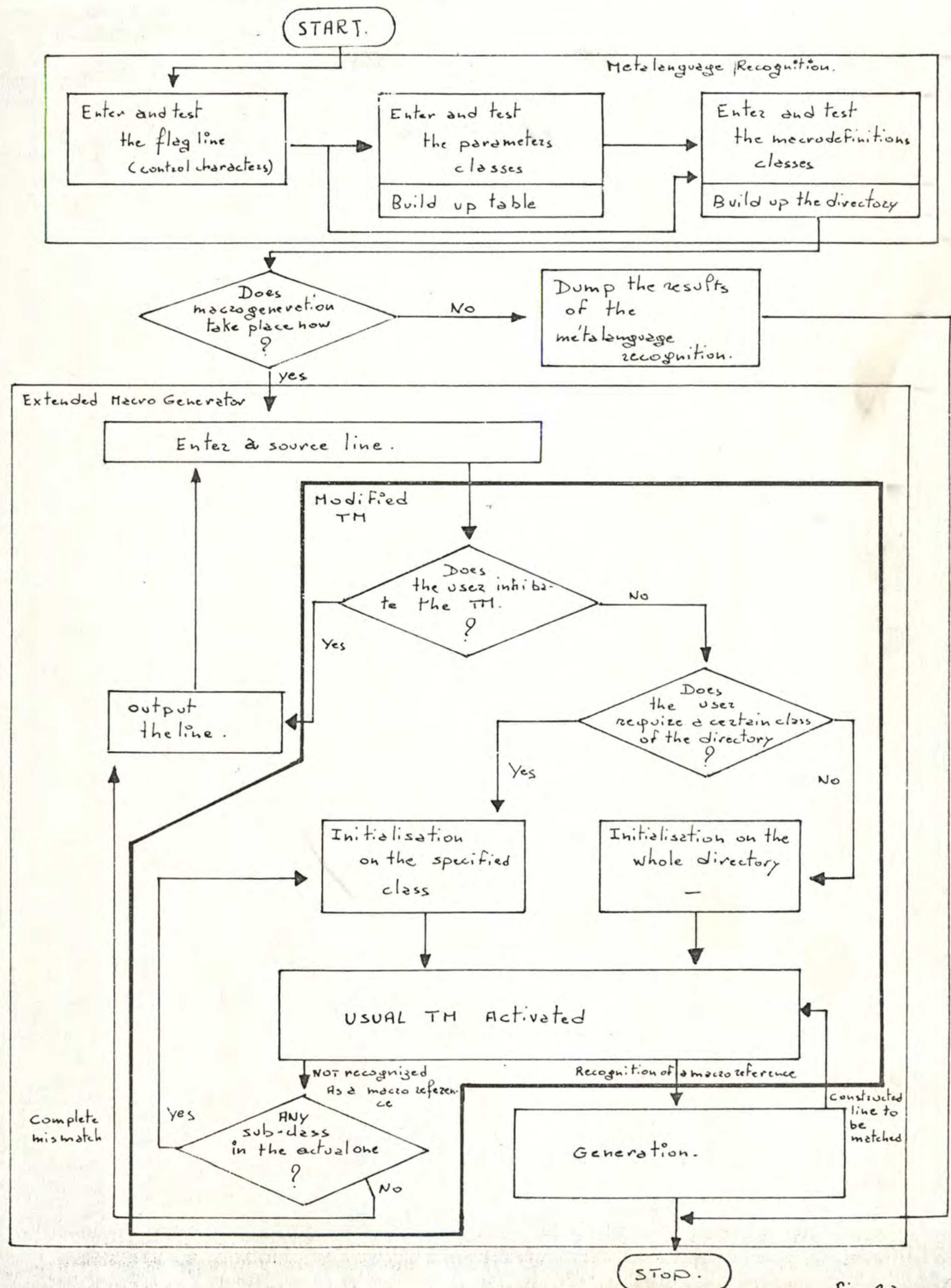


Fig. 2.3

Ce paragraphe ne reprend que les parties de l'implémentation conséquentes aux informations additionnelles fournies par le métalangage, ce qui se concrétise en 2 volets :

- sous le terme gestion, nous allons reprendre la structure interne associée aux 2 types de classes;

- dans le second volet, nous montrerons les modifications ou les additions nécessaires au template matching et à la génération.

2.4.2. Gestion des classes de paramètres actuels.

2.4.2.1 Structure de la table des symbôles - représentation d'une classe

Lors du T.M d'une référence, nous désirons une table de symbôles contenant notamment les noms de classe de paramètres afin d'assurer la reconnaissance des paramètres actuels dans la ligne.

A) Table des Symbôles.

On accède directement aux éléments de la table des symbôles, à l'aide d'une fonction de randomisation (**).

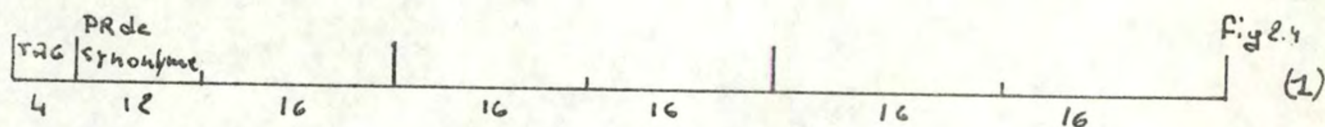
La table elle-même contient la totalité des informations dues aux méta-déclarations.

Comme il peut exister des synonymes ayant une sémantique toute différente, quatre bits ont été récupérés pour indiquer cette sémantique (tag). Excepté certaines divergences sur le contenu, la forme de la table est semblable à celle utilisée par le mémoire précédent.

Notamment, l'affectation se fait encore par le biais de la chaîne des zones libres et on ne peut affecter que par blocs fixes (Anciennement, ils étaient de cinq mots de 16 bits; ils sont portés à trois mots de 32 bits).

(**) Nous avons repris la H - fonction de Monsieur B. PIROTTE utilisant les cinq derniers bits du premier caractère du symbôle et les deux bits finaux du dernier, pour former l'adresse dans la H - table de longueur 2^7 contenant elle-même l'adresse du premier élément de la chaîne des synonymes au niveau de ces deux caractères. Les deux bits du dernier caractère forment les bits de plus faibles poids et assurent une bonne répartition dans la H - table.

Un bloc, indépendamment de son contenu se présente comme suit :



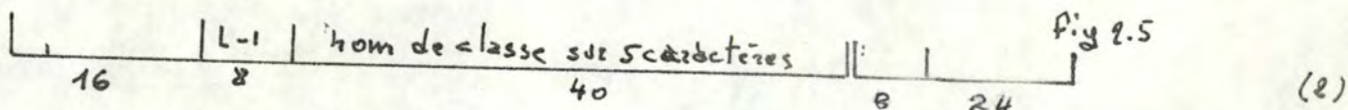
Où "PR" désigne le string "pointeur relatif". C'est-à-dire un déplacement par rapport au début de la table (12 bits lui sont nécessaire). (Nous avons essayé d'utiliser le plus souvent un adressage absolu (PA) afin d'améliorer la rapidité du système (adresse Siemens sur 24 bits)).

Outre le Tag, le pointeur de synonyme assure le chaînage des éléments présentants une synonymie au niveau de la H - Fonction. Nous donnerons les formats des blocs, parce qu'ils permettent de présenter les attributs associés à chaque entité et aiderons à la compréhension des extensions proposées au métalangage.

B) Représentation d'une classe

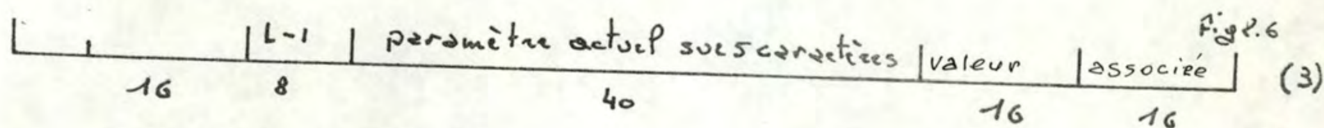
Une classe de paramètres actuels nécessite trois types de blocs que nous appellerons, pour la facilité, nom de classe, master d'alternatives et paramètre actuel.

Un bloc de nom de classe se représente par:



"L - 1" est la longueur en caractères - 1 du nom de la classe qui suit paqué sur cinq caractères maximum.
(Le premier des six caractères permis pour 1 symbôle étant testé lors de l'utilisation de la fonction de randomisation).

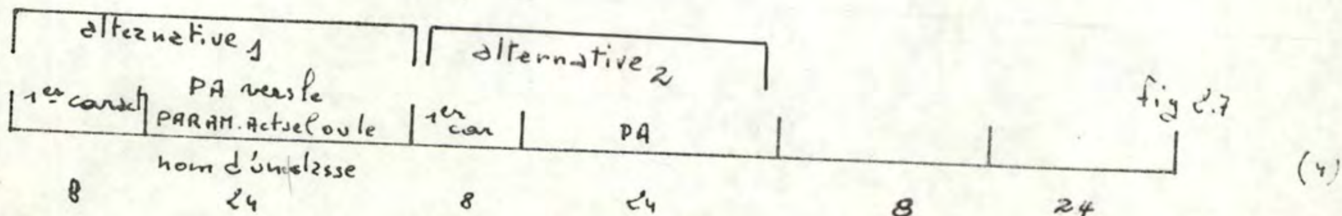
Un bloc de paramètres actuels peut s'écrire:



La même signification est donnée aux quatre zones d'information, le bloc de master d'alternatives ne suit pas la forme Standard de la fig 2.4.

Ce type de bloc assure le lien entre le nom d'une classe et son contenu. Une alternative est soit un paramètre actuel, soit un pointeur vers un autre nom de classe. Le master d'alternatives permet de ramener à un pointeur toutes les alternatives et de garder celles-ci dans des blocs standards de types (2) et (3).

La structure est :



Les deux ~~derniers~~ premiers mots correspondent à deux alternatives. La première zone indique si l'alternative est un nom de classe à considérer récursivement (zone mise à " - 1"), sinon l'alternative est un paramètre actuel dont le premier caractère est contenu dans cette zone. (Il apparaîtra lors du T.M. que cette récursion est peu coûteuse du fait qu'elle est gérée sans qu'aucune fonction ne soit appelée; seules les informations susceptibles d'être détruites sont sauvées dans un stack).

Initialement, nous espérions bloquer quatre alternatives sur deux mots afin de combattre les longues énumérations, mais nous avons été obligés de restreindre ce nombre. Nous verrons comment conserver cet avantage par l'utilisation des "Key Words" (§ 2.3.1)

Le troisième mot des blocs de types (2) et (4) sert à l'accès des différents masters d'alternatives. Il contient :

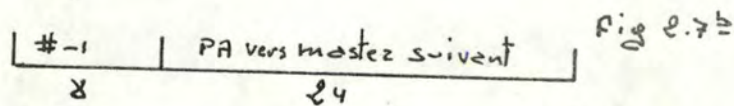


Fig 2.7 b

où "# - 1 donne le nombre d'alternatives présentes à l'adresse du pointeur. Au niveau d'un bloc de noms de classe (type (2)), cette expression décrit le premier master d'alternatives et grâce aux expressions contenues dans chaque master, nous procédons à un chaînage.

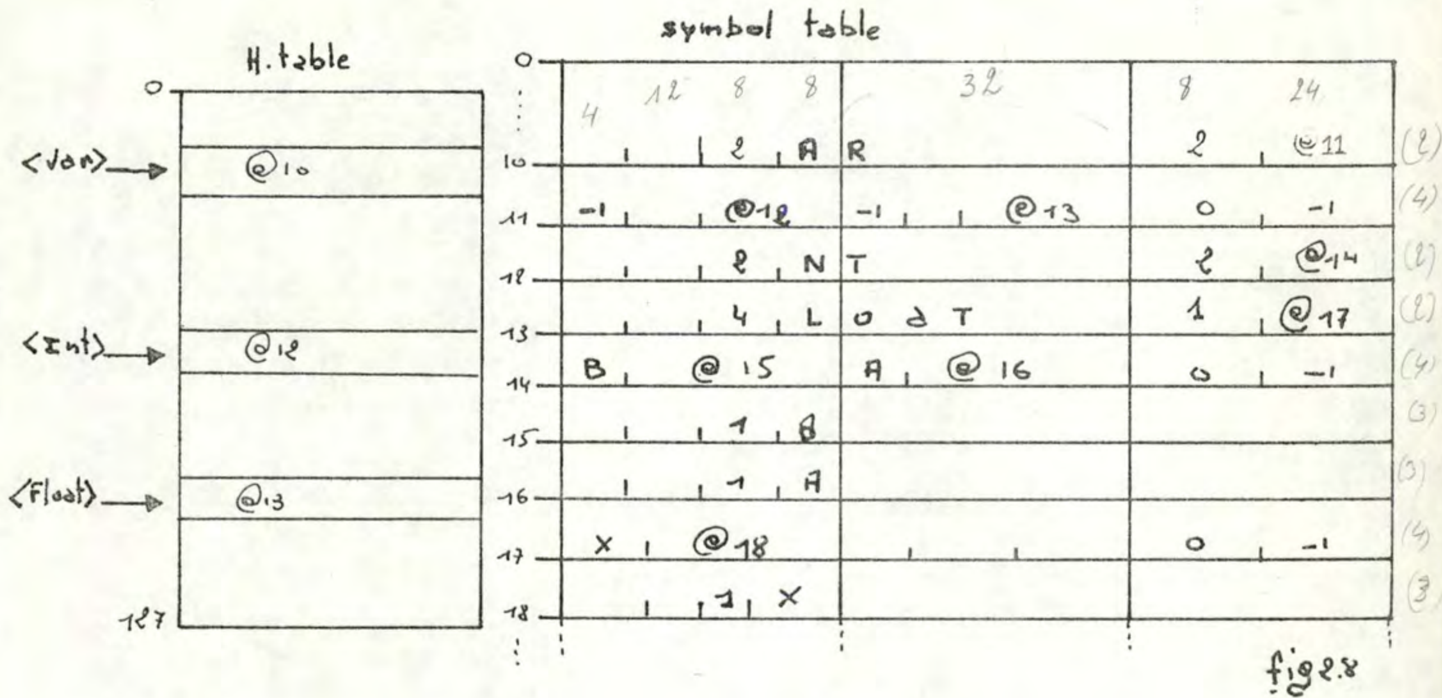
L'exemple :

<VAR> : = <INT>, <FLOAT> $\$_S$

<INT> : = BB, AA $\$_S$

<FLOAT> : = XX$\$_S$

va donner la structure suivante :



où "@N" désigne l'adresse de valeur N d'un pointeur absolu.

2.4.2.2 Possibilités d'extension offertes par la structure de la table des symboles

1) Concaténation

On peut aisément accepter la concaténation, en représentant des atomes et non plus des alternatives.

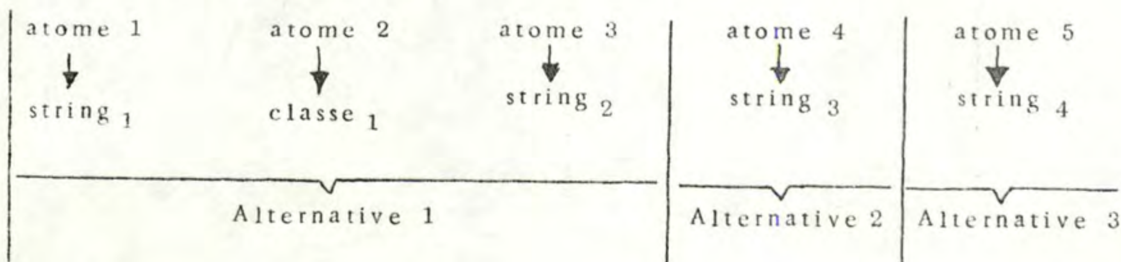
Un atome est :

- soit un nom de classe,
- soit un string, figurant dans une même alternative en partie droite de la BNF.

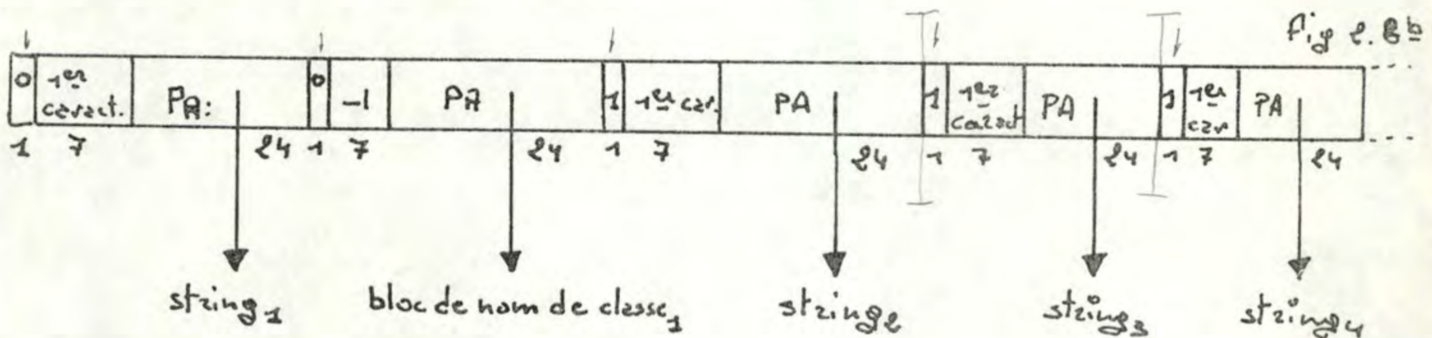
Supposons la production suivante :

<VAR> ::= string $_1$ <classe $_1$> string $_2$, string $_3$, string $_4$ $\emptyset_S$

la décomposition en atomes se présente comme suit :



Afin de rendre cette découpe, un bit dit "de virgule" est introduit dans les masters (ainsi appelé car il rend la virgule du BNF).



Les masters contiennent donc des atomes, et le bit de virgule indique la fin des alternatives.

Ce bit sera testé en même temps que la première zone contenant le premier caractère de l'atome. Cependant, une alternative peut contenir plusieurs atomes; aussi, pour représenter une production décrite à l'aide de la concaténation, le nombre de masters augmente et, conséquemment, le nombre de tests nécessaires à la reconnaissance d'un paramètre actuel. Il y a donc avantage à n'utiliser de telles productions que pour décrire la structure commune aux paramètres actuels d'une longue énumération. Ce fait s'accorde parfaitement avec l'un des avantages en souplesse cité précédemment pour cette extension.

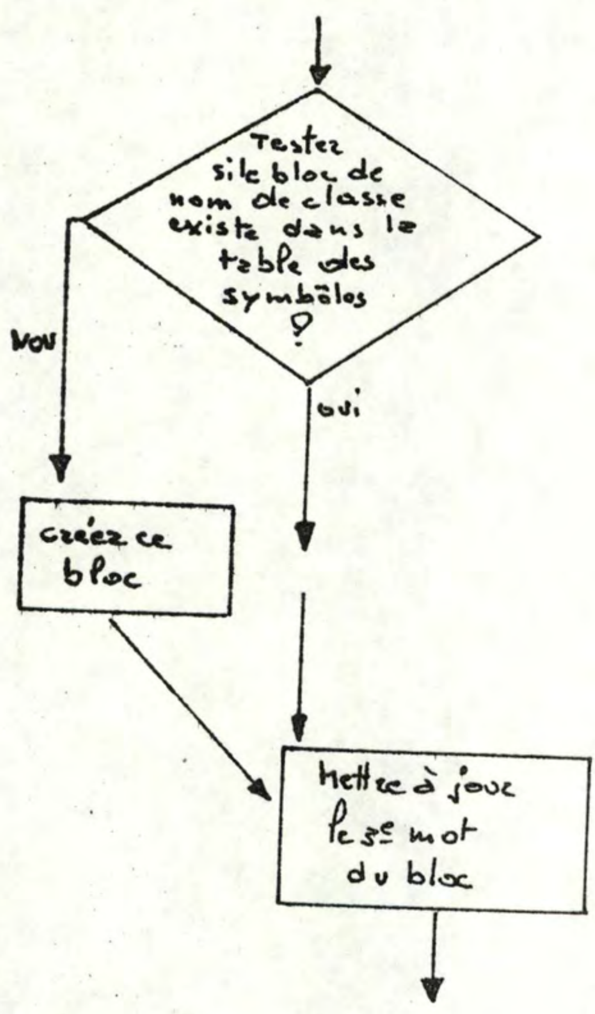
La concaténation de strings pour former un paramètre actuel est donc possible.

2) Récursivité

Nous avons constaté que l'habituel problème des définitions et des références aux non-terminaux du BNF est très simplement résolu ici.

Si la définition d'un nom de classe apparaît avant toute référence, les trois blocs de noms de classes, de masters d'alternatives et de paramètres actuels sont insérés dans la table des symboles.

Si une référence à 1 nom de classe précède sa définition, cette dernière est créée tout en laissant à vide le troisième mot du bloc (nombre - 1 d'alternatives et pointeur vers le premier master). Ce qui donne l'algorithme bien connu :



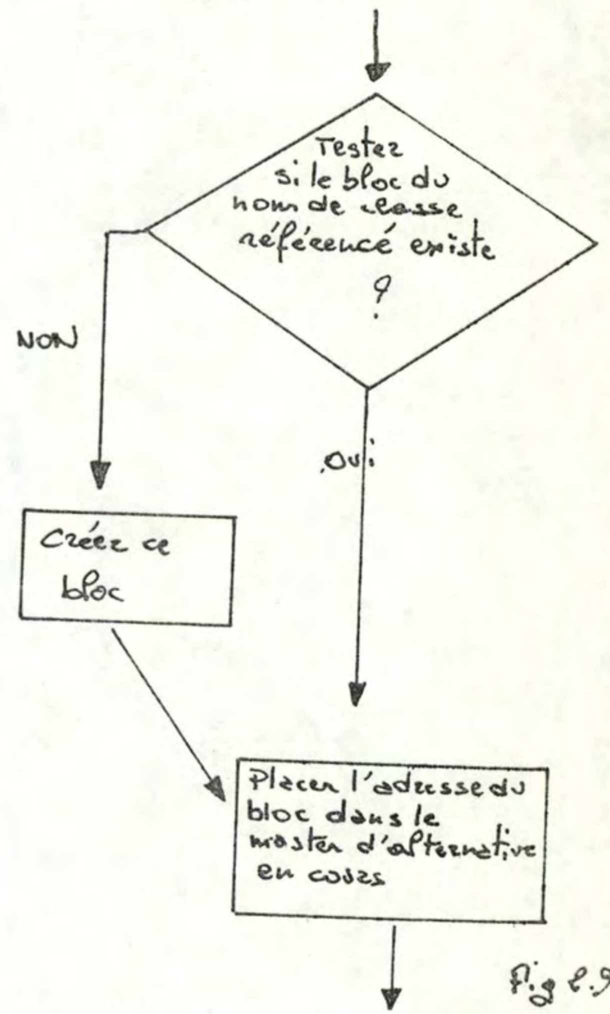


Fig 2.9

Pour réaliser ces deux traitements, il suffit d'une fonction de test et d'une fonction de création dont nous nous resservirons encore, vu que tous les blocs présentent la même structure standard.

Disposant de la concaténation et de la possibilité de faire référence à un nom de classe avant sa déclaration, les différentes formes de récusivité sont dèslors possible.

Par exemple =

$$\langle AA \rangle ::= AA \; \langle AA \rangle \; \$_S$$

$$\langle BB \rangle ::= BB \; \langle CC \rangle \; \$_S$$

$$\langle CC \rangle ::= CC \; \langle BB \rangle \; \$_S$$

Les classes se représenteront par :

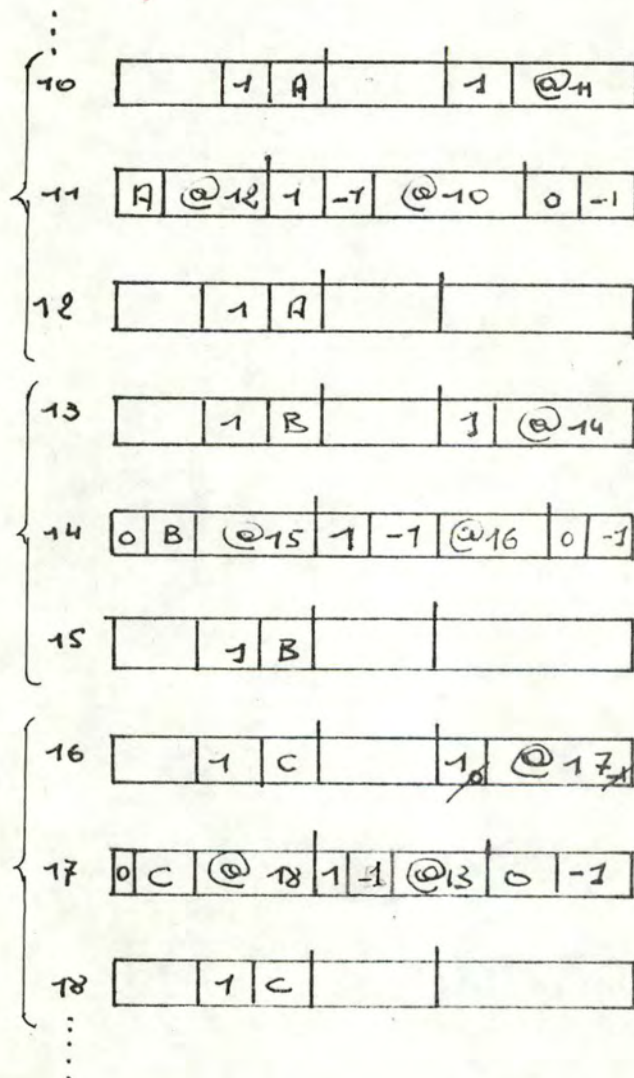


Fig 2.10.

La perte en efficacité de la récursivité repose sur la nécessité de recherche à chaque mention d'un nom de classe, par l'utilisateur, si celui-ci existe déjà dans la table des symboles.

En fait, ceci était déjà nécessaire à cause de la synonymie due à la fonction Hash.

La récursivité est essentielle à la description de certains strings.

3) "But not"

Nous pourrions aisément utiliser le concept de "But not", dû à Broocker et Morris. Il vise à fournir à l'utilisateur la possibilité de décrire une classe par complémentarité :

écrire dans le BNF :

$$\langle VAR \rangle \; : \; = \; A_1, \; A_2 \; \ldots \; A_n, \text{ But not } B_1, \; B_2, \; \ldots \; B_n \; \not\$_S$$

reviendrait à définit l'ensemble :

$$\langle VAR \rangle = C_{\bigcup_{i=1}^{n} A_i} \; \bigcup_{i=1}^{m} B_i$$

En fait, l'algorithme proposé par Broocker et Morris ne possède pas la signification exacte de cette écriture mathématique : l'ordre des alternatives, dans une définition, est significatif, parce qu'elles sont toujours utilisées de gauche à droite par la procédure d'accès qui tente de reconnaître une alternative donnée pour un paramètre formel donné

Le "But not" est employé pour exclure des membres spécifiques d'une classe définie précédemment. Ainsi, écrire

$$\langle 11 \rangle : \; = \; ABC, \; ADE, \; ACE \; \not\$_S$$

$$\langle 12 \rangle : \; = \; G \; \langle 11 \rangle \; , \text{ but not } GADE \; \not\$_S$$

signifie que <12> contient en fait "GABC" et GACE".

Cette facilité doit être implémentée de façon à tester les alternatives défendues en premier et dans l'ordre indiqué par le BNF.

Si nous désirons transposer l'implémentation de Broocker et Morris directement en terme de notre structure de données, nous pourrions implémenter le "But not" en dédicaçant un premier ensemble de Masters aux alternatives concernées par le "But not". Un bit, pris dans le dernier mot du nom de classe et des blocs de masters, indiquerait que les masters sont à considérer en mode "but not", ou non.
Ils sont placés en premier car l'algorithme suit l'ordre de la chaîne des masters.

L'exemple ci-dessus devient :

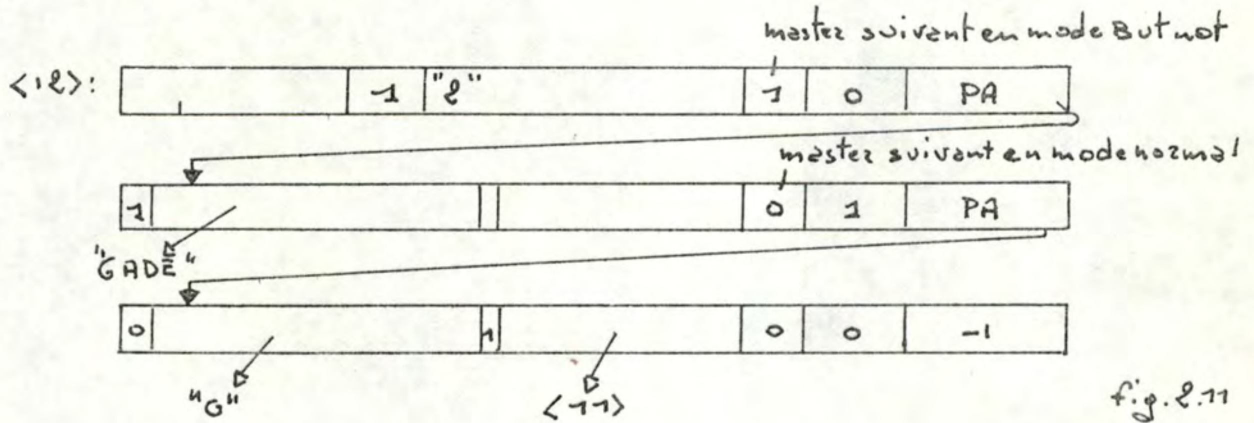


fig.2.11

Nous émettons des réserves quant à l'emploi du "But not", étant donné la prudence avec laquelle il doit être manié. Ce fait nous est démontré à travers l'exemple de Broocker et Morris :

étant donné

$$\langle A \rangle :\ = \ AB \ \not\$_S$$

$$\langle B \rangle :\ = \langle A \rangle \ G, \ \langle A \rangle \ DE \ , \ \langle A \rangle \not\$_S$$

$$\langle C \rangle :\ = \langle B \rangle \ G, \ \text{but not} \ \langle A \rangle \ G \ \not\$_S$$

alors $\langle C \rangle$ représente seulement "ABDEG", car si une alternative présente un préfixe coïncidant avec l'un des strings défendus, elle sera <u>automatiquement</u> exclue de l'ensemble admis.

Une meilleure solution d'implémentation serait de tenir compte des indications du "But not" et de recréer physiquement, par le biais des masters, le contenu réel de la classe.

4) Intersection

Un même type d'algorithme que celui du "But not" pourrait être conçu pour l'intersection.

Selon les deux énumérations indiquées par l'utilisateur, cet algorithme ne créerait physiquement que le contenu de la classe

2.4.2.3 Critique de la représentation des données

Nous désirons lever l'inefficacité inhérente à l'algorithme de T.M. Ce but sera atteint dans la mesure où l'algorithme, modifié, travaillera avec l'aide d'une structure de données dans laquelle le nombre de tests nécessaires à l'analyse sera limité

Dans notre implémentation expérimentale, notre structure d'accès est susceptible de rendre la recherche relativement longue en dehors d'énumérations courtes.

Afin d'améliorer la structure de données que nous venons de présenter, nous proposons la technique des "Key Words" de Broocker et Morris, qui est déjà partiellement réalisée.

1) Key Words

Est adjointe à chaque nom de classe, une zone contenant un bit associé à chaque symbole de l'alphabet disponible pour écrire les terminaux de BNF.

Un bit à "un" indique qu'un élément de cette classe (à un niveau de profondeur quelconque) commence par ce symbole. Un état "zéro" indique, par contre, qu'un tel élément n'existe pas.

Dès lors, le T.M. se simplifie : si le premier caractère, disponible dans la macroréférence, n'existe pas dans le Key Words, le template courant peut être abandonné (un seul nom de classe de paramètres actuels par paramètre formel dans un template).

Le principe de construction de cette zone est assez simple et permet le blocage de quatre alternatives ou de quatre atomes par master.

a) Représentation interne

La capacité d'information d'un bloc master d'alternatives est suffisante pour nos Key Words. (Sur les cent vingt huit caractères de BCPL, soixante quatre sont disponibles pour les paramètres actuels, soit les deux premiers mots d'un master).

La structure d'accès devient

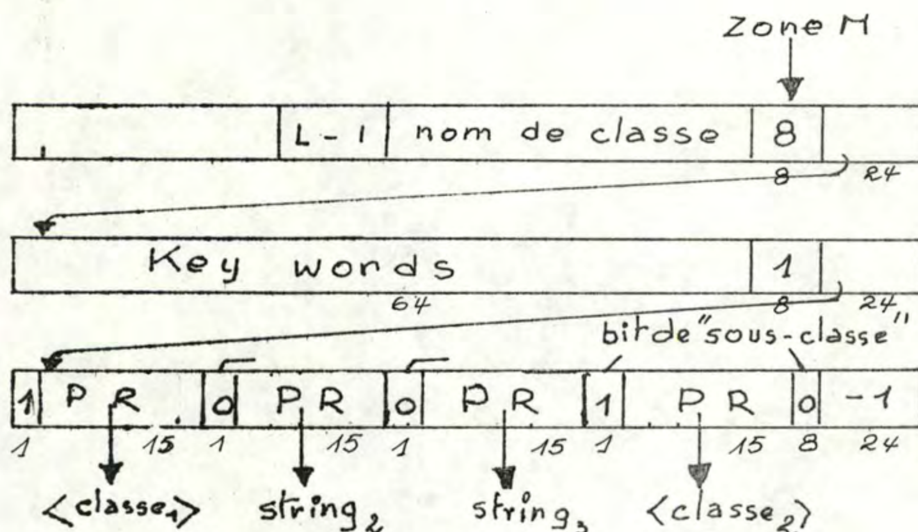


fig 2.12.

(Le bit de sous-classe permet la différenciation du type de l'élément accédé :

- Bit à "1" : bloc de sous-classes de paramètres"
- Bit à "O" : bloc de paramètres actuels"

La "zone M" , contient un nombre de valeur zéro à huit, indiquant quelles zones de huit bits, formant les Key Words, ne sont pas à zéro.)

b) Construction

Soit l'exemple :

$$\langle A \rangle ::= BC, \langle D \rangle, \langle F \rangle, XY \$_S$$

Les bits des clefs sont construits en deux passes.
Si nous notons "K_{atome}" la zone des clefs nécessaires pour l'atome considéré et "+" l'opérateur logique "ou inclusif", nous obtenons la relation :

$$K_{\langle A \rangle} = K_{\langle D \rangle} + K_{\langle F \rangle} + K_{BC} + K_{XY}$$

La construction de $K_{\langle A \rangle}$ ne peut s'effectuer dès l'acceptation de la ligne du BNF définissant $\langle A. \rangle$ Elle nécessite la connaissance des clefs $K_{\langle D \rangle}$ et $K_{\langle F \rangle}$: or, les définitions de $\langle D \rangle$ et de $\langle F \rangle$ ne sont pas nécessairement connues (problème des références et des définitions).

A ce stade, nous nous bornons à effectuer

$$K_{\langle A \rangle} = K_{BC} + K_{XY}$$

La seconde passe utilisera, après l'acceptation complète du BNF, les clefs référenciées pour exécuter :

$$K_{\langle A \rangle} = K_{\langle D \rangle} + K_{\langle F \rangle} + K'_{\langle A \rangle}$$

Par cette construction en deux passes, le problème des définitions récursives est facilement résolu :

Soit

$\langle A \rangle ::= A \langle A \rangle, \langle A \rangle \langle D \rangle, BC, \langle C \rangle \not{\$}_S$

$\langle C \rangle ::= Z \not{\$}_S$

- lors de la première passe, nous obtenons :

$$K_{\langle A \rangle} = K_A + K_{BC}$$

c'est-à-dire les Key Words :

```
AB               Z
110---   ---     0
```

- la seconde passe effectue :

$$K_{\langle A \rangle} = K_{\langle A \rangle} + K_{\langle C \rangle} + K'_{\langle A \rangle}$$

ou les Key Words :

```
AB              Z
11 0 ---    ---1
```

Ce qui nous rend bien l'image des premiers caractères permis pour toute référence à la classe $\langle A \rangle$.

D'autres propriétés sont présentées par Broocker et Morris, mais nous les jugeons hors du cadre de cette implémentation.
(Notamment, les clefs deviennent inutiles lorsqu'un string vide peut apparaître comme alternative dans le BNF; les clefs correspondantes devraient être toutes mises à un).

c) <u>Recherche dans le symbole table à l'aide des Key Words</u>

Soit <VAR> : = <INT>, <FLOAT> $\emptyset_S$

<INT> : = BB, AA $\emptyset_S$

<FLOAT> : = XX, YY $\emptyset_S$

et le Template

Calculate <VAR> - - - - - - - - $\emptyset_S$

Le string "calculate" étant reconnu dans la ligne source, nous disposons des informations suivantes :

- un pointeur spécifiant le début d'un paramètre actuel dans la ligne source;
- le nom de la classe attendue pour ce paramètre.

Ne connaissant pas la longueur réelle du string passé comme paramètre, l'algorithme se présente comme suit :

fig. 8.13

START

Application de la H. fonction sur <VAR> 1)

Accès aux keywords pour le PA du bloc de classe <VAR> 2)

test du 1er car. du paramètre actuel à l'aide des zones non à zéro des keywords, indiquées par la zone M

insuccès → 1

Rejet du template

succès

Accès, par le PA, au master suivant. 3)

Accès par PR au bloc <INT> (bit de sous-classe à 1) 4)

Accès aux keywords du bloc de classe <INT) 5)

bit du 1er caractère source à "1" ds Keywords ?

oui

Accès, par PA, au master suivant 6) (bit de "sous-classe" à 0)

Accès par PR aux paramètres actuels "1A" et "BB" 7) et 7b)

"AA", "BB" testés par rapport au string d'entrée (1er caractère non compris! keywords)

succès → 2

Conservation du template et mémorisation de l'adresse du bloc de paramètre actuel.

insuccès

NON

Accès par PR au bloc <Float> 8)

Accès, par PA, aux keywords de <Float> 9)

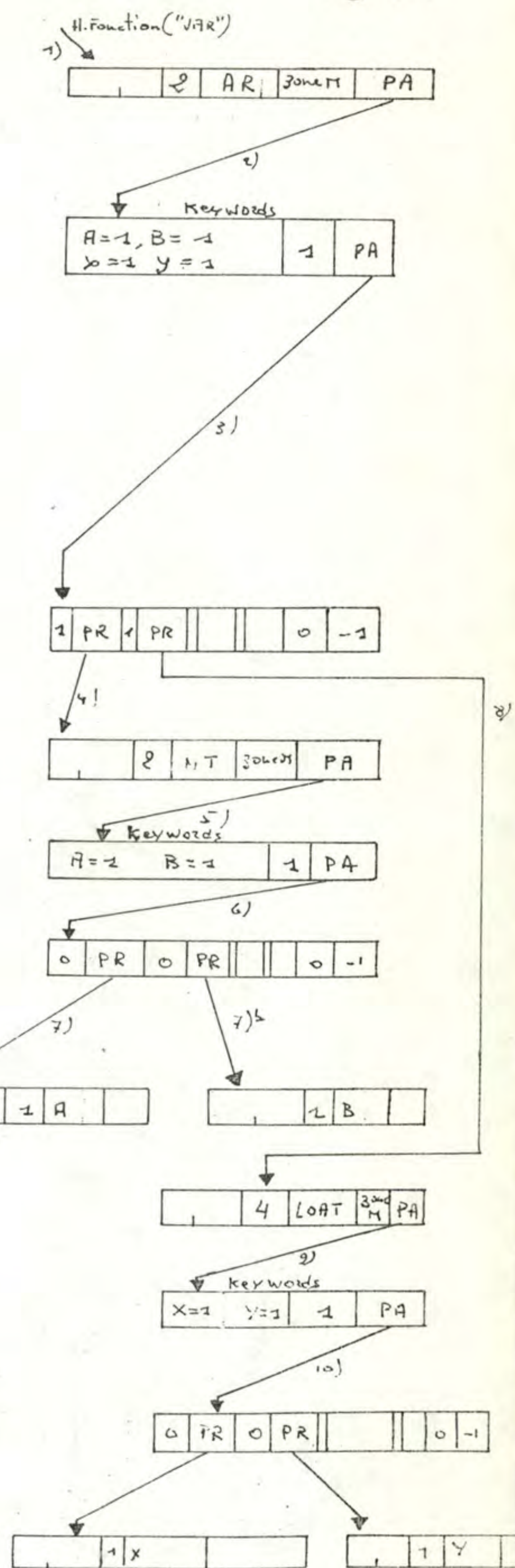
test par keywords

succès

Accès par PA, au master suivant 10)

insuccès → 1

2



(Les demis cercles indiquent la fin ou le début d'un appel récursif). (Récursivité que nous avons traduite à l'aide d'une fonction itérative et d'un stack pour sauver les données indispensables.)

2. Autres représentations

Notre implémentation repose sur un principe ; on ne connaît pas la longueur du string faisant fonction de paramètre actuel dans la ligne source.

Ce fait nous a amené à travailler directement à l'aide du type attendu : tout en parcourant le graphique correspondant à ce type, (défini par la grammaire,) on teste les feuilles, (les paramètres actuels possibles,) avec le string source, suivant le préfixe déjà reconnu.

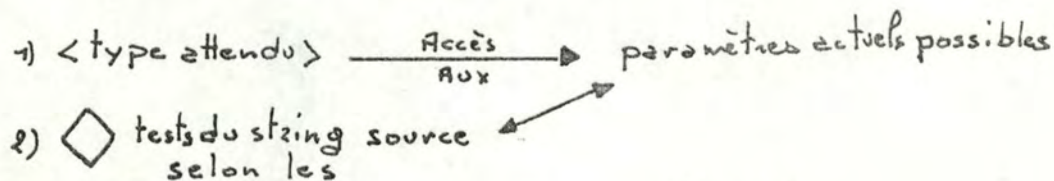


fig 2.14

On pourrait proposer la technique inverse : lors de l'acceptation du métalangage, les paramètres actuels seraient immédiatement associés à leur type et les noms de classe, associés à ceux de leurs sous-classes.

L'exemple précédent donnerait les relations :

BB => <int>
AA => <int>
XX => <Float>
YY => <Float>

<VAR> => <int>
=> <Float>

fig 2.15

Avec la ligne source :

Calculate AA : = --- --- $\not{b}_S$

nous obtiendrons la recherche

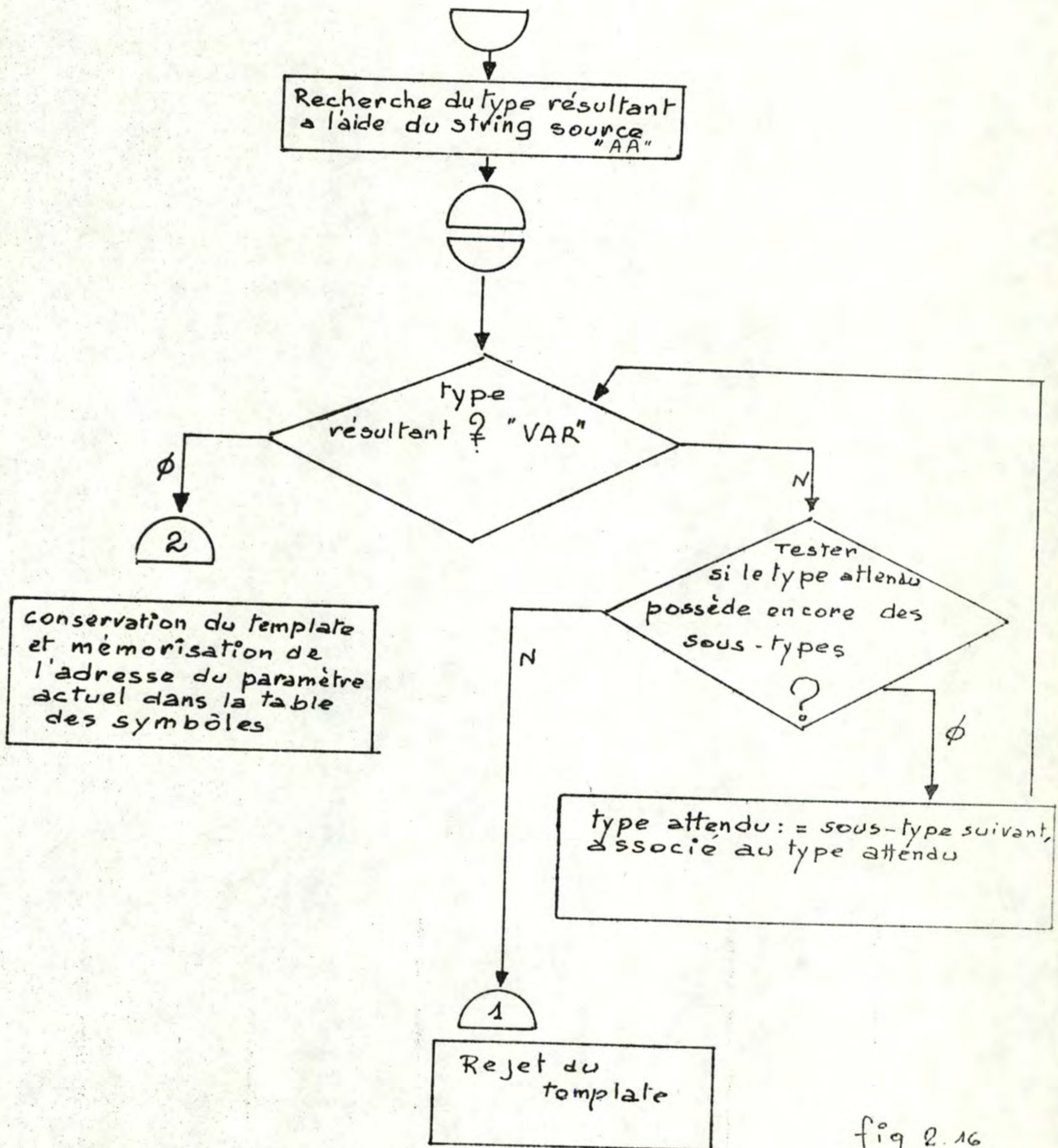
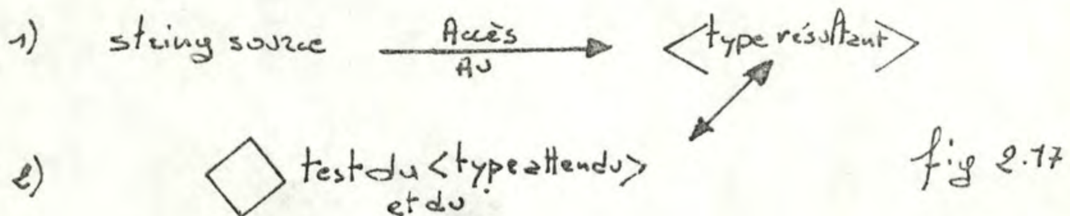


fig 2.16

C'est-à-dire que les tests portent sur le type résultant du string source et le type attendu :



Le problème à résoudre étant la correspondance

String-source ——→ <nom de classe>

puisque la longueur du string, pour correspondre au paramètre actuel, est inconnue. La structure permettant cette correspondance sera déterminante quant à l'efficacité. Nous nous bornerons à citer celles qui nous semblent les plus valables à priori.

a) Table de paramètres actuels triés (grammaire énumérative).

L'algorithme de recherche serait dichotomique sur un ensemble "concret" trié. Le type serait associé à chaque paramètre par le biais de la table.

b) Correspondance par graphe (grammaire énumérative)

L'algorithme de recherche parcourerait l'arbre. A chaque noeud, la direction prise serait fonction d e l'information contenue dans les branches issues de ce noeud. Cette information consiste en un caractère; si l'algorithme peut aboutir à une feuille, le chemin parcouru concorde avec le paramètre actuel p résent dans la ligne source. La feuille indiquerait alors le type de ce paramètre.

c) Correspondance par graphe (grammaire régulière)

Rappelons que, pour toute grammaire régulière, on peut construire un automate fini non déterministe.

L'ensemble des états forment les noeuds d'un graphe que l'on parcourt en fonction des symbôles d'entrée.

Dans notre cas, la valeur de la fonction de réponse, associée à chaque état, ne prendra plus une simple valeur binaire.

Lorsque l'automate est dans un état ne correspondant pas à un nom de classe déterminé, la réponse sera négative; dans le cas contraire, le type est donné.

Nous ne nous étendrons pas sur la structure que l'on pourrait donner à des types de grammaire plus évoluées, puisque notre métalangage doit rester très élémentaire dans un travail qui ne constitue qu'une expérience.

2.4.3 Gestion des classes de macrodéfinitions

1) Structure des données en Stage 2

Les templates, fournis à Stage 2, sont organisés en arbre où chaque caractère ou indicateur de paramètre (" ' ") correspondent à une branche.

C'est cet arbre qui constitue ce que nous avons appelé le répertoire.

Si nous avons l'ensemble de Templates :

' = ' $\not{\$}_S$
corps
$\not{\$}_T$
X = ' $\not{\$}_S$
corps
$\not{\$}_T$
X = Y' $\not{\$}_S$
corps
$\not{\$}_T$

Nous obtenons l'arbre suivant où S.PARAM et S.EØL représentent l'indicateur de paramètre (" ' ") et le marqueur de fin de ligne source (" $\not{\$}_S$ ").

Nous les distinguons car ces branches n'ont pas une représentation interne correspondant à un des caractères BCPL (nombres entiers en dehors de l'intervalle [32 - 95]) ; elle présente une sémantique toute différente.

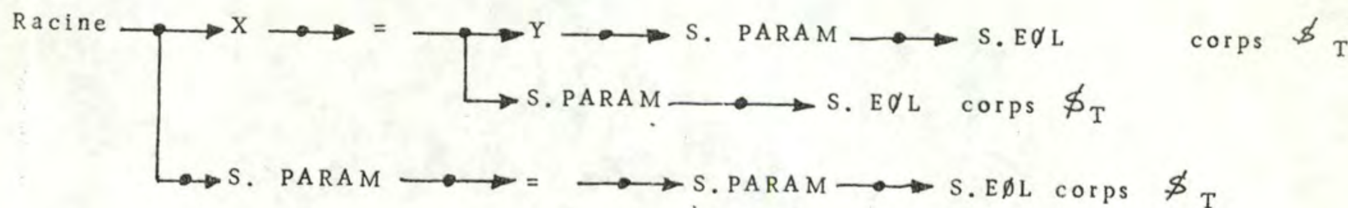


fig 2.17 b

Les flèches dans la fig 2.17 b représentent les noeuds de l'arbre. Un noeud est distingué : La racine. Il sert de point de départ à la reconnaissance. Quand une ligne concorde avec les branches depuis la racine jusqu'à - et y compris un S.EØL, elle est acceptée comme référence au template correspondant à ce S.EØL, et la routine d'expansion qui suit est "interprétée " après passage des paramètres actuels.

Rappelons que cette disposition en arbre ordonne les templates selon les règles suivantes :

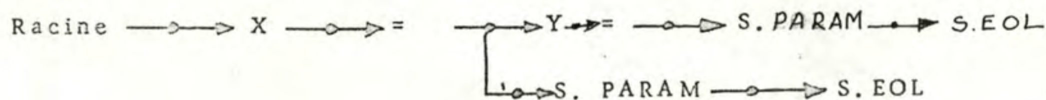
a) De 2 templates présentant un préfixe identique :

1. Si le caractère qui suit ce préfixe est un indicateur de paramètre, son template est placé en bas d'arbre :

$X = '\not\$_S$

$X = Y = '\not\$_S$

donne :

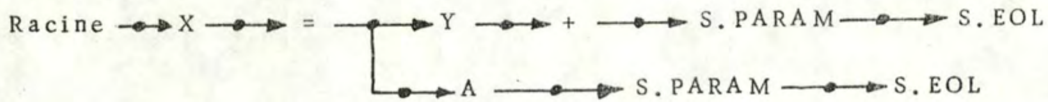


2. Sinon, le plus long template est placé au début :

$X = A\ '\not\$_S$

$X = Y + '\not\$_S$

donne :



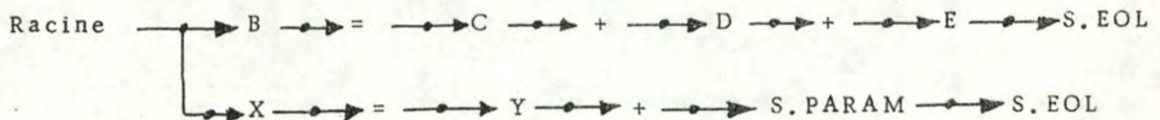
b) Les templates sans préfixe commun sont placés dans l'ordre décroissant de leur longueur.

Soit, l'exemple :

$X = Y + ' \not{\$}_S$

$B = C + D + E \not{\$}_S$

donne :



Nous reportons à l'annexe 2, quant à l'algorithme de reconnaissance d'une macroréférence qui se base sur ces remarques.

2) Structure donnée aux classes de macrodéfinition

Afin d'être à même d'utiliser un Stage 2 classique ou ~~les facilités de classes~~ le macro générateur étendu à chaque bloc :

Format class [nom] $\not{\$}_T$

⋮

$\not{\$}_T$ $\not{\$}_T$

ou Format set $\not{\$}_T$

⋮

$\not{\$}_T$ $\not{\$}_T$

va, cette fois, correspondre un arbre reprenant les macrodéfinitions de la classe.

Au début de chaque arbre, deux pointeurs absolus sont présents. Ils concrétisent la racine.

Par exemple : Format class [essais] $\not{b}_T$

' = ' $\not{b}_S$

[corps

$\not{b}_T$

X = ' $\not{b}_S$

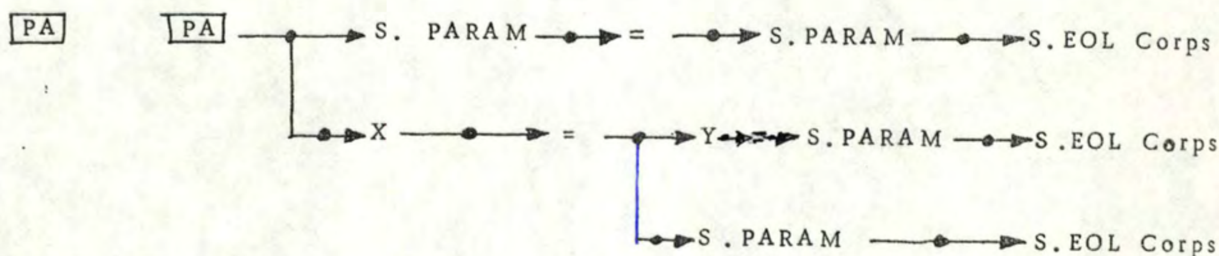
[corps

$\not{b}_T$

X = Y = ' $\not{b}_S$

[corps

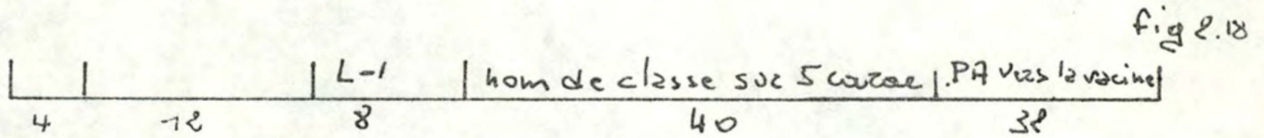
$\not{b}_T$ $\not{b}_T$



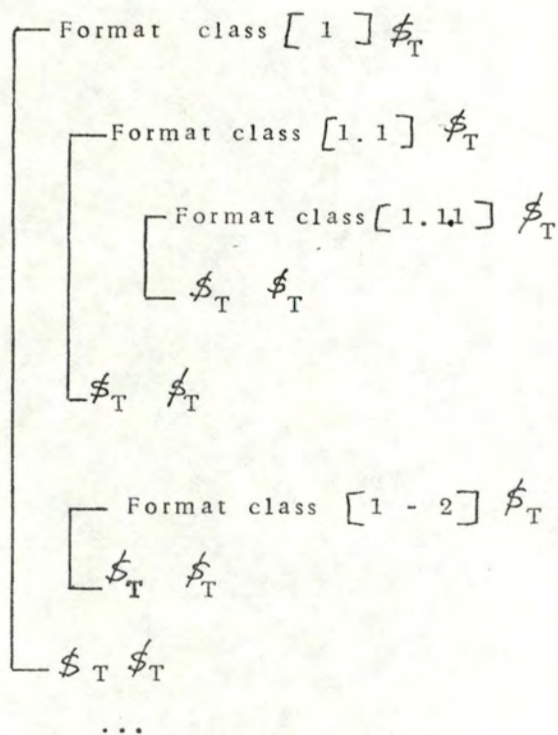
Ces deux pointeurs assurent l'imbrication :

- le premier pointeur correspond à la classe déclarée au même niveau ;
- le second pointeur à la classe déclarée comme sous-niveau à la classe en cours.

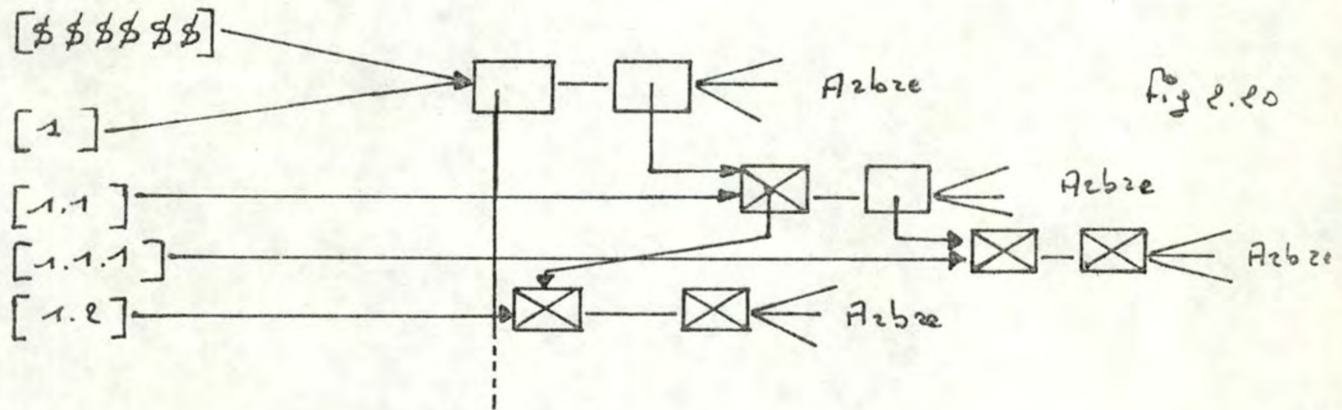
La table des symboles assure la liaison entre un nom de classe et l'arbre associé. Un bloc de classes se représente comme suit, dans cette table :



Seule la présentation, décidée par l'utilisateur, détermine les adresses associées aux pointeurs des racines.
Ainsi le texte :



présentera la structure d'accès suivante :



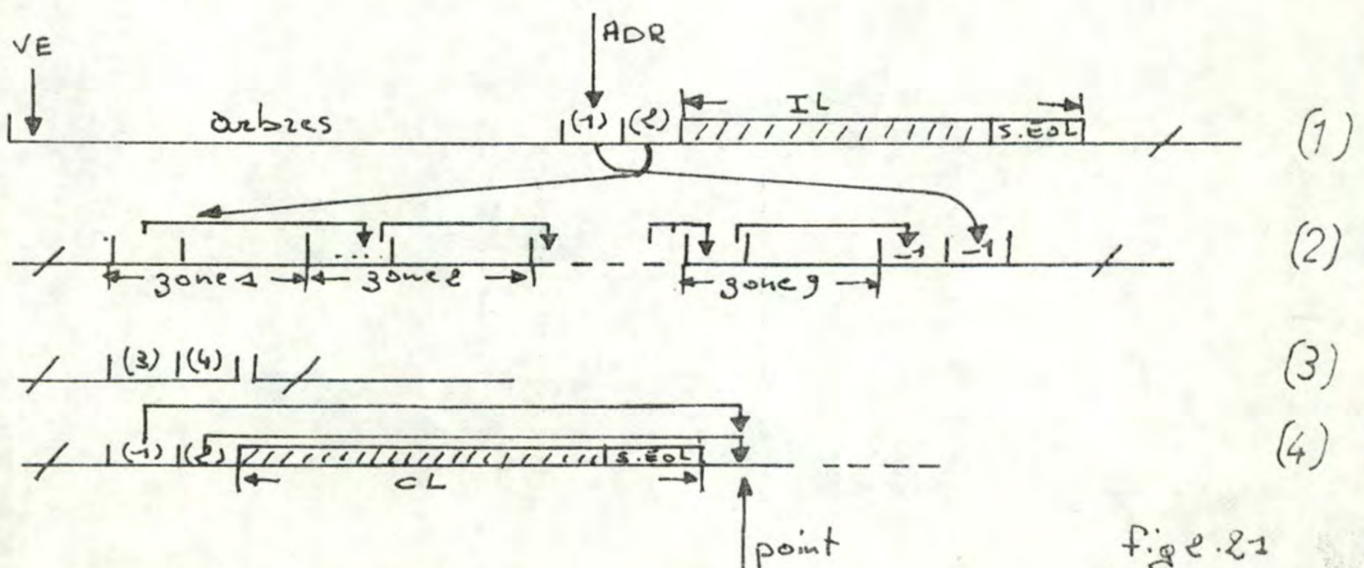
(Rappelons que le nom de classe [$$$$$$] dans la figure ci-dessus, est inséré automatiquement)

2.4.4. Template matching modifié

1) Vue générale sur l'algorithme

Nous disposons de quatre stacks distincts :

a) Le premier ("VE") est utilisé pour l'acceptation des lignes de texte source, pour la construction des lignes du généré et pour les pointeurs donnant le début et la fin des paramètres (9 au maximum). Ce stack est resté tel que l'avait conçu Monsieur Pirotte à l'exception des pointeurs des paramètres.
Rappelons sa forme :



En VE, se trouve l'ensemble des arbres créés lors de l'acceptation du métalangage - En ADR, commence le stack proprement dit. Le cas représenté ici montre une ligne de source (1) (Input Line) après reconnaissance par l'algorithme, suivie de ses pointeurs de paramètres implémentés sous forme de chaîne (2); finalement une ligne du généré (Construeted Line) (4) vient tout juste d'être construite.

Les pointeurs marqués (1) et (2) sont réservés pour représenter respectivement le début de la chaîne des paramètres et le début de la chaîne des symboles génétés.

(La fin de ces chaînes étant indiquée par une mise à "1" des derniers éléments pointés (" - 1 "). Les pointeurs (3) et (4) servent à la gestion du stack.

Le pointeur "point" est libre et sert d'index.

b) A chaque I.L. ou C.L. est associé, pour le temps du T.M., un stack ("S.T.") utilisé pour mémoriser les branches de l'arbre dépassées pendant la recherche . ST assure les retours en arrière dans l'arbre des Templates. Le stack assure également le sauvetage des données, afin de permettre différentes recursivités internes du T.M.

c) "Tree-stack" mémorise les doublets à la racine des arbres déjà rencontrés. Il permet la gestion des métainstructions donnant des directives au T.M.

d) Copy-stack" permet la reconnaissance des macroréférences passées comme paramètres. Celles-ci sont évaluées récursivement à la macroréférence qui les contient et leur chaîne de paramètres est placée dans Copy-Stack.

Il reste encore deux pointeurs : "NC" qui donnent le noeud courant dans l'arbre actuel des Templates et "CC" qui représente le caractère courant dans la ligne testée (IL ou CL).

La figure 2.22 représente l'ordinogramme complet de l'algorithme dans le but d'en donner une vue d'ensemble. Nous n'en discuterons que les points importants.

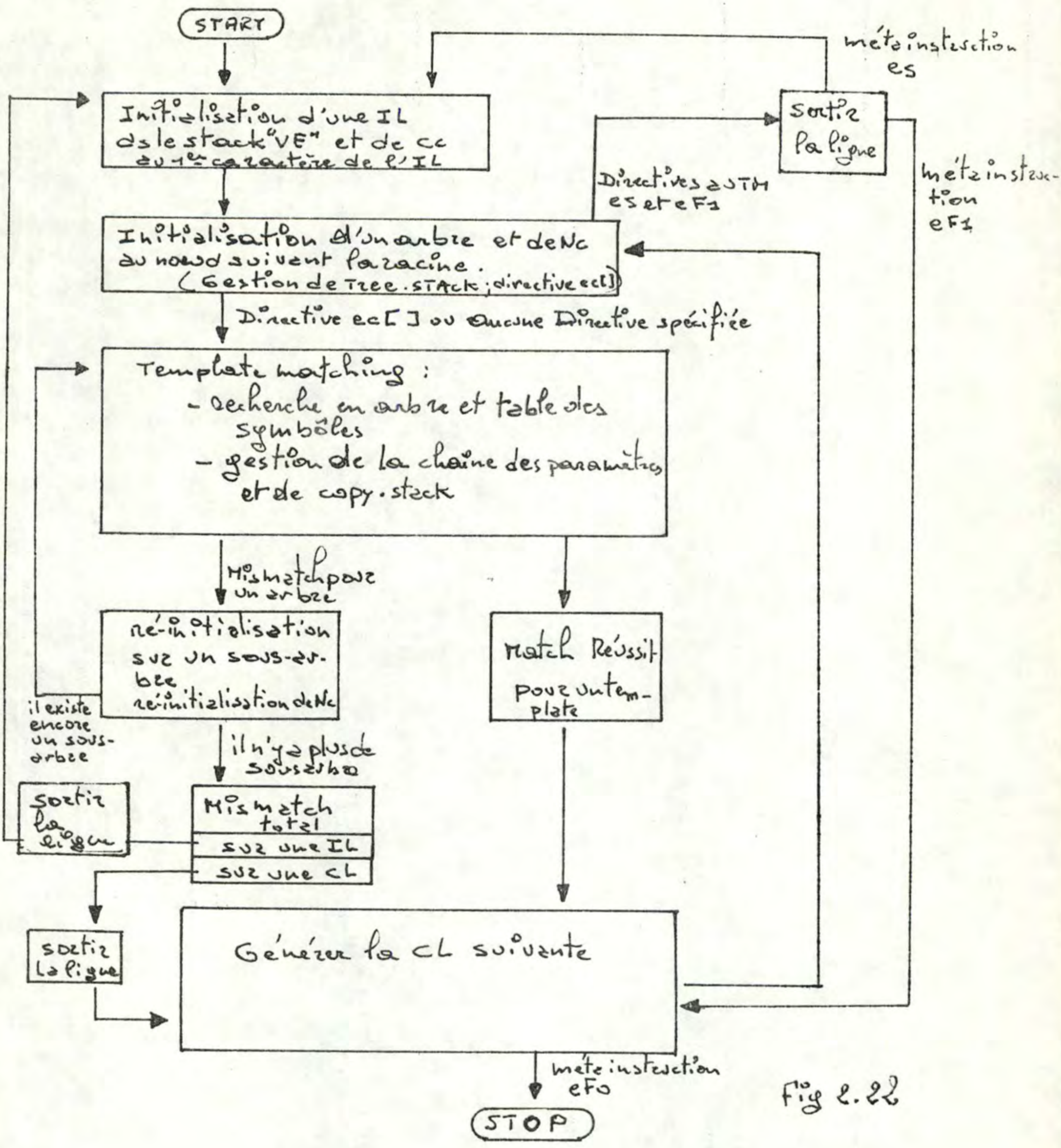


Fig 2.22

2) Sélection d'arbres.

C'est ici que les métainstructions (e F_1, eS, e C [class-name] présentes à la fin d'une IL ou d'une CL) , jouent leur rôle . (cf Fig 2.22)

a) Algorithme de sélection

Règle 1 : a) "eF_1 " rend la main à la génération, après avoir sorti la ligne sur le support extérieur

b) "eS" fait de même, mais la main est rendue à l'initialisation d'une ligne source.

c) "eC [class - name] " indique sur quel identificateur doit porter la fonction de randomisation, [$$$ $ $$] étant pris à défaut d'une des trois métainstructions, afin de déterminer la racine d'un arbre par le biais de la table des symboles - Le sélecteur introduit la racine (doublet de pointeur) au début de copy-stack :

- Si la classe a été précisée, le premier pointeur est masqué afin de refuser l'accès aux arbres de même niveaux.

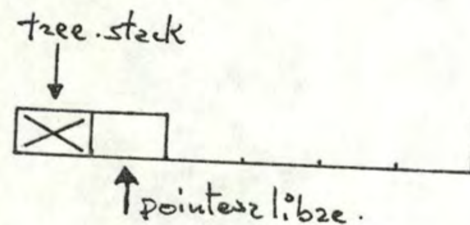


fig. 2.23

- Sinon, il utilise la classe [$$ $$$$] et ce masque n'est pas introduit

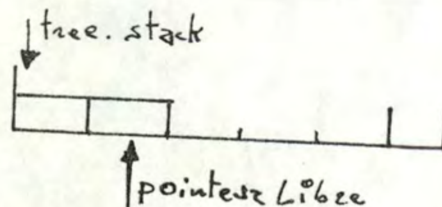


fig 2.24

Règle 2 : Lorsque le TM annonce un mismatch pour l'arbre en cours, le pointeur libre, indiquant la dernière case occupée dans copy-stack est utilisé :

- Si un sous-arbre existe, le doublet est alors inséré dans le stack, recouvrant l'adresse déjà utilisée; le pointeur libre, est incrémenté et nous retournons à la règle 2.

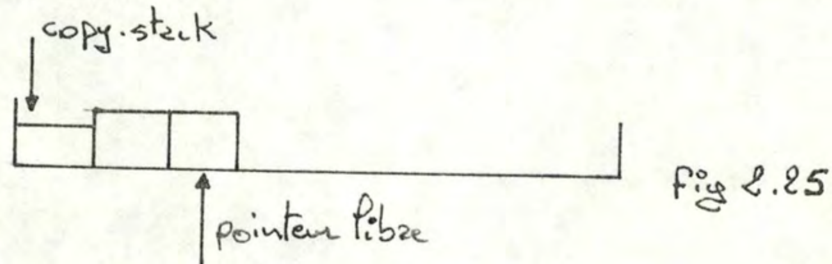


fig 2.25

- Sinon, la règle 3 est appliquée. On obtient ainsi tous les sous-niveaux.

Règle 3: Quand le sous-arbre n'existe pas, le doublet contient un masque en seconde position :

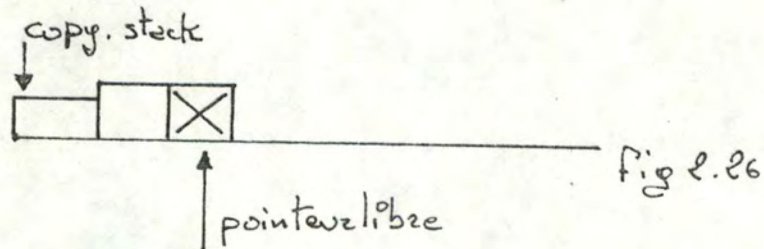


fig 2.26

Le pointeur libre est reculé d'une position et la règle 4 est appliquée.

Règle 4 : Elle effectue un travail identique à celui de la règle 2, mais les conséquences sont différentes :

- Si l'élément pointé nèst pas masqué, nous insérons la racine dans le stack, comme précédemment et retournons à la règle 2;
- Sinon, la règle 5 est appliquée.

Règle 5 : Identique à la règle 3, elle teste cependant si le début du stack n'est pas atteint.

Ce test n'apparaît que lors du recul du pointeur libre de deux unités, car nous sommes toujours certains de posséder deux éléments dans le stack dès son iniatialisation. D'où la raison de l'existence des règles 4 et 5.

Nous avons donc là un automate fini, résolvant la gestion des accès aux classes et aux sous-classes.

En fait, il y a une différence fondamentale entre une macrodéfinition appartenant lexicographiquement à une classe et l'appartenance telle qu'elle est conçue par le sélecteur d'arbres.

Si toutes sont adressables par le nom de cette classe, il y a une priorité dans leur accès par le Sélecteur.
L'exemple de la fig 2.20, de la structure syntaxique du bloc correspond à l'ensemble:

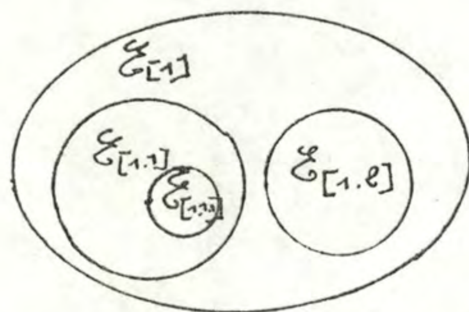


fig 2.27
Appartenance lexicographique

où $\mathcal{E}$ [class-name] représente l'ensemble des templates de la classe.

Dès qu'une racine est accédée, les tests prennent immédiatement cours sur l'arbre associé avant de considérer, à défaut d'un succès, les sous-ensembles existant.

Par exemple : si la classe [1] était proposée au TM, la suite des ensembles accédés seraient

$$\mathcal{E}'_{[1]} = \begin{cases} \mathcal{E}_{[1]} \\ \mathcal{E}_{[1.1]} \sqcup \mathcal{E}_{[1.2]} \end{cases}$$

$$\mathcal{E}'_{[1.1]} = \begin{cases} \mathcal{E}_{[1.1]} \\ \mathcal{E}_{[1.11]} \end{cases}$$

$$\mathcal{E}'_{[1.11]} = \mathcal{E}_{[1.11]}$$

$$\mathcal{E}'_{[1.2]} = \mathcal{E}_{[1.2]}$$

Donc les véritables ensembles reconnus par le système se réduisent à

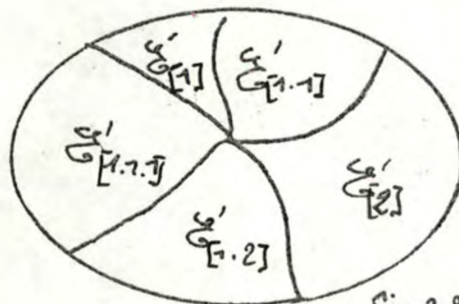


Fig 2.28 Appartenance par sélection.

b) Efficacité durant le TM

L'efficacité du T.M modifié est fonction de la manière dont l'utilisateur imbrique ces différentes classes et les référence . La manière dont sont utilisées les racines (pointeur de même niveau et de sous-niveau) peut rendre le macrogénérateur relativement lent (**).

(**) Dans un cas extrême, on peut obtenir un macrogénérateur très rapide mais trop rigide (Une seule macrodéfinition par classe et aucune imbrication), car nous en retournerions à l'identification d'une macroréférence par nom.

A l'utilisation, il y a donc avantage, lors de définition d'une classe de Templates, à éviter toutes imbrications inutiles et une référence par une directive "e C [class-name]" doit indiquer la classe la plus imbriquée possible pouvant convenir à la macroréférence.

3) Gestion de la chaîne de paramètre et de copy stack

Chaque élément de la chaîne correspond à un paramètre. Ils sont créés au fur et à mesure qu'ils sont reconnus.

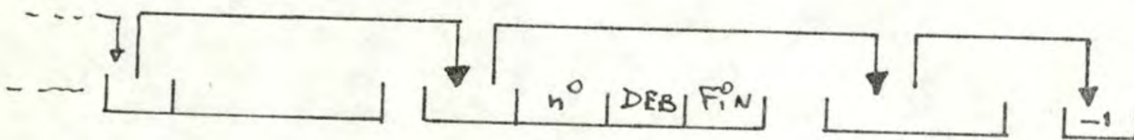


fig 2.29

La figure 2.29 reprend l'un de ces éléments. Il contient généralement quatre informations.

(1) Le "link" de chaîne

(2) Le numéro d'ordre du paramètre

(3) :(4) Les pointeurs de début et fin, du paramètre actuel, contenu dans l'IL ou la CL à laquelle est associée la chaîne.

L'information donnée par les zones (3) et (4) est toutefois différente lors du passage d'une macroréférence comme paramètre. Le template, correspondant à la ligne sous-test, indique la présence d'un tel paramètre par " [class-name] " (au lieu de l'habituel " < Class-name > ").

Le macrogénérateur étendu sait donc qu'il dispose là d'un paramètre spécial qui demande un TM. A cette occasion, il y a sauvetage de certaines données dans "ST" et la procédure du TM est réappelée.

A cet appel résultera, dans copy stack, la chaîne des paramètres concernant la macroréférence sous-évaluée.

La raison profonde de ce travail apparaftra lors de l'étude de la génération (§ 2 - 5). Nous obtenons dans copy-stack un élément de la forme

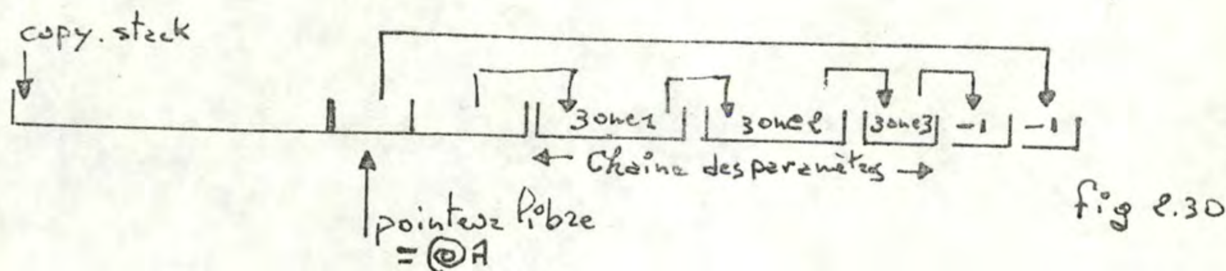


fig 2.30

Dans la chaîne des paramètres, de la macroréférence primaire, l'accès à la zone de copy-stack est assurée par l'élément :

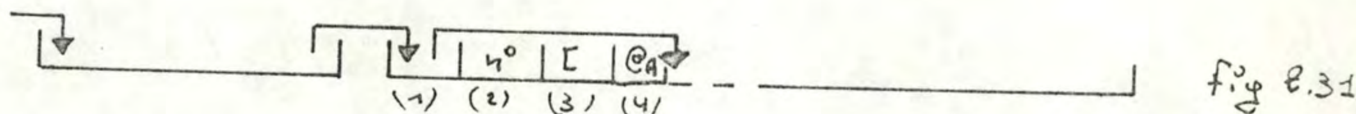


fig 2.31

Les informations (3) et (4) ont été modifiées de façon à rappeler le type de paramètre (" [") et à donner l'adresse dans copy-stack de la zone construite. Cela évitera au macrogénérateur le TM de cette macroréférence pouvant être générée un grand nombre de fois (dans une boucle , par exemple).

Il n'y a qu'une restriction à l'usage des macroréférences passées comme paramètres. Elle est inhérente à Stage 2 qui oblige l'utilisateur à fragmenter son texte source à l'aide d'indicateurs de fin de ligne (S.EØL) car, le TM d'une ligne ne se termine que grâce à cet identificateur. Il doit donc être nécessairement présent dans la macroréférence sous-évaluée.

(En fait "escape" est utilisé comme identificateur de fin de ligne.)

4) Recherche en table des symboles

Le bloc (fig 2.4 -§2.1) montre l'existence d'une certaine standardisation dans la table des symboles.

Dès qu'un symbole doit être testé, la fonction de randomisation (et peut être la chaîne des synonymes) ou un master d'alternatives, est utilisé pour obtenir le contenu informatif du bloc. Nous appelons, "contenu informatif" d'un bloc standard, la zone qui contient le tag donnant le type du bloc, le string représentant le symbole et la longueur - ou encore les informations nécessaires à la reconnaissance.

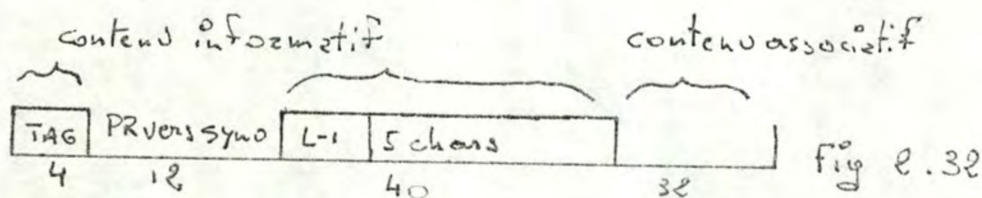


Fig 2.32

L'expression "contenu associatif" sera la zone destinée au traitement nécessitant la recherche du symbole dans la table.

a) Lors de l'accès par formule de randomisation, la même fonction de recherche est utilisée, qui permettait l'acceptation d'une référence avant la définition associée.

Cette fonction travaille sur le principe suivant :

- Comme argument, lui est passée une zone de travail de deux mots, renfermant le contenu informatif du bloc désiré de la "symbol table".
- Comme second argument lui est passé l'identificateur du symbole recherché.

Règle 1 : La fonction Hash est appliquée à l'identificateur du symbole et les différents blocs synonymes sont parcourus.

Règle 2 : Afin de déterminer le bloc recherché dans cette chaîne, le premier mot d'un bloc est comparé à celui de la zone de travail, le pointeur de la chaîne des synonymes étant masqué.
Elle assure la concordance de "tag", de longueur et du second caractère du symbole (le premier étant testé par le biais de la randomisation)
Si le premier test n'est pas concluant, la règle 2 est appliquée au bloc suivant de la chaîne, sinon nous exécutons la règle 3.

Règle 3 : Le deuxième mot du bloc en cours est alors testé à l'aide de la zone de travail. Si ces mots ne sont pas identiques, la règle 2 est reprise avec le bloc suivant, sinon l'adresse du bloc est donnée au programme appelant, afin qu'il puisse bénéficier du contenu associatif.

Quand la fin de chaîne est atteinte sans qu'aucun bloc ne réponde à la demande, ~~l'indicateur bloc ne réponde à la demande,~~ l'indicateur est retourné au programme avec la valeur "faux".

b) Quand l'accès est exécuté à l'aide d'un master d'alternatives, le même type de test est exécuté.

Le premier caractère de l'identificateur est vérifié au niveau du master lui-même.

Quand au traitement du contenu associatif des classes de paramètres, il a déjà été abordé au § 2.3.

5) Recherche en arbre

Dans ce paragraphe nous présentons :

- quelques remarques concernant la forme que prennent les indicateurs de paramètres formels au sein d'un arbre.
- les restrictions qui ont été nécessaires par suite de cette représentation.
- la justification de ces restrictions.
- l'organigramme global du T.M lorsqu'il bénéficie des informations des métadéfinitions.

1. Remarque

Lors de la création d'un arbre, le système assure déjà une partie du traitement relatif aux paramètres. C'est-à-dire que le nom d'une classe indiquant la présence d'un paramètre va être dès à présent remplacé par un pointeur absolu :

- dans le cas d'un nom de classe de paramètres actuels, l'adresse du bloc de type (2) (cf figure 2.5 du § 2.1) correspond au début de sa définition;
- pour une classe de macrodéfinitions, l'adresse de la racine de l'arbre donné.

Au sein de l'arbre, nous obtenons ainsi des éléments identiques à ceux de la figure 2.33 et 2.34 (lesquels éléments remplacent le S. PARAM en Stage 2 classique)

Fig 2.33 représentation d'une classe de paramètres actuels

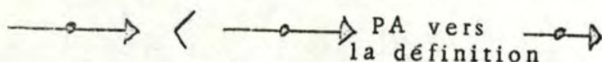
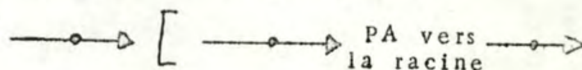


Fig 2.34 représentation d'une classe de macrodéfinitions



On remarquera que la représentation d'un nom de classe est ramenée à deux branches : " < " ou " [" suivit du pointeur absolu .Si nous faisons intervenir le tri nécessité par la construction de l'arbre, nous pouvons obtenir des structures assez complexes (cf figure 2.35) qui nous ont amenés à faire des restrictions visant à alléger l'algorithme de TM.

2. Liste des restrictions

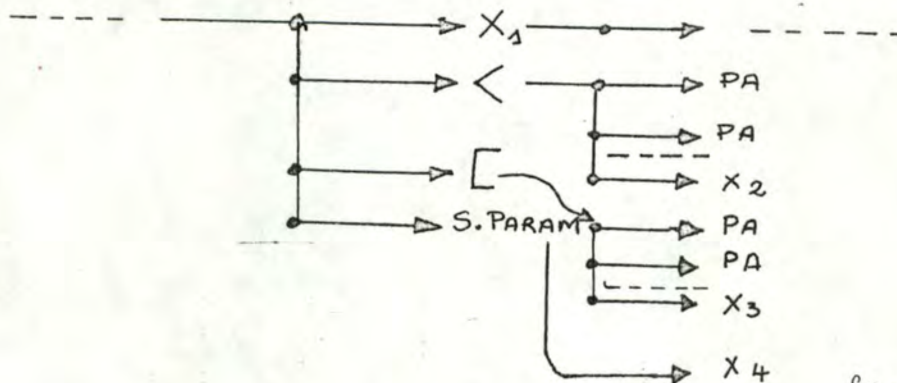


fig 2.35

"x_i" est mis pour un caractère quelconque (cf au sein des options)

- Option 1

La première des branches associées à un nom de classe sert à indiquer le type de la classe (" < " ou classe de paramètres actuels : " [" ou classe de macrodéfinitions). Ces deux caractères ont été réservés au niveau des templates afin de faciliter la reconnaissance d'un nom de classe.

Lors de l'acceptation des noms distribués, tout caractère " < " (ou " [") implique la recherche d'un caractère " > " (ou "] ") et le string ainsi délimité est considéré comme le nom d'une classe.

Plusieurs caractères " < " (ou " [") ne peuvent donc se suivre directement dans un arbre (fig 2.35 b)

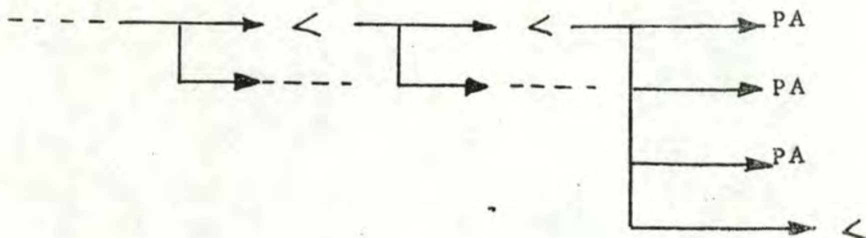


Fig 2.35 b (cas particulier de la fig 2.35 avec X_2 = " < ")

puisque, à un nom de classe, correspond une paire de branches.

Dans la fig 2.35, les caractères "X_2" et "X_3" ne peuvent être que des pointeurs associés à un paramètre.

C'est par le biais d'une représentation interne pour " < " et " [" ne correspondant pas à l'un des 64 caractères normaux connu du macrogénérateurs étendu que l'algorithme de TM sait qu'il doit reconnaître un paramètre actuel dans la ligne source

- Option 2

Dans un arbre, il a été décidé que, verticalement, n'apparaîtra jamais que, ou des paramètres contenus dans une classe, ou un paramètre sans type particulier (cas du Stage 2 classique)

On remarquera, dans la figure 2.35 que "X_1, " < ", " [", " S. PARAM" sont ordonnés verticalement au sein d'un arbre. Ce qui permettra à l'algorithme de tester les différentes branches suivant un noeud dans l'ordre croissant des difficultés de traitement (le TM utilise les branches quittant un noeud de haut en bas et sans jamais en revenir aux branches déjà testées).

- Justification de l'option 2

La raison de ce choix se conçoit à l'examen de la figure 2.36

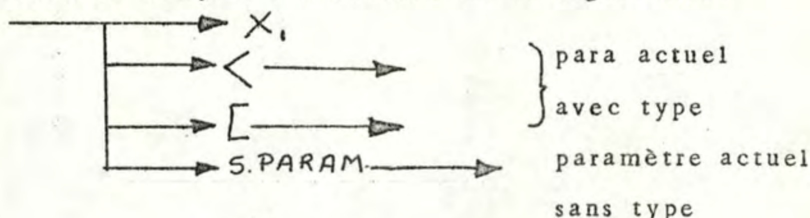
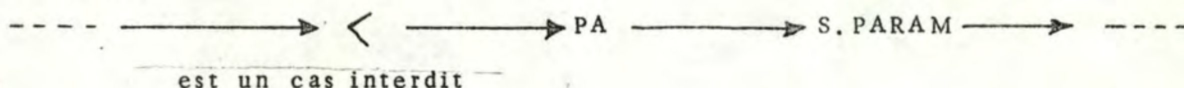


fig 2.36

Dans cette figure, à défaut de reconnaître un paramètre de type donné, l'algorithme finirait néanmoins par accepter n'importe quel string en tant que paramètre.

- Option 3

La même attitude a été adoptée en ce qui concerne le mélange de paramètre, avec et sans type, horizontalement dans l'arbre.
Par exemple



est un cas interdit

- Justification de l'options

Cette option a été admise car, finalement, même s'il est impossible à l'utisateur de définir les paramètres d'une classe, l'extension du BNF par la possibilité de concaténation lui permettrait la définition d'une classe particulière :

<ANY> : = <CHAR><ANY>, <CHAR>

- Conclusions tirées des options 1 à 3

Ces restrictions assurent l'indépendance d'utilisation du stage 2 classique et du Stage 2 étendu par les métadéclarations.

- Option 4

Une grammaire associée à un paramètre actuel est utilisée d'une manière "first fit" (le premier paramètre actuel reconnu dans la ligne source est considéré comme le dernier pouvant convenir)

- Option 5

Il en sera de même pour les macroréférences passées comme paramètre.

pour la recherche au sein d'un arbre de template (notamment

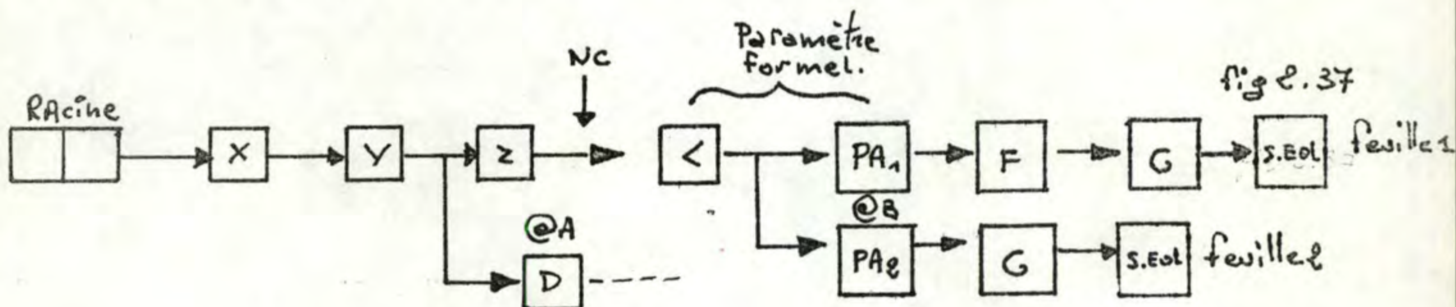
- Conséquences des options 4 et 5

Les conséquences de ces deux options sont assez lourdes par la perte en souplesse qu'elles entraînent.

a) Option 4 :

Bien qu'une solution définitive n'ait pas encore été élaborée, nous en proposons ici une approche, par une ambiguité implicite mais typique due à l'application de la règle "firt fit" portant sur les alternatives d'une classe de paramètres.

Supposons la reconnaissance de la ligne source de la figure 2.38 à l'aide de l'arbre représenté par la figure 2.37.



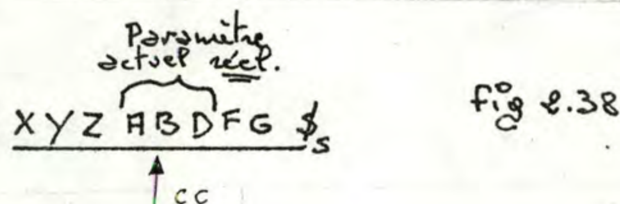


fig 2.38

Nous insistons sur le fait que la ligne est, dans la réalité, une macroréférence au template aboutissant à la feuille 1.

Dans la figure 2.37, une branche est spécifiée par un cadre reprenant son contenu; les lettres majuscules précédées de "@" et situées au dessus d'un cadre représente l'adresse physique de la branche.

Un noeud est représenté par une flèche. Les pointeurs "NC" et "CC" (noeud courant et caractère courant) indiquant l'endroit où le TM est arrivé avec succès.

Il nous reste à définir le contenu des classes de paramètres associées aux pointeurs pA_1 et pA_2 :

- pA_1 contient, dans l'ordre, les alternatives : AB et ABD.
- pA_2 contient : ABDF.

La seule branche quittant le noeud courant étant un indicateur de paramètre, l'algorithme de TM compare le caractère courant "A" avec le premier caractère de "AB", puis vient la comparaison de "B" avec le second caractère.

Réussissant à retrouver "AB" dans la ligne source, selon la règle "first fit" seule cette alternative peut convenir, sinon l'algorithme rejetera le template en cours. (En réalité, la seconde alternative est réellement l'image du paramètre actuel).

Le stack "VE" est chargé avec la valeur de "CC" (adresse de "A") et le stack "ST" avec l'adresse de la branche que l'on va dépasser :

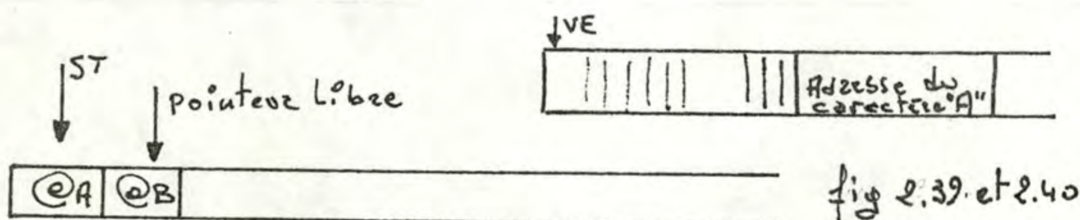


fig 2.39 et 2.40

"NC pointe à présent vers la branche [F] et "CC" vers le caractère source "D". Ceux-ci n'étant pas identiques, l'algorithme abandonne le template.
"CC" et "NC" sont remis à jour avec l'aide des éléments supérieurs de "VE" et "ST".

On reprend la règle "first fit", avec succès, sur les alternatives pointées par pA_2 et le paramètre actuel supposé est "ABCDF"

Nous avons arrangé la figure 2.38 de manière à ce que la feuille 2 soit atteinte.

Jamais le paramètre actuel et le template 1 ne seront acceptés.

La seule manière de lever ce type d'ambiguité serait de revenir dans la grammaire initiale (pA_1) dans la table des symboles pour reprendre les tests à partir de l'alternative où ils furent délaissés.

b) Option 5

Résoudre le problème soulevé par la règle "first fit" sur les alternatives d'une classe de paramètres imposerait et justifierait l'option 5. En effet, il ne peut y avoir méprise lors de la reconnaissance d'une macroréférence passée comme paramètre, que s'il y a méprise sur l'un de ces paramètres.
Par la présence de l'indicateur de fin de ligne, dans la source ou le généré, une règle "first fit" convient, puisque le début et la fin de la macroréférence sont connus contrairement au cas d'un paramètre actuel.

2.4.5. Organigramme du TM proprement dit :

Dans la situation actuelle, nous avons décrit tous les points de l'algorithme. La figure 2.41 en donne le résumé (Aucune des extensions proposées n'y sont introduites).

Les règles 1 et 2 sont identiques à celles de l'algorithme de TM de Stage 2 (CF Annexe 2)

La règle 6 remplace les règles 3, 4 et 5. Elle assure la recherche selon le principe "First Fit", appliquées aux classes de paramètres actuels et aux classes de macrodéfinitions.

START

TM.

R1

NC = noeud suivant la racine
cc = 1er caractère de la ligne.

R2

1 BQ ? cc

∅

N

cc ? S.EOl

∅

N

cc = caractère suivant
NC = noeud suivant cette BQ

RG

1 NC ? "<"

N

∅

tester toutes les alternatives pour les BQ

pas de match

Match réussi

1 BQ ? "["

N

∅

tester tous les templates pour les BQ

pas de match

match réussi

créez un élément dans la chaîne des paramètres

créez un élément dans la chaîne de paramètre

addition d'un élément dans copy-stack

NC = noeud suivant le BQ donnant le match
mise à jour des

Match

Mismatch

Fig. 2.41.b

"CC" : caractère courant de la ligne à tester

"NC" : noeud courant

"BQ" : branche quittant "NC"

"S.EOL" : Représentation interne de l'indicateur de fin de ligne.

2.4.6. Génération

Les changements nécessaires à la génération ne sont pas fort importants, exceptés les deux points suivants :

1 - a) En Stage 2, il est déjà possible d'associer une valeur aux paramètres actuels pris comme une adresse symbolique en mémoire. Nous appellerons "identificateurs de paramètres" un tel paramètre actuel.

La fonction eF_3 et la conversion ed_6 assurent l'assignation. Les conversions ed_1, ed_2 reprennent la valeur et la copie dans la ligne en construction à l'endroit de leur appel.

Un paramètre actuel passé à une macrodéfinition n'est accessible qu'au niveau de cette macro, par le biais de la chaîne des paramètres de la macro-références. (Rappelons que, dans le format "edn" d'une conversion de paramètres, "d" indique le numéro d'ordre du paramètre dans le template et "n" identifie la fonction. La conversion de paramètre travaille à l'aide du d^e élément de la chaîne des paramètres.) (Les fonctions travaillent généralement sur les éléments un à trois de cette chaîne; les valeurs associées aux identificateurs actuels ne sont donc accessibles qu'à ce niveau, à moins que ces identificateurs ne soient passés également comme paramètres à une macro de niveau plus interne).

Pour utiliser un identificateur de paramètres comme un global, il doit être généralement régénéré explicitement dans la macrodéfinition qui désire l'utiliser et les métainstructions doivent être appelées par le biais d'une macrodéfinition de "base" (cf 2.3.2. du chapitre 2).

- b) Dans le macrogénérateur étendu, par énumération simple, l'ensemble des paramètres actuels est concret.

Par l'usage d'un pseudo nom de classe réservé, une ligne de production pourrait redéfinir les identificateurs de paramètres à considérer comme globaux :

$$\langle GLOBX \rangle ::= A, BC \ \not{\$}_S$$

Une telle production assurerait une chaîne qui serait utilisée par les métainstructions d'assignation ou de recopie.

Par exemple, si dans une macro, nous désirions faire référence à la valeur associée à l'identificateur de paramètre global "BC", nous écririons :

- la métainstruction

$$e\ (Glob\ x\ (1))\ 1\ \not{\$}_T$$

: dans le corps

- et dans la grammaire

$$\langle Globx \rangle ::= BC \ \not{\$}_S$$

Lors de la génération, le macrogénérateur sait qu'il s'agit d'un global et qu'il est décrit comme première alternative de la classe de nom <Globx>

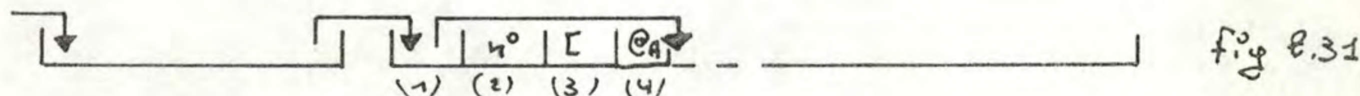
2 - La remarque (1) n'est qu'une proposition d'amélioration de la souplesse de Stage 2.

Dans le cadre de notre implémentation, il nous reste à définir l'utilisation des éléments de copy-stack par le macrogénérateur.

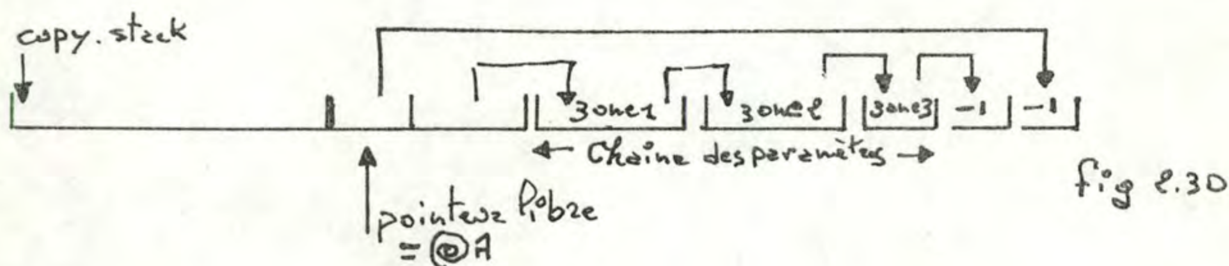
Lorsque l'utilisateur nous spécifie par le template qu'il passera comme paramètre une macroréférence, c'est que celle-ci sera toute prête à l'emploi. Seule la conversion de paramètre "edo" peut donc logiquement être utilisée (copie textuelle d'un paramètre actuel dans une ligne de construction).

Rappelons les informations stockées lors de la reconnaissance d'un tel paramètre :

- chaîne des paramètres de la ligne source (dans le stack "VE")



- chaîne des paramètres de la macroréférences passées comme paramètres (dans "copy-stack")



La demande de copie d'une macroréférence passée comme paramètre est ce que nous appelons une directive implicite au TM, par opposition aux directives explicites (σF_1, eS, eC [class-name]).
Comme indiqué précédemment, la fonction associé à la demande "edo" peut déterminer si le paramètre est, ou non, une macroréférence grâce aux informations existantes dans la chaîne de la ligne source (cf fig 2.31).

En fait, la macroréférence est ré - écrite textuellement dans la CL en cours de construction dans le stack "VE". Les pointeurs de chaînes de paramètres et de chaînes de symboles générés de cette CL sont initialisés grâce aux pointeurs placés précédemment dans copy-stack. C'est-à-dire que l'on assure ainsi une chaîne "fictive" grâce à la chaîne de paramètre de la CL déjà existante dans copy-stack.

En utilisant directement la chaîne de paramètres déjà décrite dans copy-stack, nous n'avons aucune adresse à reloger, mais simplement à mettre à jour les 2 pointeurs de début de chaîne.
Ce procédé est évidemment fort efficace, surtout si la métainstruction "edo" apparaît dans une boucle contenue dans la routine d'expansion.

Ne disposant pas d'informations suffisantes sur l'emploi des paramètres dans un corps de macrodéfinition, le contenu de Copy-Stack ne sera détruit qu'à la fin de la génération d'une IL.

Ce type de directives implicites au TM est le seul qui permette la spécification dynamique d'une classe.

3. Conclusions et projets d'extensions

- Nous pensons avoir montré que les modifications proposées sont susceptibles d'améliorer considérablement les performances d'un algorithme de TM dans un macrogénérateur de type Stage 2.

Nous regrettons beaucoup de n'avoir pu obtenir des comparaisons chiffrées de performance : nous ne pouvions, faute de temps, conduire notre implémentation à terme. Il n'en reste pas moins que les arguments développés dans le texte permettent, nous semble-t-il, d'espérer une amélioration substantielle.

- Cette expérience menée à terme, et pour autant que les résultats obtenus correspondent à notre attente, il serait nécessaire de développer le système selon les directions que nous avons déjà indiquées, en particulier :
- permettre la concaténation dans le métalangage
- abandonner la règle "first fit" (option 4) grâce à un stack supplémentaire permettant de poursuivre la reconnaissance d'un paramètre actuel au point où on l'avait arrêtée

- Améliorer la détermination des paramètres actuels, en remplaçant la technique de scanning par une recherche basée sur un graphe traduisant la grammaire.

- Ensuite, d'autres extensions seraient envisageables.

On pourrait étudier l'opportunité de ne plus distinguer les classes de paramètres des classes de macrodéfinitions, ce qui permettrait sans doute une amélioration de la souplesse. Une autre ouverture est proposée par Symple, (Syntax Macro Preprocessor for Language Evaluations) où un BNF décrit la grammaire du texte source jusqu'à un niveau tel que l'on puisse cerner les unités syntaxiques, pouvant contenir une macroréférence, et où l'on dispose par unité d'un ensemble de templates admissibles.

Cette technique nous libérerait de la contrainte des marqueurs de ligne.

- Mais, notre objectif essentiel étant l'amélioration des performances, il nous semblerait plus utile, d'envisager d'abord certaines modifications à l'algorithme de macrogénération.

En premier lieu, les macrodéfinitions n'étant pas, dans un système tel que Stage 2, générées dynamiquement, ne pourrait-on envisager de "précompiler" les routines d'expansion, de manière à transformer ces dernières en un texte directement exécutable, dont le produit serait le remplacement lui-même. De cette façon nous éviterions leur interprétation lors de chaque macroréférence.

Une autre voie que nous avons laissée totalement inexplorée serait de tenter d'appliquer une méthode du type "table de précédence" en utilisant les différents atomes significatifs des templates fournis pour générer une telle table à l'aide de laquelle s'effectuerait le TM.

- Comme conclusion finale, nous serions tentés de dire que notre travail nous a conduit à considérer que la lenteur des macrogénérateurs existant n'est peut être pas inhérente au principe de macrogénération mais bien plutôt à la façon dont ce principe est généralement appliqué. Nous pensons qu'un vaste champs d'investigation s'offre au chercheur désireux d'aborder le problème de l'efficience des macrogénérateurs. Et qu'une telle étude, serait non seulement intéressante sur le plan de ses résultats pratiques mais, croyons-nous, déboucherait de plus sur une meilleure compréhension de mécanismes essentiels au traitement de l'information.

Annexe 1

a) Exemple de classe de macrodéfinitions de base

b) Exemple syntaxique complet

a) Exemple de macrodéfinitions de base

Au début de notre stage , nous avons voulu nous familiariser avec les outils solftwares dont nous disposions, principalement :

BCPL et Stage 2.

Nous proposons ici l'ensemble des macrodéfinitions de base que nous avions alors choisies.

La classe [BASIC] qui les présentent se subdivise en deux sous-classes :

1)- [EDT - DBG] dont le contenu fournit des facilités d'impression et/ou constitue une aide à la mise au point de l'écriture des corps de macrodéfinitions. (Stage 2) ne détecte pas les fautes de logique).

2)- [FNSYST] reprend les fonctions utilitaires concernant uniquement la macro-expansion.

Remarques : - A chaque déclaration de la classe [basic] nous laissons le soin à l'utilisateur de redéfinir et préciser les classes <string> et <number>.

- La ligne de contrôle suivante est utilisée dans cette liste :

$ ¢ e ¢ (+ - * /), < > [ ]

Format class [Basic]

Format class [EDT - DBG]   Edition - Debugging

Write out <string> $

// e1∅ eF1 $ [Edition de message sans rescan
$

Error : <strings> $

e10 $ [Edition de message d'erreur avec rescan
$

Let copy source until <string> $

eF21 $ [Edition sans rescan jusqu'à détection d'une en-tête
$

Let delete source until <string> $

eF20 $
$

Output <number> Lines beginning with <string> $

e10 cF7 $

e2∅ $

eF8 $
$

print value of <string>

// value of e10 is e11 eF1 $ édition de la valeur associée à un paramètre
$

Evaluate expression <string> $

// value of e10 is e14 eF1 $
$

How long is <string> $

e15 $
$

Break list <string> $

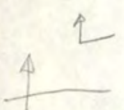
e10 e17, $

// e10 $

eF8 $
$

```
peel characters off <string> $
e10  e17 $
// e10 $
0F8 $
$
Find internal representation of <string> $
// internal representation of  e10 is e18  eF1 $
$
Tabulate <string> ,  <string> ,  <string> ,  <string> $
eF1 $
111 --- 1      2 --- 2      3 --- 3      4 --- 4 $
$$
Format class  [FNSYST]          Fonctions systèmes.
Let adress <string>  contain  <string> $
eF3 $                           Assignation à un paramètre d'une valeur
$
Skip  <number> $
eF4 $                           Saut inconditionnel
$
IF <string>  ID <string>  Skip   <number> $
eF50 $                          Saut conditionnel à une identité lexicogra-
$                               phique
IF  <string>   NOT.ID  <string>  Skip   <number> $
EF51 $
$
IF <number> <<number>    Skip    <number> $
EF6 - $
$
IF <number> <= <number>  Skip   <number> $
IF  e10   < e20    Skip    e30 + 1$
eF6Ø $
$
IF <number>   =  <number> Skip <number> $
eF60 $
$
```

```
IF <number> Not. = <number> Skip <number> $
eF61 $
$
IF <number> > <number> Skip <number> $
eF6 + $
$
IF <number> >= <number> Skip <number> $
IF e10 > e20 Skip e30 + 1 $
eF60 $
$
END $                    Arrêt de la macrogénération
// END eF1 $
eF0 $
$
Exit $                   Saut inconditionnel à la fin de la routine
eF9 $                    d'expansion
$ $
$ $
```



b) Exemple syntaxique complet

Cet exemple est extrait du Manuel de Stage 2.

Nous le présentons ici en deux volets :

- le premier est destiné au macrogénérateur Stage 2 classique
- le second, au macrogénérateur étendu

Un même texte source traduit évidemment le même généré que nous ne reprenons pas.

Les macrodéfinitions ont pour but de traduire en langage d'assemblage des expressions arithmétiques simples.

1) Texte nécessaire pour l'utilisation du Stage 2 classique :

```
$'$e  Ø⊔(+ - * /)
 Format Set
- END $
    END eF1 $
 eF Ø $
  $
-'EQU ' $
 eF3 $
 $
- SKIP '$
 eF4 $
 $
- IF ' = ' SKIP ' $
 eF5 Ø $
 $
- IF ' NE' SKIP '$
 eF51 $
 $
- VAR ' , '$
 eF1 $
 1111 Réserve 222222 $
 $
```

```
- DØ '''' SKIP ' $
  IF e41  = e2Ø SKIP 2 $
  IF e41  = 23Ø SKIP 4 $
      FetcH  e2Ø eF1 $
  cF1 $
      11111   3333333 $
  SKIP ' 50 $
  $

- SAVE ' $
  IF eIØ = AC SKIP 1 $
      STORE eIØ eF1 $

  AC EQU e1Ø $
  $

-  ' = ' ± ' $
  DØ  ADD e2Ø  e3Ø  AC SKIP 1 $
   ADD  e2Ø  eF1 $
  SAVE e1Ø $
  $

- ' = ' * ' $
  DO MULT e2Ø  e3Ø  AC SKIP 1 $
     MULT e2Ø  eF1 $
  SAVE e1Ø $
  $

- ' = ' / ' $
  DO DIV e2Ø  e3Ø AC SKIP 2 $
     FETCH  e2Ø  eF1 $
     DIV  e3Ø  cF1 $
  SAVE e1Ø $
  $ $
```

```
' = ' - ' $
DO SUB e20 e30 ac skip 2 ec Accu
NEGATE eF1 $
ADD e20. eF1 $
SAVE e1.0
$
```

```
EPM
 A = B + C $
 DOG = EAST + A $
 DOG = DOG / EAST $
 AC = B - AC $
 VAR   A, 1 $
 VAR   B , 1 $
 VAR   C, 1 $
 VAR   DOG, 1 $
 VAR   EAST, 1 $
END  $
```

2) le texte nécessaire pour le système étendu est , par exemple :

```
    $' $eOLI(+ - * / ) , < > [ ]
    Phrases
   <VAR>: = A, <AC> , B, C, EAST, DOG $
   <oper>: = ADD, SUB, MULT, DIV $
   <AC> : = AC $
   <N > : = 1, 2, 4 $
    $$

    Format class [S - INST] "instruction source" * N
   <VAR> = [EXP] $
    e 2 0 $
    SAVE  e1∅  e C   [ACCU] $
    VAR  <VAR> ,< N > $
    e F 1 $
    1 ________1   Réserve  2 ______2 $
    $
    END  $
               END  eF1 $
    eF0 $
    $ $
```

```
Format class [EXP]            "traitement d'une expression"
<VAR> + <VAR> $
DØ  ADD  e20  e30  AC  SKIP 1  e C  [ACCU] $
    ADD  e20  eF1 $
 $
 <VAR>  -  <VAR> $
DO  SUB  e20  e30  AC  SKIP 2  e C  [ACCU] $
    NEGATE  eF1 $
    ADD  e20  eF1 $
 $
<VAR> * <VAR> $
 DØ MULT  e20  e30  AC  SKIP 1  e C [ACCU] $
    MULT  e20  eF1 $
 $
<VAR> / <VAR> $
 DØ DIV  e 20  e 30  AC  SKIP 2  e C  [ACCU] $
    DIV  e 20  eF1 $
$$

Format class [ACCU] "Gestion de l'accumulateur"
DØ  <ØPER>  <VAR><VAR> <AC> [FN  Syst] $
IF  e41  =  e20  SKIP 2 eC  [FN Syst] $
IF  e41  =  e30  SKIP 4 e C [FN  Syst] $
          FETCH  e20  eF1 $
e F1 $
          1 ———————— 1 .  3 ———————— 3
e50 $
$
SAVE  <VAR> $
IF  e10  = AC  SKIP 1  e C  [FN Syst] $
          STORE e10  eF1 $
```

```
AC  EQU  e10  e C  [FN  Syst] $
 $ $
Format class  [FN  Syst] $   "Fonction système"
SKIP  <N> $
eF4 $
$
IF <VAR> = <VAR>  SKIP  <N> $
e F 50  $
$
<AC> EQU  <VAR> $
eF3 $
$
IF <VAR> NE <VAR>  SKIP  <N> $
eF51 $
$ $
EPM
A = B + C  e C [S - inst] $
DOG = EAST + AeC  [S - inst] $
DOG = DOG / EAST e C  [S - instr] $
AC  = B - AC  e C  [S - instr ] $
VAR  A, 1 e  e C  [S - instr] $
VAR  B, 1 e C  [S - instr] $
VAR  C, 1  e C  [S - instr] $
VAR  DOG, 1  e C  [S - inst] $
VAR EAST, 1  e C  [S - inst] $
END  e C  [S - instr] $
```

Annexe 2 - Extraits

a) - Algorithme de template matching en Stage 2

b) - Fonctions systèmes et conversions de paramètres

c) - Code utilisé par BCPL : USASCII à 7 moments ou Isocode

a) Algorithme de Template Matching en Stage 2

" Une ligne source est testée à l'aide de l'arbre des templates selon les règles suivantes :

Règle 1. Initialisation

La règle 2 est appliquée à la racine de l'arbre et au premier caractère de la ligne source

Règle 2. Gestion des strings dans les templates

- si l'une des branches quittant le noeud courant (CC) est identique au caractère courant (NC), alors cette branche répond au caractère courant dans la ligne source.
- sinon, la règle 3 est appliquée au noeud courant et au caractère courant.
- si la branche, répondant au caractère courant, est un S.EOL, la ligne source est reconnue.
- sinon, la règle 2 est réappliquée au noeud atteind, depuis le noeud courant, par cette branche et au caractère suivant dans la ligne source.

Règle 3. Hypothèse de longueur "O" pour un paramètre actuel

- si l'une des branches quittant le noeud courant est un S.PARAM, alors la règle 2 est appliquée au noeud atteint par cette branche et au caractère courant.
  Un string vide est supposé comme paramètre actuel.
- Sinon la règle 4 est appliquée au noeud courant

Règle 4 : Retour dans l'arbre

- Si le noeud courant est la racine, les tests se terminent sur un insuccès total
- Sinon, si la branche conduisant au noeud courant, n'est pas un S.PARAM, alors la règle 3 est appliquée au noeud précédant et au caractère source qui répondait à cette branche.

Règle 5 : Modification des hypothèses de longueur

- Le substring associé à la branche entrante au noeud courant, est prolongé par l'addition du plus court substring balancé (**) dans 'IL qui commence au caractère courant.

  La règle 2 est alors appliquée au noeud courant et au caractère source suivant le nouveau substring

- Si un tel substring ne peut être trouvé, la règle 4 est appliquée au noeud précédent et au premier caractère accepté par le S. PARAM ou, bien, si le S. PARAM a reconnu un stringuide ,au caractère courant.

Si la recherche échoue, l'IL est écrite sur le support extérieur. Sinon, le substring considéré pour le S.PARAM le plus à gauche devient le paramètre actuel $_1$, le substring considéré pour le S.PARAM suivant devient le paramètre$_2$, etc...

(Un maximum de neuf S.PARAM est permis par template).

Quand les paramètres sont tous numérotés, l'interprétation commence."

(**) C'est-à-dire un caractère ou bien un string débutant par une parenthèse gauche et se terminant par une parenthèse droite. Le nombre de parenthèses gauches étant égal au nombre de parenthèses droites.

b) - Fonctions systèmes et conversion de paramètres

Format	Action
eF0	Termine la génération
eF1	- Si elle est placée en début de ligne : - le reste de la ligne est ignoré. - la ligne suivante est interprétée comme une tabulation contrôlant un format d'édition. - Sinon la CL est recopiée sur le support externe sans l'application du TM
eF2	Copie le texte source directement sur le support externe jusqu'à la détection du paramètre $_1$ comme début d'une ligne. Cette dernière étant effacée.
eF3	Le paramètre actuel 2 est assigné comme valeur string au paramètre $_1$
eF4	évalue le paramètre 1 comme une expression pour déterminer le nombre de lignes à sauter dans la macrodéfinition
eF5K	évalue le paramètre $_3$ et effectue le saut si : - le paramètre 1 est identique au paramètre (K = 0) ou s'ils sont distinct (K = 1)
eF6K	de même, mais le saut est exécuté si : - paramètre $_1$ $<$ paramètre 2 (K = -) - paramètre 1 = paramètre 2 (K = 0) - paramètre 1 $\neq$ paramètre 2 (K = 1) - paramètre 1 $>$ paramètre 2 (K = +)

eF7	initialise un compteur d'itération à la valeur de la CL évaluée comme une expression ("DØ " Fortran)
eF8	incrémente d'une unité le compteur d'itération ("Continue " Fortran)
eF9	point de sortie dans un macrocorps.

Tableau 1

Fonctions Système

Format	Action
ed0	Copie le d^{e} paramètre dans la CL
ed1 } ed2 }	Copie la valeur string associée au d^{e} paramètre dans la CL { string vide à défaut d'existence, { symbôle généré copié dans CL et assigné au paramètre à défaut d'existence d'une valeur.
ed3	Copie, dans CL, le caractère suivant immédiatement le paramètre d dans le template (string vide par défaut)
ed4	Copie la valeur d'un paramètre évalué comme une expression arithmétique

ed5	Copie la longueur du paramètre
ed6	Assigne, au paramètre d, la CL comme valeur associée.
ed7	Initialise une itération lexicographique sur la CL en s'arrêtant sur les caractères indiqués après "ed7" (Le paramètre d est utilisé comme mémoire temporaire)
ed8	Copie, dans CL, la représentation décimale du premier caractère formant le paramètre d.

Tableau 2

Conversions de paramètre

c) Code utilisé par BCPL : ISO code (USASCII à 7 moments)

bits 765 \ bits 4321	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
000	0 Null	1	2	3	4 EOT	5	6	7 BELL	8 BS	9 TAB	10 LF	11	12 FF	13 CR	14	15
001	16	17 X-ON	18	19	20	21	22	23	24	25	26	27 ESC	28	29	30	31
010	32 SP	33 !	34 "	35 #	36 $	37 %	38 &	39 '	40 (	41)	42 *	43 +	44 ,	45 -	46 .	47 /
011	48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7	56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?
100	64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G	72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
101	80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W	88 X	89 Y	90 Z	91 [	92 \	93]	94 ↑	95 ←
110	96	97 a	98 b	99 c	100 d	101 e	102 f	103 g	104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
111	112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w	120 x	121 y	122 z	123	124	125	126	127 DEL

Représentation décimale: n

BIBLIOGRAPHIE

Mc ILROY, M.D. : "Macroinstruction extensions of compiler languages", Comm. ACM, Vol. 3 April 1960

CAMPBELL-KELLY, W : "An introduction to macros", Mac Donald/American Elsevier Computer Monographs, 21, 1973

GRIES, D. : "Compiler construction for digital computers", John Wiley and sons, 1971, pp. 412 - 434 et pp 435 - 446.

POOLE, P.C. and Waite, W.M. : "The Stage 2 macroprocessor user reference manuel", U.K. Atomic Energy Authority, Culham Laboratory, Abingdan Berkshire, 1970

LEAVENWORTH, B.M. : "Syntaxmacros and extended translation", Comm. ACM, Vol 9, November 1966, pp. 790 - 793.

FELDMAN and GRIES, D.: "Translator Writing systems", Comm, ACM, Vol 11, February 1968, pp 77

BROOKER, R.A. et MORRIS, D :

"An assembly program For phrase structure languages", Comp "Journal, Vol. 3, 1960.

"Some proposals for the realization of a certain assembly program", Comp. Journal, Vol. 3, 1961.

"A general translation program for phrase structure languages", Journal of the ACM, Vol 9, n° 1, 1962

BROOKER, R.A. and al.: "Trees and Routines", Comp. Journal, Vol. 5, 1962

"The Compiler - Compiler", Pergamon Press, Oxford, 1963

ROSEN, S. : "A compiler building system developed by Brooker and Morris," Com. ACM, Vol 7, n° 7, July 1964 pp. 403-414.

VANDER MEY, J.E., VARNEY, R.C. et PATCHEN, R.E. :

"Symple - A general syntax directed macro preprocessor".
Fall Joint Computer Conference 1969

ESSAI PROSPECTIF D'AMELIORATION D'UN MACROGENERATEUR A NOMS DISTIRBUES

ERRATA

p. 1.4 ligne 26 : lire "qui pourraient" au lieu de "qui pouvaient"
ligne 28 : lire "le point(a), cité ci-dessus," au lieu de "le point (a),cité cité ci-dessus,"

p. 2.1.2 ligne 14 : ajouter entre "25- ..." et "Le but de ...", la phrase: " ' " indique la présence d'un paramètre formel et "'" la fin d'une ligne source.
ligne 24 : lire "Mais un prefixe peut-être commun" au lieu de "Mais un prefixe commun peut-être commun".

p. 2.1.3 ligne 10 : Après, "actuel" lire "ABCDE" au lieu de "INABCDEFGH"

p. 2.1.5 ligne 8 :lire "existence; ou en terme" au lieu de "existence; en terme".

p. 2.2.1 ligne 18 :lire "de façon à ceque le macrogénérateur en tire" au lieu de "de façon qu'il en tire profit".
ligne 27 : reporter les points 3. et 4. à la fin du paragraphe 2.2.

p. 2.3.4 ligne 3 : lire " la position des paramètres formel" au lieu de "la position des identifications"

p. 2.3.8 ligne 1 : à remplacer par : "Inutile de préciser que tout nom de classe référencé doit finalement être déclaré et...".
ligne 11 : lire "Un corps de macro définition " au lieu de "Une définition sémantique".
ligne 24 : lire "des corps de macrodéfinitions" au lieu de "des définition sémantiques"

p. 2.3.13 ligne 13 : lire "soient nécessaires" au lieu de "soit nécessaire".

p. 2.4.21 ligne 15 : lire "ou le macrogénérateur étendu" au lieu de "ou les facilités de classes".

p. 2.4.31 ligne 1 :lire "lors de la définition" au lieu de "lors de définiti

p. 2.4.33 ligne 10 : lire "la même fonction" au lieu de "le fonction même".

p. 2.4.34 ligne 2 : lire "un indicateur est" au lieu de ", l'indicateur bloc ne réponde à la demande, l'indicateur est".

p. 2.4.37 dernière ligne : lire " pour la recherche au sein d'un arbre de templates (notament pour les macroréférences passées comme paramètre)" au lieu de "pour les macroréférences passées comme paramètre".

p. 2.4.39 avant dernière phrase : ajouter à la suite de "passée comme paramètre","que s'il y a méprise sur l'un de ces propres paramètres".

p. 2.4.42 ligne 15 : lire "-chaine des paramètres de la macroréférence" au lieu de "-chaine des macroréférences".
ligne 18 : lire "implicite au TM" au lieu de "implicite au template".

ann.1 p.b2 : lire l'entête de macro "'='+'$" au lieu de "'='±'$"
ajouter , à sa suite, la macrodéfinition :

```
" ' ='-'$
  DO sub e20 e30 ac skip 2 eC ACCU$
  NEGATE eF1$
  ADD    e20 eF1$
  SAVE e10
  $   "
```