



HAL
open science

Notation et processus de traduction des langages symboliques

Louis Bolliet

► **To cite this version:**

Louis Bolliet. Notation et processus de traduction des langages symboliques. Autre [cs.OH]. Université Joseph-Fourier - Grenoble I, 1967. Français. NNT: . tel-00008411

HAL Id: tel-00008411

<https://theses.hal.science/tel-00008411>

Submitted on 8 Feb 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre :

THESES

présentées à

LA FACULTE DES SCIENCES DE GRENOBLE

pour obtenir

LE GRADE DE DOCTEUR ES SCIENCES APPLIQUEES

par

L. Bolliet

Ingénieur I. E. G. . I. R. G. Licencié ès Sciences

Première thèse :

NOTATION ET PROCESSUS DE TRADUCTION DES LANGAGES SYMBOLIQUES

Deuxième thèse :

PROPOSITIONS DONNEES PAR LA FACULTE

Thèses soutenues le Juin 1967, devant la commission d'examen :

MM. J. KUNTZMANN,	Président
B. VAUQUOIS	
N. GASTINEL	Examineurs.
J. ARSAC	

PROFESSEURS TITULAIRES (suite)

MM.	KLEIN J.	Mathématiques
	VAILLANT F.	Zoologie et Hydrobiologie
	ARNAUD Paul	Chaire de Chimie
	SENGEL P.	Chaire de Zoologie
	BARNOUD F.	Chaire de Biozythèse de la Cellulose
	BRISSONNEAU P.	Physique
	GAGNAIRE	Chaire de Chimie Physique
Mme	KÖFLER L.	Botanique
	DEGRANGE Charles	Zoologie
	PEBAY-PEROULA J.C.	Physique
	RASSAT A.	Chaire de Chimie Systématique
	DUCROS P.	Vhaire de Cristallographie Physique
	DODU Jacques	Chaire de Mécanique Appliquée I.U.T.
	ANGLES D'AURIAC P.	Mécanique des Fluides
	LACAZE A.	Thermodynamique

PROFESSEURS SANS CHAIRE

MM.	GIDON P.	Géologie et Mnéralogie
	GIRAUD P.	Géologie
	PERRET R.	Servomécanisme
Mme	BARBIER M.J.	Electrochimie
Mme	SOUTIF J.	Physique
	COHEN J.	Electrotechnique
	DEPASSEL R.	Mécanique des Fluides
	GASTINEL N.	Mathématiques Appliquées
	GLENAT R.	Chimie
	BARRA J.R.	Mathématiques Appliquées
	COUMES A.	Electronique
	PERRIAUX J.	Géologie et Minéralogie
	ROBERT A.	Chimie Papetière
	BIAREZ J.P.	Mécanique Physique
	BONNET G.	Electronique
	CAUQUIS G.	Chimie Générale
	BONNETAIN L.	Chimie Minérale
	DEPOMMIER P.	Etude Nucléaire et Génie Atomique
	HACQUES Gérard	Calcul Numérique
	POLOUJADOFF M.	Electrotechnique

PROFESSEURS ASSOCIES

MM.	NAPP-ZINN	Botanique
	RÓDRIGUES Alexandre	Mathématiques Pures
	STANDING Kenneth	Physique Nucléaire

MAITRES DE CONFERENCES

MM.	LANCIA Roland	Physique Atomique
Mme	KAHANE J.	Physique
	DEPORTES C.	Chimie
Mme	BOUCHE L.	Mathématiques
	SARROT-REYNAUD	Géologie Propédeutique

MAITRES DE CONFERENCES (suite)

Mme	BONNIER M.J.	Chimie
MM.	KAHANE A.	Physique Générale
	DOLIQUE J.M.	Electronique
	BRIERE G.	Physique M.P.C.
	DESRE G.	Chimie S.P.C.N.
	LAJZROWICZ J.	Physique M.P.C.
	VALENTIN P.	Physique M.P.C.
	BERTRANDIAS J.P.	Mathématiques Appliquées T.M.P.
	LAURENT P.J.	Mathématiques Appliquées T.M.P.
	CAUBET J.P.	Mathématiques Pures
	PAVAN J.J.	Mathématiques
Mme	BERTRANDIAS F.	Mathématiques Pures M.P.C.
	LONGEGUEUE J.P.	Physique
	NIVAT M.	Mathématiques Appliquées
	SOHM J.C.	Electrochimie
	ZADWORNY F.	Electronique
	DURAND F.	Chimie Physique
	CARLER G.	Biologie Végétale
	AUBERT G.	Physique M.P.C.
	DELPUECH J.J.	Chimie Organique
	PFISTER J.C.	Physique C.P.E.M.
	CHIBON P.	Biologie Animale
	IDELMAN S.	Physiologie Animale
	BOUVARD Maurice	Hydrologie
	RICHARD Lucien	Botanique
	PELMONT Jean	Physiologie Animale
	BLOCH D.	Electrotechnique I.P.
	BOUSSARD J. Claude	Mathématiques Appliquées I.P.
	MOREAU René	Hydraulique I.P.
	BRUGEL L.	Energétique I.U.T.
	SIBILLE R.	Construction Mécanique I.U.T.
	ARMAND Yves	Chimie I.U.T.
	BOLLIET Louis	Informatique I.U.T.
	KUHN Gérard	Energétique I.U.T.
	GERMAIN J.P.	Construction Mécanique I.U.T.
	CONTE René	Thermodynamique
	JOLY Jean René	Mathématiques Pures
Mme	PIERY Yvette	Biologie Animale
	BENZAKEN Claude	Mathématiques Appliquées

MAITRE DE CONFERENCES ASSOCIES

MM.	SAWCZUK A.	Mécanique des Fluides
	CHEEKE J.	Thermodynamique
	YAMADA O.	Physique du Solide
	NATR Lubomir	B.M.P.V.
	NAVLOP Arch	Physique Industrielle
	SILBER Léo	Radioélectricité
	NOZAKI Akihiro	Mathématiques Appliquées
	RUTLEDGE Joseph	Mathématiques Appliquées
	DONOHU Paul	Physique Générale
	EGGER Kurt	B.M.P.V.

Je tiens à exprimer ma très grande reconnaissance :

A Monsieur le Professeur J. KUNTZMANN

Directeur de l'Institut de Mathématiques Appliquées de GRENOBLE qui par sa bienveillante insistance a rendu ce travail possible et dont la confiance et les conseils m'ont été indispensables pour le mener à bien.

A Monsieur le Professeur B. VAUQUOIS

Directeur du Centre d'Etudes pour la Traduction Automatique qui s'est intéressé à mon travail et m'a prodigué ses conseils,

A Monsieur le Professeur N. GASTINEL

Directeur de l'Institut de Programmation de GRENOBLE, qui m'a fait largement profiter de son expérience dans le domaine de la programmation au cours de nombreuses conversations et discussions.

A Monsieur le Professeur J. ARSAC

Directeur de l'Institut de Programmation de PARIS, dont j'ai eu souvent l'occasion d'apprécier la compétence dans ce domaine et qui m'a fait le grand honneur de participer au jury.

Ce travail a été réalisé dans le cadre d'une active collaboration avec les constructeurs de machines et la Délégation Générale à la Recherche Scientifique et Technique sous forme de contrats d'études et de recherches et m'a donné l'occasion de nouer de nombreuses relations personnelles avec des collègues de l'industrie que je tiens à remercier ici pour leur réel esprit de coopération.

Je remercie également tous mes collègues du Laboratoire qui m'ont aidé dans la réalisation matérielle de ce travail.

TABLE DES MATIERES

1^{ère} Partie

NOTATIONS DE BASE

	<u>Pages</u>
<u>CHAPITRE I - Notations algorithmiques</u>	
<u>I - Aspects contradictoires de l'écriture usuelle et de l'écriture machine</u>	
A) Ecriture usuelle	A-I-1
Opérations associatives	A-I-3
Opérations commutatives	A-I-3
Opérations commutatives et associatives	A-I-3
Opérations distributives	A-I-4
Priorité des opérateurs	A-I-4
Ambiguïté de l'ordre d'évaluation	A-I-5
B) Ecriture machine	
Forme unilinéaire de l'écriture	A-I-5
Notation fonctionnelle des indices	A-I-6
Opération de division	A-I-7
Désignation des quantités	A-I-7
Notation fonctionnelle des opérations spéciales ..	A-I-7
Associativité - Commutativité - Priorité des opérations	A-I-7
Opérations associatives et non équivalence dynamique	A-I-7
Opérations commutatives et effet de bord	A-I-8
Opérations commutatives et associatives. Influence des arrondis	A-I-9
Priorité des opérations	A-I-10
Cas des opérateurs unaires	A-I-10
Ordre d'évaluation des opérations de même priorité	A-I-11
Utilisation du contexte	A-I-12

	<u>Pages</u>
II - <u>Propriétés dynamiques des notations algorithmiques</u>	A-I-12
- Substitution dynamique	A-I-12
- Structure de blocs	A-I-14
- Exécution séquentielle	A-I-15
- Exécution parallèle	A-I-16
CHAPITRE II - <u>Notations syntaxiques</u>	A-II-1
I - <u>Formalisation de la description syntaxique</u>	A-II-3
Chaînes	A-II-4
Langage	A-II-4
Dualité génération - reconnaissance	A-II-5
Classification des méthodes de description syntaxique ..	A-II-5
Méthodes dérivées de la logique formelle	A-II-5
Méthodes algébriques	A-II-8
Métalangage et forme normale de Backus	A-II-12
Utilisation de la forme normale de Backus pour la des-	
cription des notations utilisées dans les langages de	
programmation	A-II-13
- La notation complètement parenthésée	A-II-13
- Ordre lexicographique. Ordre d'évaluation. Prio-	
rité des opérateurs	A-II-15
Ordre lexicographique et ordre dynamique	A-II-15
Ordre lexicographique direct et inverse	A-II-15
- Classification des notations utilisées pour les	
expressions	A-II-16
II - <u>Propriétés des notations syntaxiques</u>	
- Suppression et addition de symboles non terminaux	A-II-22
- Autre aspect de la dualité génération - reconnaissance :	
le problème de l'analyse syntaxique	A-II-22
- Récursivité à droite ou à gauche. Priorité de gauche à	
droite. Ambiguïté	A-II-24
- Priorité des opérateurs	A-II-34
- Hiérarchie des règles de priorité et forme normale de	
Backus	A-II-35
- Interférence de la syntaxe et de la sémantique	A-II-36
- Insuffisances du formalisme utilisé dans la descrip-	
tion syntaxique	A-II-39

III - Autres méthodes de description syntaxique

- Cartes syntaxiques A-II-44
- La notation bidimensionnelle de Cobol A-II-46
- La notation syntaxique de PL/I A-II-49
- Autres extensions de la forme normale de Backus A-II-53

Conclusion générale A-II-56

CHAPITRE III - Notations sémantiques

- I - La notation utilisée dans le rapport Algol A-III-1
- II - Description sémantique et mécanisme d'évaluation A-III-5
- III - Etude de quelques mécanismes de description sémantique ... A-III-6
 - La notation de Church A-III-6
 - Le concept de fonction A-III-6
 - Fonction de plusieurs variables A-III-7
 - Abstractions fonctionnelles A-III-8
 - Règles d'écriture A-III-9
 - Intérêt de la notation de Church A-III-11
 - Valeurs d'une fonction A-III-12
 - Le langage LISP et la notation de Church A-III-12
 - Description syntaxique et sémantique du langage Micro-
Algol A-III-13
 - Syntaxe analytique A-III-13
 - Définition du langage Micro-Algol A-III-15
 - Mécanisme d'évaluation des programmes et λ -expressions
selon Landin A-III-19
 - La structure opérateur - opérande A-III-19
 - Les diverses formes d'expression A-III-21
 - Système interprétatif pour la description sémantique ... A-III-27

Conclusion générale A-III-39

CHAPITRE IV - Notations parenthésées

- I - Propriétés des notations parenthésées A-IV-1
 - Cas où il existe une seule paire de parenthèses A-IV-3
 - Poids d'un élément A-IV-3
 - Fonction de distribution d'une suite d'éléments . A-IV-3
 - Couplage des parenthèses A-IV-4
 - Exemples de couplages A-IV-5
 - Couplage correct des parenthèses A-IV-5
 - Emboîtement - Recherche des paires les plus in-
térieures A-IV-6

	<u>Pages</u>
Cas où il existe des parenthèses de diverses espèces ...	A-IV-8
Cas où il existe une hiérarchie des couplages	A-IV-9
Délimiteurs ayant une double fonction de délimitation ..	A-IV-9
II - <u>Analyse des niveaux de parenthèses</u>	A-IV-12
Sous-expression	A-IV-12
Degré d'imbrication d'une sous-expression	A-IV-12
Hauteur d'une sous-expression	A-IV-13
Relation entre la hauteur et le couplage des parenthèses	A-IV-13
Rang d'une sous-expression	A-IV-14

2^{ème} Partie

PROCESSUS DE SUBSTITUTION

	<u>Pages</u>
<u>CHAPITRE I - Définitions et résultats élémentaires</u>	
1 - Introduction	B-I-1
2 - <u>Substitutions statiques et substitutions dynamiques</u>	B-I-4
2.1. Exemples de substitutions statiques	B-I-4
Assemblage	B-I-4
Chargement	B-I-5
Compilation	B-I-6
2.2. Exemples de substitutions dynamiques	B-I-7
Interprétation	B-I-7
Appel par valeur	B-I-8
Appel par nom	B-I-8
Macrogénération	B-I-9
Allocation dynamique	B-I-12
3 - <u>Systèmes interpréteurs et systèmes compilateurs</u>	B-I-13
3.1. Expressions complètement parenthésées	B-I-16
Propriétés élémentaires	B-I-16
Technique des piles de mémoire	B-I-17
Evaluation directe	B-I-18
Méthode itérative	B-I-18
Méthode récursive	B-I-21
Compilation	B-I-23
3.2. Expressions sans parenthèses. Notations préfixée et postfixée	B-I-23
La notation préfixée	B-I-23
La notation postfixée	B-I-26
Propriété caractéristique	B-I-27
Evaluation d'une expression postfixée	B-I-29
Transformation d'une expression complètement parenthésée en expression postfixée	B-I-30

	<u>Pages</u>
3.3. Expressions non complètement parenthésées	B-I-32
Portée des opérateurs	B-I-32
Transformation des expressions non complètement pa- renthésées	B-I-33
Passage à la forme complètement parenthésée ..	B-I-33
Passage à la forme postfixée	B-I-37
Evaluation directe	B-I-42
Compilation	B-I-43
3.4. Fonctions. Variables indicées. Expressions condi- tionnelles	B-I-47
3.5. Instructions composées	B-I-50
3.6. Structure de blocs	B-I-51
3.7. Structure de sous-programmes	B-I-51

CHAPITRE II - Macroévaluation

1 - Structure du processus de compilation	B-II-1
2 - Macroévaluation	B-II-3
Macrosubstitution de PL/I	B-II-5
Macro-algol	B-II-8
Redéfinition des opérateurs	B-II-10
Macro extensions syntaxiques	B-II-12

CHAPITRE III - Evaluation syntaxique

1 - Analyse lexicographique	B-III-3
Règles liées de la description syntaxique	B-III-3
Analyse lexicographique et édition	B-III-4
Utilisation de tables d'états pour l'analyse lexico- graphique	B-III-5
2 - Algorithmes particuliers	B-III-12
Méthode des états syntaxiques	B-III-12
Méthode des fonctions syntaxiques	B-III-18
Vérificateurs syntaxiques	B-III-21
3 - Algorithmes généraux	B-III-28
Analyse unique et analyse multiple	B-III-28
Analyse ascendante et descendante	B-III-29
Analyse descendante	B-III-31
Analyse ascendante	B-III-38
Sélectivité	B-III-41

CHAPITRE IV - Evaluation sémantique

1 - Définition et évaluation des programmes objet	B-IV-1
1.1. Définition des programmes objet	B-IV-1
Expressions. Affectations. Langages intermédiaires issus de la notation postfixée	B-IV-3
Diverses formes de langages intermédiaires	B-IV-4
Affectation	B-IV-5
Profil qualitatif	B-IV-6
Variables indicées	B-IV-7
1.2. Adressage dynamique	B-IV-8
Paramètres	B-IV-10
Tête de procédure	B-IV-10
2 - Evaluation des programmes objet. Interpréteur Algol	B-IV-11
2.1. Pile d'évaluation	B-IV-11
2.2. Accumulateurs	B-IV-12
2.3. Langage objet	B-IV-14
2.4. Exemples de séquences de programmes objet	B-IV-18
1) Expressions et instructions d'affectation ...	B-IV-18
2) Tableaux	B-IV-19
3) Activation des blocs et des procédures	B-IV-20
4) Correspondance paramètres effectifs. Paramètres formels. Appel par nom	B-IV-20
5) Etiquette. Expression de désignation. Instruction aller a	B-IV-24
6) Aiguillage	B-IV-24
7) Instruction pour	B-IV-27
Interpréteur Algol	B-IV-30
Génération des programmes objet	B-IV-40
1 - Méthode de la double priorité	B-IV-40
2 - Méthode des procédures syntaxiques	B-IV-48
3 - Méthode des précédences avec vérification syntaxique complète	B-IV-55
4 - Utilisation des macroinstruction dans la génération	B-IV-60
Compilateurs modulaires et paramétriques	B-IV-68

3^e Partie

COMPILATEURS CONVERSATIONNELS ET INCREMENTIELS

	<u>Pages</u>
<u>CHAPITRE I - Compilateurs pour systèmes à accès multiple</u>	C-I-1
Compilateurs non conversationnels	C-I-1
Compilateurs conversationnels	C-I-3
Analyseurs syntaxiques	C-I-3
Interpréteurs	C-I-4
Compilateurs incrémentiels ou différentiels	C-I-4
Compilateurs symétriques	C-I-5
Interpréteurs incrémentiels	C-I-5
Un compilateur Fortran incrémentiel et symétrique	C-I-5
Langage source	C-I-5
Langage objet	C-I-6
Représentation des programmes objet	C-I-7
Décomposition des expressions	C-I-8
Exécution des programmes objet	C-I-9
Recomposition des expressions et reconstruction des programmes source	C-I-10
Recomposition des expressions postfixées	C-I-10
Recomposition des expressions à partir d'un langage intermédiaire à 2 adresses	C-I-13
 <u>CHAPITRE II - Compilateurs et interpréteurs incrémentiels</u>	
La compilation incrémentielle	C-II-1
L'évaluation incrémentielle	C-II-2
Un système incrémentiel pour la construction et l'évaluation récursives des programmes Algol	C-II-3
Langage source	C-II-3
Interconnection des instructions	C-II-4
Représentation des éléments syntaxiques associés à un programme	C-II-10
Table des références symboliques	C-II-11
Modification des programmes	C-II-12
Exécution des programmes	C-II-13
Un système incrémentiel pour la construction et l'évaluation itérative des programmes Algol	C-II-14

	<u>Pages</u>
Programmation conversationnelle	C-II-14
Compilation incrémentielle itérative	C-II-15
Evaluation incrémentielle	C-II-16
Langage de programmation et langage de commande	C-II-16
Modes d'utilisation du système	C-II-17
Mode système et commandes de manipulation de files ...	C-II-19
Mode compilateur et commandes de compilation	C-II-22
Mode moniteur	C-II-23
Mode conversationnel	C-II-24
Numérotation syntaxique et compilation in-	
crémentielle des instructions	C-II-25
Mécanisme d'inclusion des blocs	C-II-28
Transformation des programmes mode conver-	
sationnel mode moniteur	C-II-29
Mode Interpréteur et commandes d'interprétation .	C-II-30
Lecture et affectation dynamique	C-II-32
Contrôle dynamique et l'interprétation	C-II-33
Mode calculateur de bureau	C-II-34

Conclusion générale

C H A P I T R E I

NOTATIONS ALGORITHMIQUESI - ASPECTS CONTRADICTOIRES DE L'ECRITURE USUELLE ET DE L'ECRITURE MACHINE.

A) L'écriture usuelle qui paraît très normalisée au profane résulte en réalité d'apports successifs souvent indépendants.

On y trouve en particulier :

- des variations suivant les auteurs : cas de la notation matricielle par exemple.
- des redondances : utilisation des radicaux et des exposants fractionnaires
- des ambiguïtés : par exemple, e^d peut signifier :

$\exp(d)$, e désigne alors la fonction exponentielle,

e^d représentant une quantité élevée à la puissance d

$f(x+y)$ peut signifier :

- la valeur de la fonction f pour la valeur $x+y$ de l'argument
- le produit des quantités désignées par f et $x+y$

Si la priorité des opérations n'est pas strictement définie $A + B \times C$ peut signifier

$$(A + B) \times C$$

ou $A + (B \times C)$

L'écriture usuelle répond à un besoin de saisie globale ce qui amène :

- à rompre la monotonie au moyen d'une notation multilinéaire :

barres de fraction, écriture des exposants au dessus de la ligne et des indices au dessous de la ligne, désignations multiples permises par une même lettre en utilisant des signes spéciaux (A' , A'' , \bar{A} , \hat{A} , etc.)

Cette forme non unilinéaire de l'écriture est mise en évidence dans l'exemple suivant :

$$A' + \frac{B-C}{D+E} + F^3 - K_{i,j} \times \int_0^1 f(x) dx + \sqrt{x}$$

- à faire un certain nombre d'omissions ou d'abréviations :

omission du signe de multiplication et d'exponentiation.

L'omission du signe de multiplication nécessite la désignation des quantités par une seule lettre.

Exemples : BC a même signification que $B \times C$

$A(B+C)$ " " " " $A \times (B + C)$
 $(A+B) (C+D)$ " " " " $(A+B) \times (C+D)$

L'omission du signe d'exponentiation nécessite l'écriture exponentiée.

Remarquons que l'omission du signe de multiplication peut amener des confusions lorsqu'on veut désigner la longueur d'un segment : en géométrie, la longueur du segment AB est désignée par \overline{AB} pour éviter la confusion avec AB signifiant $A \times B$.

En raison de son importance pour la suite nous allons examiner en détail les règles de suppression des parenthèses :

Les parenthèses servent à combiner les opérations dans un certain ordre : le contenu d'une paire de parenthèses étant équivalent à un simple opérande, les opérations contenues à l'intérieur des parenthèses doivent être effectuées les premières.

Exemple :

$$\underbrace{(A+B)}_1 \times \underbrace{(C-(D \times E))}_2$$

3

1°) Opérations associatives

Une opération θ est dite associative si étant donné A,B,C alors $(A \theta B) \theta C$ a le même sens que $A \theta (B \theta C)$.

Etant donné un nombre fini d'opérandes

A,B,....., Z

alors l'expression :

$$A \theta (B \theta (C \dots \theta Z))$$

avec n'importe quelle disposition de parenthèses a toujours le même sens qui est par définition celui de l'expression sans parenthèses :

$$A \theta B \theta C \dots \theta Z$$

2°) Opérations commutatives

Une opération θ est dite commutative si étant donné A et B

$$A \theta B$$

a le même sens que :

$$B \theta A$$

3°) Opérations commutatives et associatives

Si l'opération θ est commutative et associative alors étant donné un nombre fini d'opérandes, l'expression

$$A \theta B \dots \theta Z$$

a un sens indépendant de l'ordre des lettres et de la disposition des parenthèses.

4°) Opérations distributives

Distribution de la multiplication par rapport à l'addition

Elle se traduit par l'équivalence des deux expressions

$$(A + B) \times C$$

et $A \times C + B \times C$

On remarque que l'utilisation de cette propriété change le nombre d'opérations.

5°) Priorité des opérateurs

Considérons l'expression

$$\alpha \theta_1 \beta \theta_2 \gamma$$

dans laquelle α , β et γ sont des opérandes, θ_1 et θ_2 des opérateurs ; si l'on a :

$$\text{priorité } (\theta_1) \gg \text{priorité } (\theta_2)$$

alors l'expression ci-dessus est équivalente à

$$(\alpha \theta_1 \beta) \theta_2 \gamma$$

sinon elle est équivalente à

$$\alpha \theta_1 (\beta \theta_2 \gamma)$$

L'utilisation des priorités d'opérateurs permet l'omission d'un grand nombre de parenthèses dans l'écriture des expressions.

Les notations usuelles utilisent pour les opérateurs arithmétiques la convention de priorité suivante dans l'ordre croissant :

- opérateurs additifs
- opérateurs multiplicatifs
- opérateurs d'exponentiation
- fonctions

Exemple : L'expression $A + B \times C^D$

a même sens que l'expression

$$\left(A + \left(B \times \underbrace{(C + D)}_1 \right) \right)_2 \underbrace{\hspace{10em}}_3$$

The diagram illustrates the evaluation order of the expression $(A + (B \times (C + D)))$. It uses three levels of brackets to show the sequence of operations from innermost to outermost:

- Level 1: A bracket under the sub-expression $(C + D)$.
- Level 2: A bracket under the sub-expression $(B \times (C + D))$.
- Level 3: A large bracket under the entire expression $(A + (B \times (C + D)))$.

Ambiguïté de l'ordre d'évaluation

L'utilisation simultanée de l'associativité et de la commutativité des opérations n'impose pas d'ordre strict dans l'évaluation des opérations. Cela signifie que les opérations, compte tenu des priorités d'opérateurs, des parenthèses et de la non commutativité de certaines opérations, peuvent s'exécuter aussi bien de gauche à droite que de droite à gauche.

B) L'écriture machine qui permet la description des algorithmes pour les ordinateurs est caractérisée par les propriétés suivantes :

- 1) Ecriture unilinéaire
- 2) Formalisation des règles d'écriture
- 3) Absence d'ambiguïtés (un même texte ne doit jamais avoir deux interprétations différentes).

Forme unilinéaire de l'écriture

Cette forme unilinéaire est imposée par les dispositifs d'entrée des machines. Quel que soit le dispositif utilisé : clavier, carte perforée, bande perforée, une donnée transmise à une calculatrice est toujours une suite de caractères, lue et interprétée caractère par caractère.

En particulier, un programme symbolique est une donnée qui doit respecter ce processus de transmission séquentielle.

Cette forme unilinéaire des programmes a les conséquences suivantes :

a) notation fonctionnelle des indices

L'écriture des indices peut être rendue unilinéaire en utilisant la notation fonctionnelle des indices qui consiste à écrire la liste d'indices d'une variable indicée comme la liste d'arguments d'une fonction.

Exemples :

a_i s'écrit $a(i)$

b_{ij} s'écrit $b(i,j)$

$c_{i,j,m}$ " $c(i(j),m)$

Remarque 1 Il est important de différencier l'écriture des variables indicées de celle des fonctions.

En notation ALGOL, par exemple, on utilise les crochets au lieu des parenthèses pour l'écriture des listes d'indices.

b_{ij} s'écrit $b[i,j]$

Remarque 2 Cette différence d'écriture exprime en fait la différence de nature entre les quantités désignées par une variable indicée et une fonction.

Une variable indicée $A[i,j]$ désigne une mémoire qui contient une certaine valeur correspondant à un élément du tableau A : suivant les cas la variable indicée peut désigner la mémoire ou la valeur qu'elle contient.

Une fonction $F(x,y)$ désigne une valeur, la valeur de la fonction F par les arguments x et y .

Ainsi dans une expression, une variable indicée peut apparaître en partie gauche ou en partie droite d'une instruction d'affectation alors qu'un indicateur de fonction ne peut apparaître qu'en partie droite (sauf dans le cas très particulier de la définition (déclaration) associée à cette fonction).

b) l'opération de division doit être représentée par le signe $/$ au lieu de la barre horizontale de fraction - Cette convention nécessite l'utilisation de parenthèses pour la délimitation du numérateur et du dénominateur.

Exemple : $A + \frac{B-C}{D+E}$ s'écrira $A + (B-C) / (D + E)$

c) désignation des quantités par des identificateurs constitués de mots librement choisis par le programmeur. Cette règle permet d'éviter l'emploi de signes spéciaux associés aux lettres (A' , A , etc) et donne une liberté beaucoup plus grande dans le choix des désignations.

d) notation fonctionnelle des opérations spéciales

Les symboles tels que $\sqrt{\quad}$, \int sont remplacés par des identificateurs auxquels sont associés les arguments correspondants.

\sqrt{x} s'écrira $R A C (x)$

$\int_0^1 f(x,t) dx$ s'écrira $S O M M E (0, 1, f(x, t), x)$

Associativité - Commutativité - Priorité des opérations.

a) opérations associatives et non équivalence dynamique

Les opérations approchées ne sont que très rarement associatives.

Considérons en effet l'expression :

$$\begin{array}{l} A + B + C \\ \text{avec } A = 9.998 \qquad B = 0.007 \qquad C = 0.005 \end{array}$$

En effectuant les opérations avec une précision de 4 chiffres significatifs et sans arrondi on trouvera :

$$\begin{aligned} (A + B) + C &= (9.998 + 0.007) + 0.005 = 10.00 + 0.005 = 10.00 \quad \text{et} \\ A + (B + C) &= 9.998 + (0.007 + 0.005) = 9.998 + 0.012 = 10.01 \end{aligned}$$

Remarque 3 Au point de vue dynamique de la réalisation, les diverses écritures confondues par l'associativité sont très différentes.

Remarque 4 On peut donner un sens dynamique à l'expression :

$$A \theta B \text{ ----- } \theta Z$$

c'est celui de l'expression :

$$(\text{-----}((A \theta B) \theta C)\text{---}) \theta Z$$

b) opérations commutatives et effet de bord

Les opérations usuellement commutatives, comme l'addition et la multiplication peuvent ne plus avoir cette propriété dans un langage de programmation.

Considérons par exemple le cas d'une variable indicée ou d'une fonction en Algol. L'évaluation d'une expression d'indice ou d'un argument peut changer la valeur d'une autre variable figurant dans la même expression (propriété connue sous le nom d'effet de bord).

Ainsi le résultat de l'opération
 $F(E1, E2, E3) \times A$
 n'est pas le même que celui de l'opération
 $A \times F(E1, E2, E3)$
 si l'évaluation des expressions E1, E2, E3 peut changer la valeur
 de la variable A.

c) opérations commutatives et associatives. Influence des arrondis

En général les opérations des langages algorithmiques ne sont ni commutatives, ni associatives.

Dans le langage FORTRAN, on utilise pourtant la propriété des opérations commutatives et associatives (à savoir que l'expression

$$A \theta B \text{ ----- } \theta Z$$

à un sens indépendant de l'ordre des lettres, si θ est un opérateur commutatif et associatif) pour minimiser le nombre de transferts pour le calcul.

En effet considérons l'opération de division comme une opération de multiplication par l'inverse.

$$\text{Alors } A \times B/C \text{ s'écrit } A \times B \times C^{-1}$$

En considérant \times comme un opérateur commutatif et associatif, on voit que les deux expressions

$$A \times B \times C^{-1} \text{ et } A \times C^{-1} \times B$$

ont même valeur.

Dans la compilation des expressions, cette propriété est utilisée pour réaliser une alternance régulière de multiplications et divisions : si le nombre de multiplication est différent de celui des divisions, les opérations excédentaires sont exécutées en dernier lieu.

Exemple :

$A \times B / C / D \times E / F$

sera transformé en

$A \times B / C \times E / D / F$

d) Priorité des opérations

La priorité des opérations est en général identique à celle des notations usuelles à savoir dans l'ordre décroissant.

- fonctions
- opérateur d'exponentiation
- opérateur multiplicatif
- opérateur additif

Cas des opérateurs unaires + et -

Le cas de l'opérateur unaire + correspond à une redondance qu'il est facile d'éliminer soit dans la définition du langage, soit dans les programmes sources préalablement à la compilation.

Pour l'opérateur unaire -, il y a trois priorités possibles :

- priorité de l'opérateur additif
- priorité de l'opérateur multiplicatif
- priorité de l'opérateur d'exponentiation

L'expression $- A \times B$ est donc équivalente à l'expression :

- $(A \times B)$ dans le premier cas
- $(-A) \times B$ dans le second et le troisième

Remarque : Si l'on considère l'opération unaire - comme une multiplication par (-1), il est plus normal de choisir la priorité de l'opérateur multiplicatif.

Si l'opérateur unaire à une priorité égale à celle de l'opérateur d'exponentiation (cas du langage P L/1)

L'expression :

$$A * * - B$$

Est équivalente à :

$$A * * (-B)$$

e) Ordre d'évaluation des opérations de même priorité

En général, l'ordre d'évaluation est de gauche à droite, mais le texte d'un programme source devant être interprété par l'ordinateur qui le traduit, l'ordre d'évaluation peut également être choisi de droite à gauche ou même être différent d'une opération à l'autre.

En Algol et en Fortran et dans la plupart des langages l'ordre d'évaluation est de gauche à droite.

En PL/1 l'ordre d'évaluation est de gauche à droite sauf pour les opérations d'exponentiation (* *) et les opérations unaires d'addition et de soustraction où l'ordre d'évaluation est de droite à gauche.

L'expression a^{-b} s'écrit :

$a \uparrow (-b)$ en ALGOL

$a ** -b$ en P L/1

Dans certains langages (G A T et I T) l'ordre d'évaluation est de droite à gauche.

f) Utilisation du contexte

Dans un langage de programmation, on distingue deux sortes de concepts.

- Les concepts impératifs qui correspondent aux instructions et à l'exécution d'un programme.

- Les concepts descriptifs qui correspondent aux déclarations et donnent les informations relatives aux diverses quantités : nature, type, dimension, précision, définition formelle (cas des procédures).

La partie descriptive constitue un contexte à l'intérieur duquel le programme doit s'exécuter.

On peut remarquer que cette partie descriptive pourrait être incorporée dans la partie impérative si l'on attachait directement et à chacune des occurrences d'identificateur l'information qui lui est associée.

Cette partie descriptive permet donc une économie d'écriture considérable en laissant au compilateur le soin de redistribuer cette information à l'intérieur du programme.

II - PROPRIETES DYNAMIQUES DES NOTATIONS ALGORITHMIQUES.

Les notations algorithmiques utilisent des concepts dynamiques liés à la notion de temps.

1) Substitution dynamique.

Cette notion est la transposition, au niveau d'un langage de programmation, d'une propriété caractéristique des ordinateurs. Les ordinateurs sont dotés de mémoires dont l'adresse reste fixe alors que leur contenu peut changer au cours du temps. On peut se représenter une mémoire élémentaire comme une boîte dont le nom reste le même et dont le contenu varie.

Dans un langage de programmation, la substitution dynamique s'exprime au moyen de l'instruction d'affectation qui permet d'affecter la valeur d'une expression à un nom.

Le signe '=' a deux significations qu'il faut distinguer.

On a choisi de garder le signe '=' pour exprimer la relation d'égalité :

par exemple : $A = B + C$

exprime une relation entre les expressions A et B + C : la valeur de cette relation est une quantité de type booléen qui est :

vrai si valeur (A) = valeur (B + C) et faux si valeur (A) ≠ valeur (B + C)

Différents symboles ont été utilisés pour représenter l'opération d'affectation.

Rutishauser a utilisé le symbole → dans le premier langage algorithmique (1951).

La notation FORTRAN utilise le symbole '=' et l'opérateur de relation est représenté par . E Q .

La notation ALGOL utilise le symbole ':=' .

$A := B + C$

exprime que la valeur de l'expression B + C est affectée au nom A (A désigne la mémoire, B + C la valeur à ranger dans cette mémoire).

Les désignations qui figurent à droite de l'opérateur d'affectation - ici B et C - représentent les valeurs des quantités ainsi désignées (ici les valeurs des quantités désignées par B et C)

La désignation qui figure à gauche de l'opérateur d'affectation représente un nom (une adresse) auquel on affecte la valeur de l'expression écrite à droite (définie après l'évaluation des opérations qui la constituent).

Toute occurrence ultérieure de A dans une partie droite désigne la quantité qui a été affectée à A.

Remarque 1 Il est possible de trouver une même désignation simultanément à droite et à gauche de l'opérateur d'affectation.

A := A + 1

La désignation à droite représente la valeur rangée dans la mémoire d'adresse A, la désignation à gauche représente l'adresse de cette mémoire A. L'affectation ci-dessus exprime.

- que la valeur actuelle de A est augmentée de 1
- que cette nouvelle valeur A + 1 sera dorénavant appelée A

Les algorithmes utilisant des formules de récurrence s'écrivent ainsi très simplement en utilisant ce concept d'instruction d'affectation.

Par exemple la formule itérative de Newton pour déterminer la racine carrée de A s'écrit :

$X := (X + A / X) / 2$

2) Structure de Blocs.

La structure de blocs permet d'attacher une signification locale aux identificateurs utilisés dans un programme - cette possibilité offre un double avantage :

- aux programmeurs elle permet la segmentation aisée des programmes en rendant possible une séparation des variables globales à plusieurs segments, dont les désignations sont communes, et des variables locales pour lesquelles le programmeur peut choisir librement les désignations.

- aux constructeurs de compilateurs elle permet de faire une économie de mémoire considérable en permettant une allocation dynamique automatique étant donné que deux blocs disjoints (non emboîtés) peuvent utiliser la même zone de mémoire pour l'allocation des variables.

Alors que la substitution dynamique permet de changer la valeur d'une même variable pendant l'exécution d'un programme, la structure de blocs permet de changer la variable associée à une même désignation.

Le même identificateur peut désigner des variables appartenant à des blocs différents et qui correspondent donc à des variables différentes pendant l'exécution du programme.

3) Exécution séquentielle.

Cette notion est aussi la transposition au niveau du langage de programmation d'une propriété caractéristique des ordinateurs. Le programme d'un ordinateur est composé d'instructions exécutées séquentiellement, il est cependant possible de sauter l'exécution d'une instruction ou d'un ensemble d'instructions au moyen d'instructions spéciales dites de rupture de séquence. La rupture de séquence peut être conditionnelle ou inconditionnelle suivant qu'elle dépend ou non de la valeur actuelle d'une quantité. Cela suppose que toutes les instructions ont une adresse : pour un programme en langage machine cette adresse est égale au numéro d'ordre (à une constante près).

Dans un programme en langage symbolique, il est inutile de définir l'adresse de chaque instruction, il suffit d'associer une adresse (une désignation) uniquement à celles des instructions correspondant aux points de reprise associés aux ruptures de séquence.

Les notions de substitution dynamique, de structure de blocs et de séquence qui permettent l'introduction du temps dans un formalisme sont certainement l'un des apports les plus significatifs des langages de programmation dans le domaine des mathématiques algorithmiques.

Remarque importante Les langages machine possèdent une propriété dynamique remarquable absente dans la plupart des langages algorithmiques : La possibilité pour un programme de se modifier lui-même, due à la représentation codée des instructions machines qui permet de les traiter comme des nombres. Cette possibilité existe dans le langage L I S P où les fonctions sont représentées par des listes qui peuvent être modifiées en cours d'exécution.

4) Exécution parallèle. (Simultanéité d'opérations)

Il s'agit aussi d'une propriété caractéristique des ordinateurs de la nouvelle génération. Sur les premières machines, une simultanéité d'exécution existait déjà au niveau des opérations d'entrée - sortie et de décodification et d'exécution des instructions.

Les ordinateurs actuels permettent l'exécution simultanée de plusieurs opérations indépendantes correspondant à un même programme ou à des programmes différents.

L'introduction de la simultanéité dans un langage algorithmique permet de supprimer l'ambiguïté relative aux effets de bords : il suffit de dire par exemple que les deux opérands d'une opération sont toujours évalués simultanément et que dans une liste d'indices ou de paramètres effectifs, tous les éléments sont évalués simultanément.

La simultanéité introduit également un autre aspect important au niveau des sous-programmes pouvant être appelés simultanément à partir d'autres sous-programmes : l'obligation de ne pas se modifier eux-mêmes pendant leur exécution c'est-à-dire d'être réentrants. La perte de cette propriété importante des programmes enregistrés se trouve dans ce cas compensé par le partage simultané d'un même sous-programme (activation multiple).

C H A P I T R E II

NOTATIONS SYNTAXIQUES

La description d'un langage de programmation présente deux aspects opposés et complémentaires selon qu'elle est destinée aux programmeurs qui écrivent des programmes ou aux compilateurs qui doivent analyser et traduire ces programmes.

Cette dualité génération-reconnaissance correspond aux deux fonctions de synthèse et d'analyse permettant la construction et la compilation des programmes.

La description à l'usage des programmeurs consiste essentiellement à définir des règles de génération permettant la construction d'expressions et d'instructions, éléments de base pour l'écriture des programmes et à associer aux règles de génération, les règles d'interprétation qui permettent de préciser la signification des éléments.

La description à l'usage des compilateurs consiste à définir des règles de reconnaissance ou d'analyse permettant la décomposition d'un programme en éléments de base et à associer à ces règles de reconnaissance des règles d'évaluation permettant la traduction des éléments correctement analysés.

L'ensemble des règles de génération (ou de reconnaissance) définit la syntaxe du langage et l'ensemble des règles d'interprétation (ou d'évaluation) définit la sémantique.

Dans un langage machine, la syntaxe est définie par le code d'instructions et la sémantique par la description des opérations, les instructions machine sont analysées et exécutées séquentiellement, cette traduction directe est appelée interprétation.

Dans un langage de programmation, les instructions ne sont pas analysées et exécutées séquentiellement car cela conduirait à utiliser un programme d'interprétation (interpréteur) capable d'analyser et d'exécuter directement les instructions du langage de programmation dans un langage machine. L'utilisation d'un tel interpréteur serait très coûteuse en particulier pour les parties répétitives étant donné qu'à chaque exécution d'un même programme on devrait répéter inutilement le processus d'analyse.

L'utilisation d'un compilateur permet de réaliser l'exécution d'un programme en deux phases distinctes.

- une phase de compilation qui permet l'analyse du programme source et la construction d'un programme objet équivalent en langage intermédiaire. (Dans certains cas, le langage intermédiaire est identique à un langage machine).

- une phase d'évaluation correspondant à l'exécution du programme objet.

La séparation de ces deux tâches permet de ne faire qu'une seule fois la compilation quel que soit le nombre d'exécutions du programme donné.

La compilation complète (langage source traduit en langage machine) suppose qu'il n'existe pas dans le programme source d'éléments définis pendant l'exécution.

Dans un langage à caractéristiques dynamiques, la compilation est toujours incomplète (langage source traduit en langage intermédiaire) : il subsiste une phase d'interprétation pendant l'exécution du programme qui correspond à l'interprétation du langage intermédiaire en langage machine (en ALGOL par exemple, réservation de mémoire pour les tableaux à bornes variables, appel de procédures récursives).

Description formalisée.

La description d'un langage est formalisée s'il existe un ensemble de règles définissant rigoureusement la syntaxe et la sémantique des programmes écrits dans ce langage.

La formalisation de la description a l'avantage de permettre une interprétation rigoureuse et universelle des programmes.

A l'heure actuelle, cette formalisation n'est possible que pour la syntaxe, la sémantique étant définie généralement au moyen de commentaires associés à la description syntaxique.

L'absence de formalisation pour la description sémantique conduit à des interprétations différentes et parfois opposées d'un même concept et représente un obstacle important pour l'automatisation des processus de compilation et d'interprétation.

I - FORMALISATION DE LA DESCRIPTION SYNTAXIQUE.

Vu sous l'angle de la programmation, la description syntaxique au moyen de règles de génération, semble effectivement le moyen le plus naturel, puisque le programmeur doit effectivement construire, générer ses programmes en respectant certaines règles d'écriture.

Vu sous l'angle de la compilation, le problème est fondamentalement différent car il s'agit ici du processus exactement inverse : celui de la reconnaissance des structures syntaxiques construites à partir de règles de génération.

La tâche d'un compilateur présente deux aspects différents et indépendants :

- analyse syntaxique d'un texte-source afin de déterminer s'il est correctement écrit.
- production d'un texte objet qui représente la traduction du texte-source.

Toutefois, la phase de production du texte objet ne doit s'exécuter que sur un texte syntaxiquement correct, d'où l'importance primordiale de la phase d'analyse syntaxique.

Théoriquement, il serait possible à partir des règles de génération de déterminer toutes les formes possibles d'une certaine entité (par exemple identificateur) ; la reconnaissance consisterait alors à comparer une structure donnée à toutes les structures permises rangées dans un certain ordre. Un tel procédé est inutilisable sur le plan pratique.

Néanmoins, il existe des cas où il est plus simple de donner la liste des valeurs d'une certaine entité, plutôt que les règles de syntaxe, spécialement si le nombre de formes possibles est peu élevé : par exemple on donne presque toujours sous forme de liste, les noms de fonctions standard utilisables dans un langage.

Chaînes.

Considérons un ensemble fini de symboles, par exemple les symboles de base d'un langage. Une suite quelconque de symboles (identiques ou non) constitue une chaîne. La longueur d'une chaîne est égale au nombre de symboles dont elle est formée. Les chaînes sont construites récursivement par concaté-
nation de symboles ou de chaînes.

Langage.

Etant donné l'ensemble des symboles de base d'un certain langage, toutes les chaînes construites à partir de ces symboles de base ne représentent pas automatiquement des assemblages permis dans ce langage.

On désigne par mots ou phrases, celles des chaînes qui représentent des assemblages autorisés par la description syntaxique du langage.

Dualité génération - reconnaissance.

Le problème est donc de définir un langage par les moyens les plus simples :

- soit en donnant une manière de générer récursivement toutes les chaînes du langage et elles seulement par un algorithme de génération
- soit en donnant une manière de reconnaître si une chaîne appartient au langage par un algorithme de reconnaissance ou d'analyse.

Les procédés de description syntaxique que nous allons étudier correspondent à des algorithmes de génération.

Classification des méthodes de description syntaxique.

1) Méthodes logiques dérivées de la logique formelle.

On se donne une liste de symboles formels (alphabet), et des règles de construction.

Exemple : Expressions arithmétiques FORTRAN [1]

1) Alphabet FORTRAN : 45 symboles formels ou caractères. (La barre verticale joue le rôle de séparateur).

A | B | --- | Z | 0 | 1 --- | 9 | (|) | + | - | * | / | = | , | .

2) Noms : les noms sont des chaînes de l'une des catégories suivantes :

- constantes entières : une constante entière est une chaîne

$K = k_1 k_2 \dots k_n$ telle que $\forall_i \quad 0 \leq k_i \leq 9$

- constantes flottantes : une constante flottante est une chaîne
 $F = f_1 f_2 \dots f_n$ telle qu'il y ait au moins et au plus un j tel que $f_j =$.

$$\text{et } \forall_i (i \neq j) \ 0 \leq f_i \leq 9$$

- variables entières : une variable entière est une chaîne
 $J = j_1 j_2 \dots j_n$ telle que :

- j_1 est une des lettres I, J, K, L, M, N
- $\forall_k (k \neq 1) \ j_k$ est une lettre ou un chiffre
- la longueur de I est inférieure ou égale à 6

- variables flottantes : une variable flottante est une chaîne

$\Gamma = \gamma_1 \gamma_2 \dots \gamma_n$ telle que :

- γ_1 est une lettre quelconque sauf I, J, K, L, M, N
- $\forall_k (k \neq 1) \ \gamma_k$ est une lettre ou un chiffre

La longueur de Γ est inférieure ou égale à 6

- noms de fonction : un nom de fonction est une chaîne

$\Phi = \varphi_1 \varphi_2 \dots \varphi_n$ telle que :

φ_1 est une lettre

$\forall_k (k \neq 1) \ \varphi_k$ est une lettre ou un chiffre avec l'une des deux conditions supplémentaires :

1) Longueur $(\Phi) \leq 6$ et

soit Φ n'apparaît pas dans un ordre dimension et longueur $(\Phi) < 4$.

soit $\varphi_n \neq F$

2) $4 \leq \text{longueur } (\Phi) \leq 7$ et $\varphi_n = F$

3) Indices : un indice est une chaîne de l'une des formes suivantes :

$$c * v + k$$

$$c * v - k$$

$$c * v$$

$$v + k$$

$$v - k$$

$$v$$

$$k$$

où c et k sont des constantes entières
et v une variable entière

4) Variables indicées : une variable indicée est une chaîne de la forme :

$$\Sigma (i_1, i_2, \dots, i_k)$$

où Σ est un nom de variable entière en flottante tel que :

- ou Σ apparaît dans un ordre dimension

- ou longueur (Σ) < 4 et $\sigma_n \neq F$

et $k \leq 3$

5) Expressions : l'ensemble des expressions est défini récursivement par les règles suivantes :

R1 - Une constante ou un nom de variable Ω est une expression de même mode (entier ou flottant) que Ω

$$\text{Soit } \Omega = w_1 w_2 \dots w_n$$

si : - $L(\Omega) < 4$ ou $w_n \neq F$

- Ω apparaît dans un ordre dimension

- i_1, \dots, i_k ($1 \leq k \leq 3$) sont des indices

alors $\Omega (i_1, i_2, \dots, i_k)$ est une expression du même mode que Ω

R2 - Si Ω est une expression qui ne commence pas par $+$ ou $-$ alors $+\Omega$ et $-\Omega$ sont des expressions de même mode que Ω

R3 - Si Ω est une expression alors (Ω) est une expression du même mode que Ω

R4 - Si Ω est un nom de fonction n-aire et e_1, e_2, \dots, e_n sont des expressions alors

$$\Omega (e_1, e_2, \dots, e_n)$$

est une expression du même type que Ω

R5 - Si Ω et π sont des expressions du même mode et π une expression qui ne commence pas par + ou - alors $\Omega + \pi$, $\Omega - \pi$, $\Omega * \pi$ et Ω / π sont des expressions de même mode que Ω et π

R6 - Si Ω et π sont des expressions et π est de mode flottant seulement si Ω est aussi de mode flottant, alors Ω et π n'étant pas de la forme $v_1 ** v_2$ (v_1 et v_2 désignant des constantes, des variables ou des fonctions) alors $\Omega ** \pi$ est une expression du même type que Ω

R7 - Il n'y a pas d'autres expressions arithmétiques que celles définies par les règles précédentes.

2) Méthodes algébriques.

Dans la méthode précédente, l'ensemble à partir duquel on construit les expressions ne comporte que les caractères de l'alphabet utilisé (ou symboles de base) et c'est par concaténation de ces symboles de base que l'on construit les expressions par application des règles de génération.

Dans les méthodes algébriques, l'ensemble est composé non seulement de symboles de base mais aussi de symboles intermédiaires et le langage est défini au moyen de relations binaires dont les éléments sont des symboles de base et des symboles intermédiaires.

Vocabulaire - grammaire - langage. [2]

Un vocabulaire est un ensemble fini de symboles dont l'un est distingué et constitue l'axiome.

Le vocabulaire comprend les symboles de base ou symboles terminaux et les symboles intermédiaires ou symboles non terminaux. L'axiome est un symbole intermédiaire.

Une grammaire est un algorithme qui permet de générer des mots (chaînes de symboles) à partir d'un vocabulaire et d'un ensemble fini de relations (ou règles de réécriture) de la forme :

$$\varphi \rightarrow \psi$$

où φ et ψ sont des mots du vocabulaire

Un langage est l'ensemble des mots générés par une grammaire

Génération des mots d'un langage.

Pour générer un mot d'un langage, on part de l'axiome qui constitue le point de départ et en utilisant les règles de réécriture, on forme de nouvelles chaînes jusqu'à l'obtention d'une chaîne finale qui ne contient que des symboles terminaux.

Exemple de description d'un langage utilisant une grammaire.

Définition d'une variable simple en ALGOL

1) Vocabulaire

Symboles terminaux = {A,B,C, -----, Z,0,1, -----,9}

Symboles intermédiaires = { λ , μ , ν , ω }

2) Axiome : c'est le symbole intermédiaire ω

3) Relations

$$\begin{array}{l} \lambda \rightarrow A \\ \lambda \rightarrow B \\ | \\ | \\ | \\ \lambda \rightarrow Z \\ \mu \rightarrow 0 \\ | \\ | \\ | \\ \mu \rightarrow 9 \\ \nu \rightarrow \lambda \\ \nu \rightarrow \nu \lambda \\ \nu \rightarrow \nu \mu \\ \omega \rightarrow \nu \end{array}$$

Remarque : afin d'éviter les ambiguïtés de symboles, on utilise l'alphabet grec pour les symboles intermédiaires et l'alphabet latin pour les symboles terminaux.

Pour générer un mot de ce langage, on part de l'axiome ω qui constitue le point de départ et en utilisant les règles de réécriture on forme une suite de nouvelles chaînes afin d'obtenir une chaîne finale qui ne contient que des symboles terminaux.

Exemple :

Relations utilisées

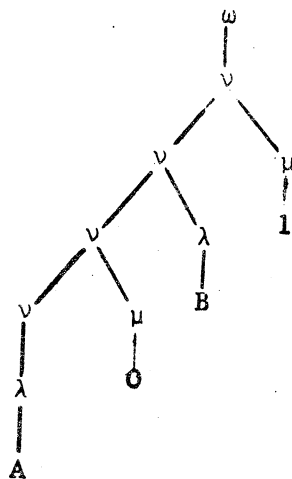
- $\omega \rightarrow v$
- $v \rightarrow v \mu$
- $\mu \rightarrow l$
- $v \rightarrow v \lambda$
- $\lambda \rightarrow B$
- $v \rightarrow v \mu$
- $\mu \rightarrow O$
- $v \rightarrow \lambda$
- $\lambda \rightarrow A$

Suite de chaînes formées

- v
- $v \mu$
- $v l$
- $v \lambda l$
- $v B l$
- $v \mu B l$
- $v O B l$
- $\lambda O B l$
- $A O B l$

La suite des chaînes ainsi formées constitue la dérivation de la chaîne finale.

Cette chaîne peut se représenter sous forme d'un arbre de dérivation ou arbre syntaxique de la façon suivante :



Métalangage et forme normale de Backus. [3]

Pour les langages de programmation, J. BACKUS a défini une forme plus pratique et plus concise, permettant le regroupement en une seule règle de toutes les règles qui ont la même partie gauche. Les parties droites des règles ainsi réunies portent le nom d'alternatives et sont séparées par une barre verticale équivalente au "ou d'énumération".

Le signe \rightarrow est remplacé par le symbole $:: =$

Les règles précédentes s'écrivent alors :

$$\begin{aligned} \lambda &:: = A \mid B \mid C \text{ -----} \mid Z \\ \mu &:: = 0 \mid 1 \mid \text{-----} \mid 9 \\ \nu &:: = \lambda \mid \nu \lambda \mid \nu \mu \\ \omega &:: = \nu \end{aligned}$$

De plus pour supprimer les restrictions concernant la différence de représentation des symboles terminaux et non terminaux, on utilise des méta-parenthèses : \langle et \rangle qui permettent :

- 1) la désignation des symboles non terminaux en utilisant le même alphabet que celui des symboles terminaux.
- 2) d'attacher une valeur sémantique aux éléments du vocabulaire non terminal.

Ce métalangage et ces conventions d'écritures sont connus sous le nom de forme normale de Backus.

L'exemple de la définition de variable simple donné précédemment devient en forme normale de Backus.

<LETTRE>	:: = A B C ----- Z
<CHIFFRE>	:: = 0 1 2 ----- 9
<IDENTIFIEUR>	:: = <LETTRE> <IDENTIFIEUR> <LETTRE> <IDENTIFIEUR> <CHIFFRE>
<VARIABLE SIMPLE>	:: = <IDENTIFIEUR>

Si l'on compare cette notation à la précédente, on remarque que :

- les symboles non terminaux sont toujours écrits entre métacrochets (on évite ainsi l'utilisation de l'alphabet grec)
- que le symbole "→" correspond au méta-symbole " ::= " et que l'utilisation de la barre verticale permet le regroupement de toutes les relations dont la partie gauche utilise le même symbole non terminal. Il y a donc équivalence entre les deux systèmes de description.

Utilisation de la forme normale de Backus pour la description des notations utilisées dans les langages de programmation.

1 - La notation complètement parenthésée.

C'est une notation dans laquelle on ne définit pas de priorités d'opérateurs, ni d'ordre d'évaluation pour les opérations de même priorité.

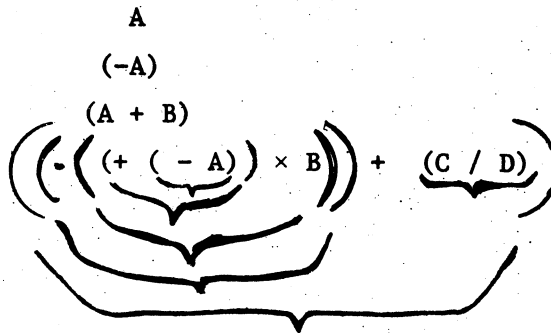
L'interprétation des expressions est déterminée à l'aide des parenthèses : toute opération est écrite à l'intérieur d'une paire de parenthèses.

On peut considérer cette notation comme une notation de référence, les autres notations peuvent être définies à partir de cette notation et représentent des formes abrégées.

Cette notation est définie de la façon suivante en forme normale de Backus (on admet que les définitions de <variable> et de <constante> sont connues).

<opérateur binaire>	:: = + - × / †
<opérateur unaire>	:: = + -
<opérande>	:: = <variable> <constante> <expression>
<expression>	:: = <opérande> (<opérateur unaire> <expression>) (<expression> <opérateur binaire> <opérande>)

Exemples :



sont des expressions complètement parenthésées.

(- A + B)
 A + (B × C)
 (A + B × C)

ne sont pas des expressions complètement parenthésées.

Remarque : tous les opérateurs sont unaires ou binaires donc

((a+b) + c) est à priori correct

tandis que a+b+c ne l'est pas.

2 - Ordre lexicographique - Ordre d'évaluation - Priorité des opérateurs.

Ordre lexicographique et ordre dynamique.

On désigne par ordre lexicographique (ou ordre d'écriture) l'ordre dans lequel sont écrits les différents éléments d'une expression.

Par exemple pour l'expression :

$$A + B \times C \uparrow D$$

l'ordre lexicographique des variables est (de gauche à droite)

A, B, C, D

l'ordre lexicographique des opérations est (de gauche à droite)

+, ×, ↑

On désigne par ordre dynamique (ou ordre d'évaluation) l'ordre dans lequel sont évaluées les différentes opérations.

Pour l'exemple ci-dessus l'ordre dynamique dépend de la priorité des opérations et de l'ordre d'évaluation (de gauche à droite ou de droite à gauche).

Sans priorité d'opérations et avec un ordre d'évaluation de gauche à droite, l'ordre dynamique des opérations coïncide avec l'ordre lexicographique de gauche à droite. En général, l'usage des parenthèses, la priorité des opérations et le sens d'évaluation font que les 2 ordres sont différents.

L'un des buts de la compilation est précisément le passage d'un ordre lexicographique à un ordre dynamique.

Ordre lexicographique direct et inverse.

L'ordre lexicographique peut-être défini de gauche à droite ou de

droite à gauche suivant le sens de lecture choisi. En effet, une expression, s'il est plus commode de l'écrire normalement de gauche à droite peut être analysée sur une machine aussi bien dans un sens que dans l'autre.

Il convient donc d'ajouter aux règles précédentes une règle précisant le sens de l'ordre lexicographique.

L'ordre lexicographique, et la priorité des opérations définissent sans ambiguïté l'ordre dynamique.

En forme normale de Backus, le sens de l'ordre lexicographique est fixé par la définition récursive de <expression>

Ainsi dans la description de la notation complètement parenthésée la définition récursive :

<expression> ::= (<expression> <opérateur binaire> <opérande>)

impose une direction d'analyse de gauche à droite

2) Classification des notations utilisées pour les expressions.

La notation complètement parenthésée, très utile comme notation de référence a l'inconvénient de nécessiter un grand nombre de parenthèses :

On peut alléger considérablement l'écriture des expressions en donnant deux règles supplémentaires :

- règle d'exécution de gauche à droite (ou de droite à gauche).

- règle des priorités d'opérateurs.

qui permettent de supprimer des parenthèses sans introduire d'ambiguïtés.

Toutes les notations utilisées pour les expressions peuvent se classer en fonction de ces 2 règles.

Exemple 1 : notation de gauche à droite sans priorités d'opérateurs.

Dans cet exemple et les suivants nous supposons connues les définitions de <variable> et <constante> et nous n'utilisons que des opérateurs unaires et binaires.

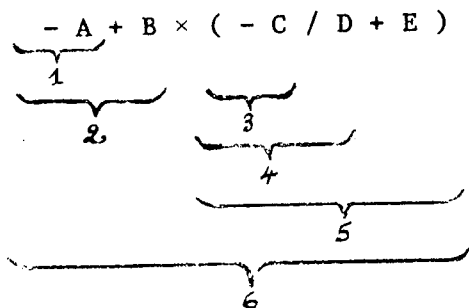
<opérande> ::= <variable> | <constante> | (expression)
 <expression> ::= <opérande> | <opérateur unaire> <expression> |
 <expression> <opérateur binaire> <opérande>

La règle de gauche à droite est impliquée par la définition récursive à gauche de <expression>.

L'expression

- A + B × (- C / D + E)

est interprétée comme



et est équivalente à l'expression complètement parenthésée

$$\left(\left((- A) + B \right) \times \left(\left((- C) / D \right) + E \right) \right)$$

Exemple 2 : notation de gauche à droite sans priorités d'opérateurs et sans parenthèses.

C'est une notation encore plus simple que la notation de gauche à droite sans priorités, du fait de l'absence de parenthèses.

L'absence de parenthèses implique une associativité systématique de gauche à droite.

<opérande> ::= <variable> | <constante>

<expression> ::= <opérande> | <opérateur unaire> <opérande> |
 <expression> <opérateur binaire> <opérande>

L'expression

$A \times Y + B \times Y + C \times Y + D$ est interprétée comme

et est équivalente à l'expression complètement parenthésée.

$(((((A \times Y) + B) \times Y) + C) \times Y) + D)$

Remarque :

L'absence de parenthèses rend plus compliquée l'écriture des expressions qui ne se réduisent pas à des schémas de Hörner et impossible l'écriture de certaines expressions.

Par exemple si l'on veut écrire $(A \times B) + (C \times D)$, il faudra deux instructions d'affectations :

$X := A \times B$

$Y := C \times D + X$

et dans la deuxième instruction il faut faire attention à l'ordre des opérations en effet : $X + C \times D$ aurait le sens de $(X + C) \times D$.

L'expression $A \times B + C \times D$ aurait le sens de $((A \times B) + C) \times D$

Cette notation a été utilisée sur de petites machines car elle permet une compilation directe et simple des expressions : il suffit de remarquer en effet que l'alternance régulière d'opérandes et d'opérateurs permet, en un seul balayage des expressions, la génération directe d'instructions à une adresse.

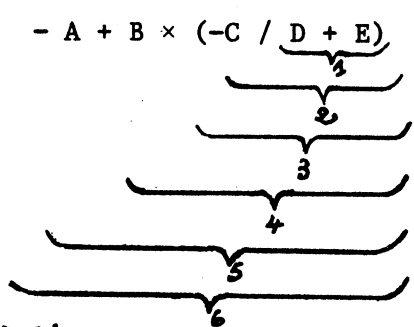
Exemple 3 : notation de droite à gauche sans priorité d'opérateurs

<opérande> ::= <variable> | <constante> | (<expression>)
 <expression> ::= <opérande> | <opérateur unaire> <expression> |
 <opérande> <opérateur binaire> <expression>

L'expression :

$$- A + B \times (-C / D + E)$$

est interprétée comme



et est équivalente à l'expression complètement parenthésée.

$$(- (A + (B \times (- (C / (D + E))))))$$

On remarque la différence d'interprétation avec l'exemple 1

Remarque : cette notation a été utilisée dans le langage IT [4] avec la particularité que pour chaque opération les opérandes sont considérés dans l'ordre habituel à savoir :

- l'opérande associé à un opérateur unaire est écrit à droite de celui-ci.
- le premier opérande et le deuxième opérande d'une opération binaire sont écrits respectivement à gauche et à droite de celui-ci.

L'ordre lexicographique des variables n'est donc pas réellement de droite à gauche ; seul l'ordre lexicographique des opérations est de droite à gauche.

Exemple 4 : notation de droite à gauche avec priorité d'opérateurs.

Cette notation est utilisée dans le langage GAT [5]

```

<primaire> ::= <variable> | <constante> | (<expression>)
<terme>    ::= <primaire> | <primaire> <opérateur multiplicatif> <terme>
<expression> ::= <terme> | <opérateur additif> <terme> |
                <terme> <opérateur additif> <expression>
    
```

Comme dans la notation de droite à gauche sans priorités les opérandes de chaque opération binaire sont pris dans l'ordre habituel, premier à gauche, second à droite.

La priorité des opérateurs est dans l'ordre décroissant :

- opérateur multiplicatif
- opérateur additif

Exemple 5 : notation de gauche à droite avec priorités d'opérateurs.

Cette notation est utilisée pour l'écriture des expressions dans les langages FORTRAN et ALGOL.

Les expressions FORTRAN ont été décrites dans ce chapitre.

Les expressions ALGOL sont décrites dans le chapitre IV.

Exemple 6 : notation mixte.

La règle de gauche à droite (ou de droite à gauche) revient à dire que les opérations de même priorité sont toujours interprétées de gauche à droite (ou de droite à gauche).

On donne ainsi un sens dynamique à l'expression

$$A \oplus B \text{ ----- } \oplus Z$$

toujours équivalente à l'expression complètement parenthésée

$$(((A \oplus B) \oplus \text{-----}) \oplus Z)$$

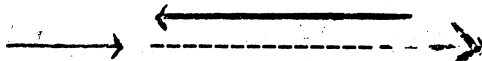
PL/I est le seul langage qui admette 2 règles différentes selon la nature des opérateurs.

- pour l'opérateur d'exponentiation et les opérateurs additifs unaires la règle d'association est de droite à gauche.

- pour tous les autres opérateurs, la règle d'association est de gauche à droite.

Ainsi l'expression PL/I :

$$A + B \times C ** (- D) ** E / F$$



est équivalente à l'expression complètement parenthésée :

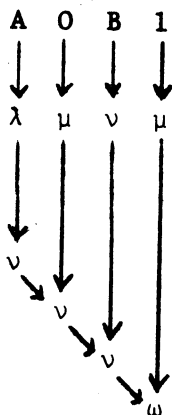
$$(A + ((B \times (C ** ((- D) ** E))) / F))$$

Dans la compilation c'est évidemment le problème inverse de la reconnaissance ou analyse syntaxique qu'il faut résoudre.

Il s'agit en fait d'utiliser les règles en permutant les parties droite et gauche pour déterminer si un mot est correct c'est-à-dire s'il conduit à l'axiome correspondant.

L'analyse syntaxique consiste en fait à produire un arbre de décomposition en partant des symboles terminaux d'un mot et de vérifier que l'extrémité vérifie l'axiome relatif à ce mot.

Ainsi en reprenant l'exemple précédent, l'analyse syntaxique se schématise ainsi



Dans les techniques de compilation basées sur l'analyse syntaxique nous verrons comment ces arbres sont construits, et utilisés pour la génération du programme objet. Dans tous les cas l'analyse syntaxique de chaque mot est nécessaire afin de reconnaître si ce mot est bien formé ou non ; le propre des compilateurs syntaxiques est d'utiliser cette analyse non seulement dans un but de vérification mais d'articuler tout le processus de génération sur cette analyse.

Réversivité à droite ou à gauche - Priorité de gauche à droite - Ambiguïté.

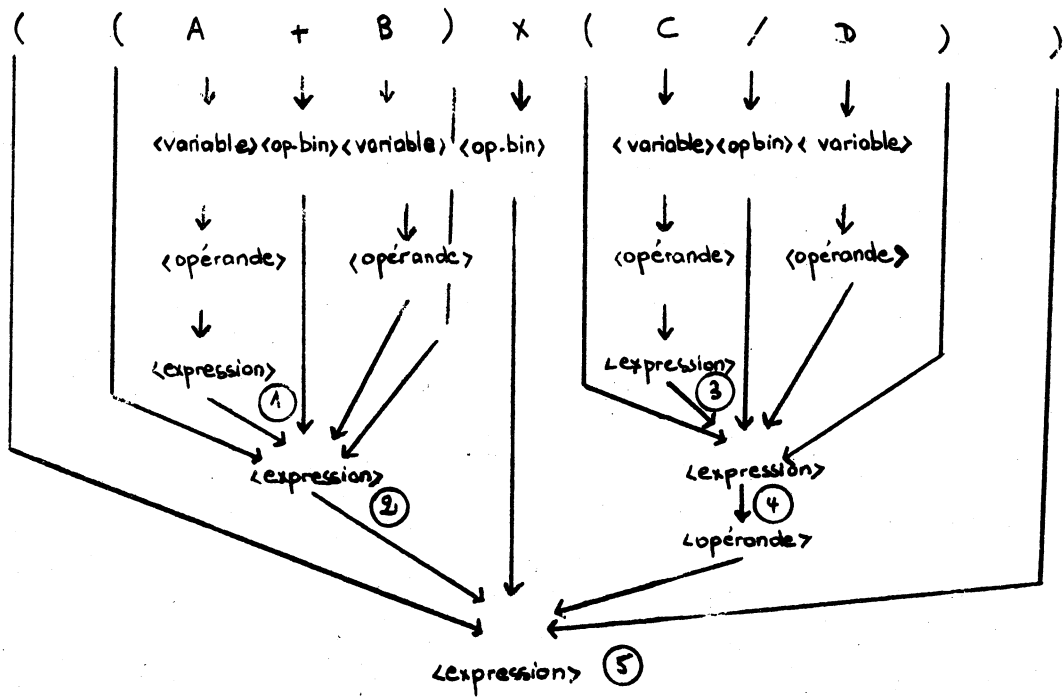
Dans l'exemple relatif à la notation complètement parenthésée, on remarque que la définition de <expression> est réursive.

<expression> ::= (<opérande>) | (<expression> <opérateur binaire> <opérande>)

Cette définition est dite réursive à gauche : en effet la variable métalinguistique <expression> est le symbole non terminal le plus à gauche dans la deuxième alternative qui définit <expression> . Ceci signifie que dans la suite des substitutions ou ce qui revient au même l'analyse syntaxique de l'expression doit être réalisée de gauche à droite.

Autrement dit la définition ci-dessus détermine un sens d'interprétation des expressions de gauche à droite.

Exemple :

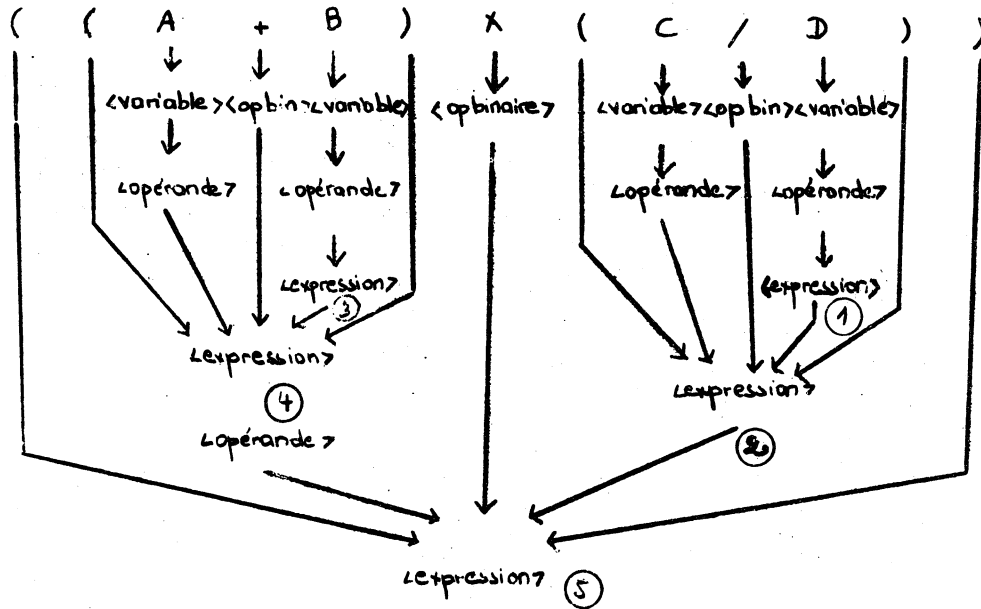


Adoptons maintenant une convention d'interprétation de droite à gauche, la définition devient :

$\langle \text{expression} \rangle ::= \langle \text{opérande} \rangle | (\langle \text{opérande} \rangle \langle \text{opérateur binaire} \rangle \langle \text{expression} \rangle)$

La définition est réursive à droite et les opérations de substitution doivent être effectuées de droite à gauche.

L'exemple précédent est maintenant analysé comme suit :



On pourrait également écrire la définition d'expression de la façon suivante :

$\langle \text{expression} \rangle ::= (\langle \text{opérande} \rangle) | (\langle \text{expression} \rangle \langle \text{opérateur binaire} \rangle \langle \text{expression} \rangle)$

C'est-à-dire laisser arbitraire le sens d'interprétation : on dit qu'il y a dans ce cas ambiguïté syntaxique; cela signifie que l'on a deux décompositions possibles, deux arbres syntaxiques différents suivant que l'on analyse de droite à gauche ou de gauche à droite.

Voici un autre exemple d'ambiguïté syntaxique tiré de la version non révisée du rapport ALGOL 60.

Nous utilisons les définitions suivantes (allégées pour l'exemple) :

1. <expr arith simple> ::= <variable> | (<expr arith>)
 2. <expr arith> ::= <expr arith simple> | <proposition si> <expr arith simple>
sinon <expr arith>
 3. <opér de relation> ::= < | < | = | > | > | ≠
4. <relation> ::= <expr arith> <opér de relation> <expr arith>
5. <expr bool simple> ::= <variable> | <relation> | (<expr bool>)
 6. <expr bool> ::= <expr booléenne simple> | <proposition si> <expr bool simple>
sinon <expr bool>
 7. <proposition si> ::= si <expr bool> alors

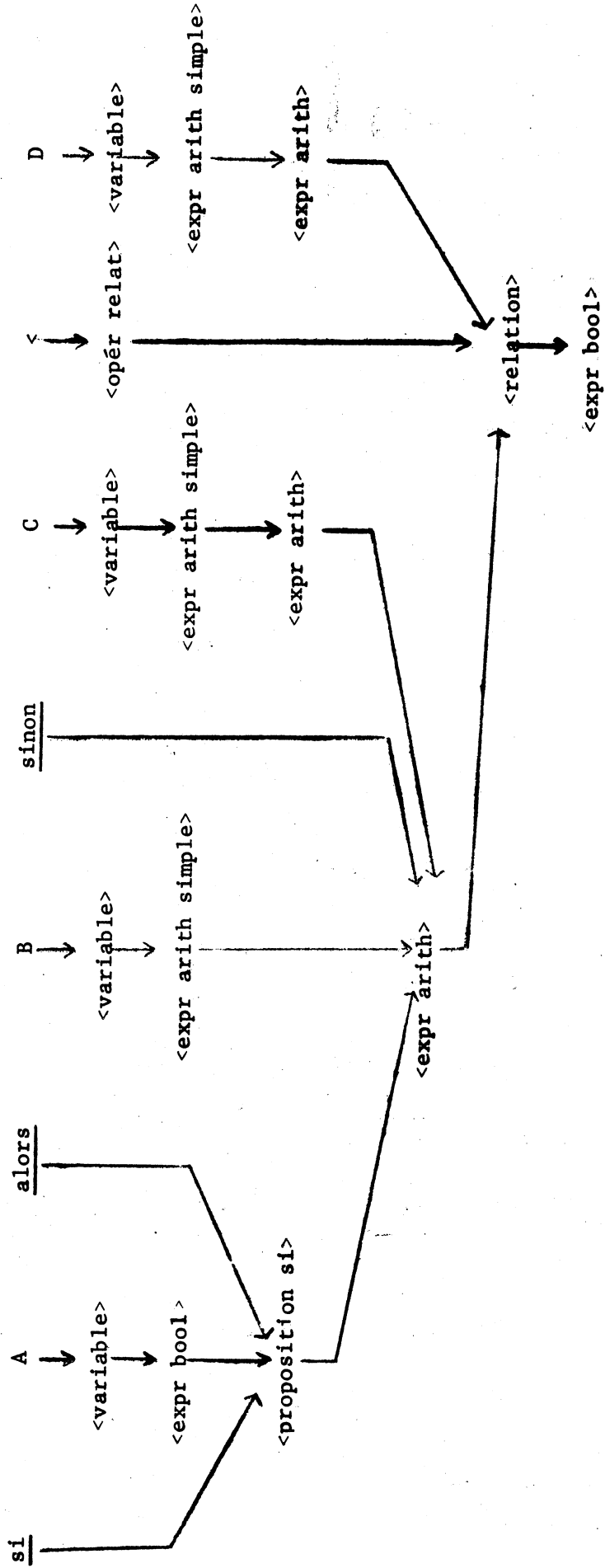
Considérons maintenant la chaîne suivante :

si A alors B sinon C < D

L'analyse syntaxique de cette expression est possible de deux manières différentes selon que l'on considère :

- ① "si A alors B sinon C" et "D" comme deux expressions arithmétiques séparées par l'opérateur de relation <, si A alors B sinon C < D est alors une expression booléenne simple (relation)
- ② "A", "B" et "C < D" comme trois expressions booléennes, si A alors B sinon C < D étant alors une expression booléenne conditionnelle.

La première analyse donne :



Ce cas d'ambiguïté est d'ailleurs identique à celui que nous avons donné pour la notation complètement parenthésée, c'est-à-dire est due à la définition de la relation :

$\langle \text{relation} \rangle ::= \langle \text{expr arith} \rangle \langle \text{opér de relation} \rangle \langle \text{expr arith} \rangle$

qui contient le symbole intermédiaire $\langle \text{expr arith} \rangle$ une fois à gauche et une fois à droite. Il en résulte que le sens d'analyse reste arbitraire et il est normal que l'on obtienne 2 analyses syntaxiques différentes suivant le sens d'interprétation choisi.

Cette ambiguïté peut être levée de deux façons différentes :

- a) au moyen d'une information supplémentaire fournie par le contexte mais ceci a le grave inconvénient de compliquer le processus d'analyse syntaxique qui n'est plus strictement séquentiel (par exemple dans le cas ci-dessus on peut utiliser la déclaration de B : de type arithmétique ou booléen, qui permettrait de lever l'indétermination de la décomposition).
- b) en modifiant la définition syntaxique de $\langle \text{relation} \rangle$ afin d'imposer un sens d'analyse de gauche à droite (sens habituel de concaténation des chaînes de symboles), on peut par exemple définir relation comme :

$\langle \text{relation} \rangle ::= \langle \text{expr arith simple} \rangle \langle \text{oper de relation} \rangle \langle \text{expr arith} \rangle$

ainsi à gauche de l'opérateur de relation on ne peut trouver qu'une variable ou une expression entre parenthèses, l'exemple donné devra alors s'écrire :

si A alors B sinon (C < D)

ou si l'on désire avoir l'autre interprétation :

(si A alors B sinon C) < D

Les auteurs d'ALGOL dans la version révisée ont donné la définition suivante :

<relation> ::= <expr arith simple> <opér de relation> <expr arith simple>

Un exemple d'ambiguïté de même nature existait dans la version non révisée d'ALGOL sur la syntaxe de l'instruction POUR.

L'instruction POUR était définie comme une instruction incondi-
tionnelle, de telle sorte qu'il était possible d'écrire une instruction POUR comme
instruction suivant alors dans une instruction conditionnelle.

Considérons alors l'instruction conditionnelle :

si A alors pour I := 1 pas 1 jusqu'à N faire si B alors C:=D sinon C:=E

Deux décompositions syntaxiques étaient possibles.

si A alors pour I:= 1 pas jusqu'à N faire si B alors C:=D sinon C:=D,

<instruction conditionnelle>

<instruction pour>

<instruction si>

ou bien :

si A alors pour I := 1 pas 1 jusqu'à N faire si B alors C := D sinon C := E

<instruction si>

<instruction pour>

<instruction conditionnelle>

Dans le rapport révisé, l'instruction POUR constitue une troisième catégorie d'instruction, les deux autres catégories : (instructions conditionnelles et inconditionnelles demeurant inchangées. La syntaxe de l'instruction si devient :

si <expres. Booléenne> alors <instruction POUR>.

de telle sorte que l'exemple ci-dessus a nécessairement la première interprétation (instruction si contenant une instruction POUR)

On peut remarquer que la nouvelle syntaxe restreint beaucoup l'emploi de l'instruction POUR en la considérant comme une instruction à part, alors qu'en fait elle peut toujours être classée dans la catégorie conditionnelle ou inconditionnelle selon la nature de l'instruction qui suit le symbole faire.

On pourrait par exemple classer l'instruction POUR dans la catégorie instruction conditionnelle lorsque l'instruction qui suit le symbole faire, est conditionnelle et dans la catégorie instruction inconditionnelle si l'instruction qui suit le symbole faire est inconditionnelle.

Si l'on fait la même distinction pour l'instruction conditionnelle en la classant dans l'une ou l'autre catégorie suivant la nature de l'instruction qui suit sinon (l'instruction suivant alors devant toujours être incondi-
tionnelle) alors on permet une interprétation non ambiguë de toutes les écritures possibles : cela revient à dire que l'on associe toujours à un symbole sinon le dernier symbole "si" qui le précède.

Priorités des opérateurs. Dans la forme normale de Backus, cette priorité peut s'exprimer au moyen de symboles intermédiaires. Considérons par exemple les deux ensembles de règles qui définissent des expressions arithmétiques simples :

<opérateur mult> ::= × | /
 <opérateur add> ::= + | -
 <opérateur arith> ::= <opér mult> | <opér add>

I { <primaire> ::= <variable> | (<expr arith>)
 <expr arith> ::= <primaire> | <expr arith> <opér arith> <primaire>

II { <facteur> ::= <variable> | (<expr arith>)
 <terme> ::= <facteur> | <term> <opér mult> <facteur>
 <expr arith> ::= <term> | <expr arith> <opér add> <terme>

Les groupes de relation I et II expriment tous les deux les règles d'écriture des expressions avec une priorité de gauche à droite. Mais les règles II expriment de plus, la priorité de l'opérateur multiplicatif sur l'opérateur additif grâce aux définitions supplémentaires de <facteur> et de <terme>.

Remarque : la priorité supposée par le programmeur d'une entité sur une autre est exprimée en forme normale de Backus par le fait que l'une des deux est reconnue à un niveau inférieur. Par exemple dire que <terme> à une priorité inférieure à <facteur> c'est dire que <facteur> est reconnu avant <terme> dans l'analyse syntaxique.

- 3 - Priorité de gauche à droite pour les opérations de priorité identique : définitions récurives à gauche de <terme> et <expr arith>

Interférence de la syntaxe et de la sémantique.

Nous avons vu dans les paragraphes précédents deux exemples d'ambiguïté provenant d'une définition trop large au niveau d'un symbole intermédiaire. Ce genre d'ambiguïté peut toujours être levé au niveau de la syntaxe de deux façons différentes.

- en restreignant la définition d'un symbole intermédiaire (cas de la relation et de l'instruction POUR)
- en donnant des règles supplémentaires qui ne sont pas directement exprimables avec le formalisme métalinguistique.

Nous allons expliquer ce dernier point en prenant l'exemple de l'instruction conditionnelle ALGOL :

Cette instruction conditionnelle est définie par la syntaxe suivante :

<instr si> ::= <proposition si> <instr incond>
<instr cond> ::= <instr si> | <instr si> sinon <instr>

Le point important est que cette syntaxe exige que l'instruction qui suit la proposition si soit inconditionnelle. Supposons en effet que l'instruction suivant la proposition si puisse être conditionnelle alors l'exemple suivant :

si a alors si b alors I1 sinon I2
pourra s'interpréter soit comme :

si a alors si b alors I1 sinon I2

 <instr cond>

 <instr si>

soit comme :

si a alors si b alors I1 sinon I2

 <instr si>

 <instr cond>

Toutefois ce genre d'ambiguïté peut être levé sans modifier la syntaxe mais en donnant une règle de correspondance des si et sinon lorsque leur nombre n'est pas égal, par exemple un sinon est toujours associé avec le premier si situé à sa gauche : la description obtenue sera alors non ambiguë. Cette dernière règle est utilisée dans le langage PL/I.

Remarquons que cette façon d'associer les si et sinon est la règle normale dans les expressions conditionnelles où le sinon n'est jamais facultatif.

Un autre cas intéressant d'ambiguïté provient de l'utilisation d'un même symbole de base dans deux règles syntaxiques définissant des entités fort différentes :

a) cas des délimiteurs si et sinon

considérons la suite de symboles de base :

si A alors B sinon C

- si B et C sont des instructions procédures sans paramètres alors la chaîne représente une instruction conditionnelle.

- si B et C sont des variables, alors on a

- soit une expression arithmétique si B et C sont des quantités arithmétiques
- soit une expression booléenne si B et C sont des quantités booléennes
- soit une expression de désignation si B et C sont des étiquettes

Si l'on veut considérer les délimiteurs si, alors, sinon comme des opérateurs et leur affecter une priorité analogue à celle des opérateurs arithmétiques, logiques et de relation, cette priorité variera selon les cas d'utilisation de ces délimiteurs.

Considérons les 2 instructions (qui expriment le même résultat pour R).

R := si A alors B sinon C
si A alors R := B sinon R := C

On voit que le symbole si ne joue pas du tout le même rôle dans les deux constructions vis à vis du symbole :=.

De même considérons les deux constructions :

R := si A alors B sinon C ∨ D
R = si A alors B sinon C ∨ D

dans le premier cas la signification est la suivante :

$R := \underline{\text{si}} \ A \ \underline{\text{alors}} \ B \ \underline{\text{sinon}} \ (C \ \vee \ D)$

et dans le second :

$R = (\underline{\text{si}} \ A \ \underline{\text{alors}} \ B \ \underline{\text{sinon}} \ C) \ \vee \ D$

La valeur du symbole sinon vis à vis du symbole \vee dépend effectivement de ce qui précède (présence de " := " "ou" = ").

Nous donnerons enfin un dernier exemple relatif au symbole ":"

la chaîne :

si A alors B sinon C : D

peut être considérée :

- soit comme deux expressions séparées par ":" qui constituent les bornes supérieure et inférieure d'une déclaration de tableau.
- soit comme une instruction conditionnelle où D est une instruction procédure (étiquetée C)

Insuffisances du formalisme utilisé dans la description syntaxique.

Dans le paragraphe précédent, nous avons montré des exemples d'ambiguïtés provenant soit de l'imprécision d'une règle de syntaxe, soit de l'usage multiple d'un même symbole de base.

Dans ces deux cas, on peut lever l'ambiguïté :

- soit en reformulant plus strictement une règle syntaxique.

- soit en augmentant le nombre de symboles de base de manière à éviter plusieurs interprétations sémantiques différentes d'un même assemblage de symboles.
- soit en donnant des règles supplémentaires non exprimables dans le formalisme métalinguistique.

En fait, le principal intérêt de la formalisation syntaxique vue du côté du programmeur est d'être concise et précise en même temps et il est souvent préférable de donner sous forme de commentaires les règles qui se prêtent mal à la formalisation.

Voici deux exemples montrant l'insuffisance du formalisme présenté.

Expressions mixtes : la syntaxe d'ALGOL permet l'écriture d'expressions mixtes c'est-à-dire comportant des opérations portant sur des opérandes de type entier et réel. Il serait possible de séparer la syntaxe des expressions de type entier et de type réel ou même d'inclure le type des opérandes et du résultat dans la définition de facteur, terme et expression arithmétique. On alourdirait alors considérablement la description syntaxique des expressions.

Les auteurs d'ALGOL ont choisi de donner des règles syntaxiques aussi simples que possible et de reporter dans des paragraphes annexes les règles supplémentaires nécessaires à la description complète.

Ordre d'évaluation des primaires d'une expression et des parties gauche et droite d'une instruction d'affectation.

Nous avons vu que la forme normale de Backus permet d'exprimer pour une expression :

- la priorité de gauche à droite en précisant l'ordre d'évaluation
- la priorité des parenthèses
- la priorité des opérateurs

On peut également exprimer les règles d'associativité de l'addition et de la multiplication au moyen des règles suivantes :

$\langle \text{facteur} \rangle ::= \langle \text{variable} \rangle \mid \langle \text{expr arith} \rangle$
 $\langle \text{terme} \rangle ::= \langle \text{facteur} \rangle \mid \langle \text{terme} \rangle \times \langle \text{terme} \rangle \mid \langle \text{terme} \rangle / \langle \text{facteur} \rangle$
 $\langle \text{expr arith} \rangle ::= \langle \text{terme} \rangle \mid \langle \text{expr arith} \rangle + \langle \text{expr arith} \rangle \mid \langle \text{expr arith} \rangle - \langle \text{terme} \rangle$

Les règles de commutativité seraient définies par :

$\langle \text{terme} \rangle ::= \langle \text{facteur} \rangle \mid \langle \text{terme} \rangle \times \langle \text{facteur} \rangle \mid \langle \text{facteur} \rangle \times \langle \text{terme} \rangle \mid \langle \text{terme} \rangle / \langle \text{facteur} \rangle$
 $\langle \text{expr arith} \rangle ::= \langle \text{terme} \rangle \mid \langle \text{expr arith} \rangle + \langle \text{terme} \rangle \mid \langle \text{terme} \rangle + \langle \text{expr arith} \rangle \mid$
 $\langle \text{expr arith} \rangle - \langle \text{terme} \rangle$

Effet de bord :

On appelle effet de bord la possibilité de modifier dans un corps de procédure (qui est un bloc au sens ordinaire) des variables non locales au corps de cette déclaration de procédure.

Considérons une opération binaire commutative au sens ordinaire

$\alpha \omega \beta$

et supposons que α soit un indicateur de fonction dont l'appel (activation du bloc correspondant à sa déclaration) modifie la variable β non locale à ce bloc, alors dans cette hypothèse

$\alpha \omega \beta \neq \beta \omega \alpha$

si l'on admet que les deux opérands sont déterminées avant que l'opération s'exécute.

En effet, soit β' la valeur de β après l'appel de la fonction α , alors le résultat de la première opération est $\alpha \omega \beta'$ tandis que celui de la seconde est $\beta \omega \alpha$.

Le seul point commun est qu'après l'opération la valeur de β est devenue β' dans les deux cas.

Une ambiguïté d'interprétation devient possible lorsqu'une expression fait intervenir deux opérations commutatives de priorité différente par exemple

$$\alpha + \beta \times \gamma$$

Normalement la priorité des opérateurs indique que le produit doit être effectué d'abord et l'addition ensuite. Cependant si β est une fonction dont l'appel modifie les valeurs de α et γ le résultat peut être différent selon que les valeurs de α et γ utilisées sont celles définies avant ou après l'évaluation de β .

Remarque : alors que l'effet de bord à l'intérieur d'une procédure est une possibilité intéressante, il convient de souligner le danger d'un tel effet dans l'évaluation des expressions. La solution la plus logique consiste à préciser explicitement que l'ordre d'évaluation des primaires est indéfini : par exemple pour toute opération binaire, les deux primaires sont évalués simultanément.

Cas des variables indicées : la liste d'indices est une liste d'expressions pouvant contenir des fonctions ; ainsi l'évaluation d'une expression d'indice peut affecter la valeur d'un élément figurant dans une autre expression.

Cas de l'instruction d'affectation : si la partie gauche d'une instruction d'affectation est une variable indicée, l'évaluation des expressions d'indice peut modifier des variables figurant dans l'expression de la partie droite.

Il convient donc de préciser si l'évaluation de la partie gauche (détermination de l'adresse) est faite avant celle de la partie droite (détermination de la valeur).

Dans le rapport révisé d'ALGOL, la règle est la suivante :

L'instruction d'affectation est interprétée en trois étapes :

1) Toutes les expressions d'indice apparaissant dans les variables de la partie gauche sont évaluées successivement de gauche à droite.

2) L'expression qui constitue la partie droite de l'instruction est évaluée.

3) La valeur de cette expression est affectée à toutes les variables de la partie gauche, les indices ayant les valeurs calculées dans 1.

La règle logique, pour tenir compte des possibilités d'effet de bord devrait être que toutes les variables d'une expression soient évaluées séquentiellement de gauche à droite c'est-à-dire que l'ordre d'évaluation des opérandes et des opérations soient strictement celui défini par l'écriture post fixée de l'expression.

L'inconvénient d'une telle solution est qu'elle impose un mécanisme d'exécution du programme objet et interdit pratiquement toute possibilité d'optimisation.

La solution générale consiste à préciser que pour toute opération (y compris les variables indicées et les fonctions), il n'y a pas d'ordre d'accès spécifique aux opérandes (ou aux arguments). Si les opérandes (ou les arguments) d'une opération sont évalués simultanément, l'effet de bord est bien indéfini.

On peut encore signaler parmi les insuffisances de la description syntaxique telle que nous l'avons définie, la difficulté d'exprimer la longueur d'un certain mot construit à partir des règles.

Par exemple si l'on veut limiter la longueur c'est-à-dire le nombre de symboles de base d'un identificateur, cette propriété n'est pas exprimable simplement dans le métalangage.

Si par exemple on veut définir syntaxiquement des identificateurs de longueur 3 au plus, on peut écrire :

```
<identificateur> ::= <lettre> | <lettre> <lettre> | <lettre> <chiffre> |
                   <lettre> <lettre> <lettre> | <lettre> <lettre> <chiffre> |
                   | <lettre> <chiffre> <lettre> | <lettre> <chiffre> <chiffre>
```

On perd ainsi le principal avantage de la notation qui est de définir récursivement les variables intermédiaires.

III - AUTRES METHODES DE DESCRIPTION SYNTAXIQUE.

1) Cartes syntaxiques. [6]

On peut définir un symbolisme graphique strictement équivalent à la forme normale de Backus avec les conventions suivantes :

- les symboles de base sont représentés à l'intérieur d'un cercle : ○
- les variables intermédiaires (éléments non terminaux) sont définies par des ovales : ○

ainsi la représentation suivante :

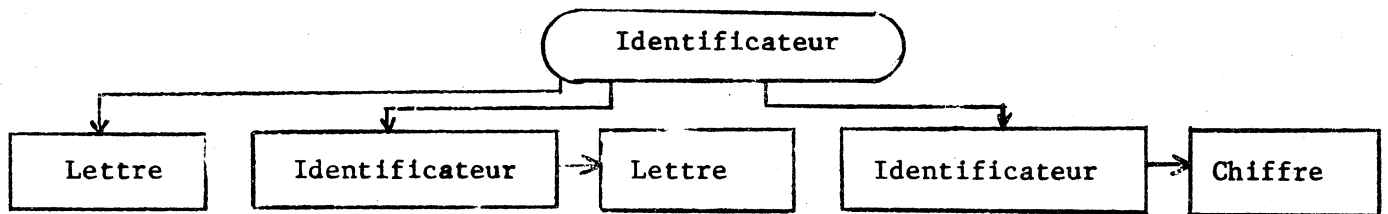
○ est équivalente à : <expression>

- la barre verticale est représentée par une flèche verticale dirigée vers le bas (en fait, il y a autant de flèches verticales que d'alternatives dans une définition).
- la concaténation des symboles (terminaux et non terminaux) est indiquée par des flèches horizontales → dirigées vers la droite.
- les variables intermédiaires déjà définies sont représentées par des rectangles.

La définition de l'identificateur ALGOL en notation de Backus

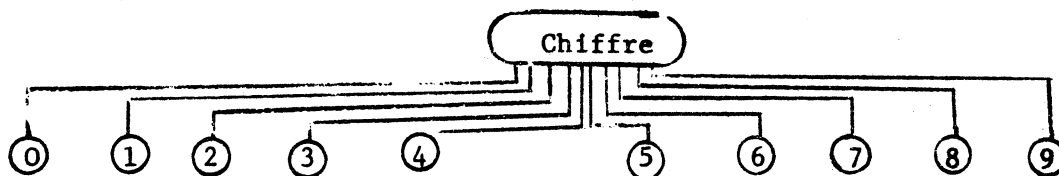
$\langle \text{identificateur} \rangle ::= \langle \text{lettre} \rangle \mid \langle \text{identificateur} \rangle \langle \text{lettre} \rangle \mid \langle \text{identificateur} \rangle \langle \text{chiffre} \rangle$

devient en utilisant ce symbolisme :



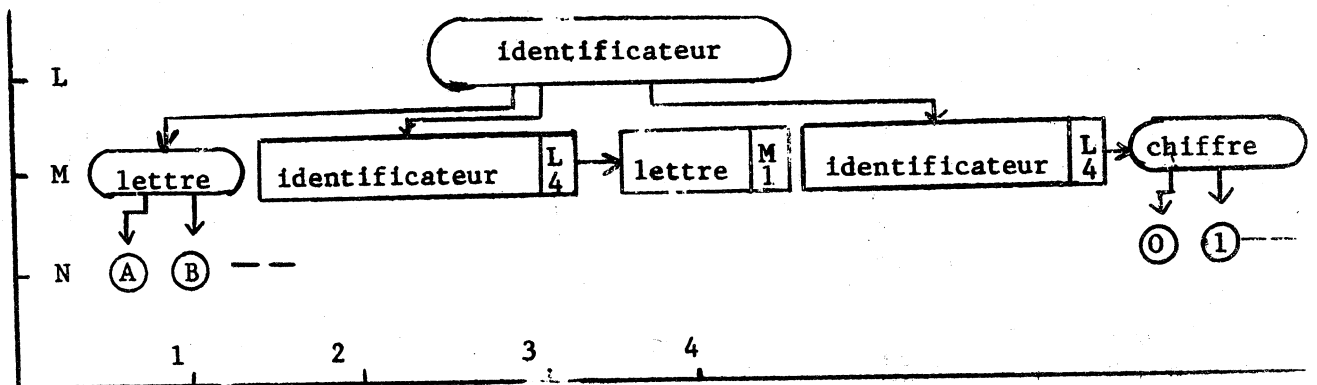
La définition de chiffre : $\langle \text{chiffre} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

devient



Le plus souvent, la carte est représentée à l'intérieur d'un quadrillage comportant des coordonnées et chaque rectangle (représentant une variable intermédiaire déjà définie) portent les coordonnées de l'ovale correspondant à la définition de cette variable intermédiaire.

Exemple :



L'intérêt d'une telle représentation graphique est qu'elle permet une utilisation synthétique de la description syntaxique. De plus elle renseigne immédiatement sur les caractéristiques syntaxiques du langage : suivant que la carte syntaxique se développe principalement dans le sens vertical ou dans le sens horizontal on est renseigné sur la concision ou la verbosité de la description syntaxique.

2) La notation bidimensionnelle de COBOL. [7]

Cette notation utilise le symbolisme et les règles suivantes :

- a) les variables métalinguistiques sont désignées par des mots formés de lettres minuscules, de chiffres et du tiret.

Exemple : nom-donnée-1

b) les constantes métalinguistiques autres que les symboles de base sont représentées par des mots écrits en lettres majuscules.

- si ces mots sont soulignés leur présence est impérative
- s'ils ne sont pas soulignés leur présence est facultative

Exemple : ON SIZE ERROR

ON est facultatif, SIZE et ERROR obligatoires.

c) l'opération de concaténation est implicitement définie par la juxtaposition des variables et des constantes métalinguistiques.

d) la paire d'accollades : { } permet d'écrire les listes d'alternatives sous forme de colonnes.

Dans une colonne d'alternatives écrites entre accolades, une alternative doit être choisie obligatoirement.

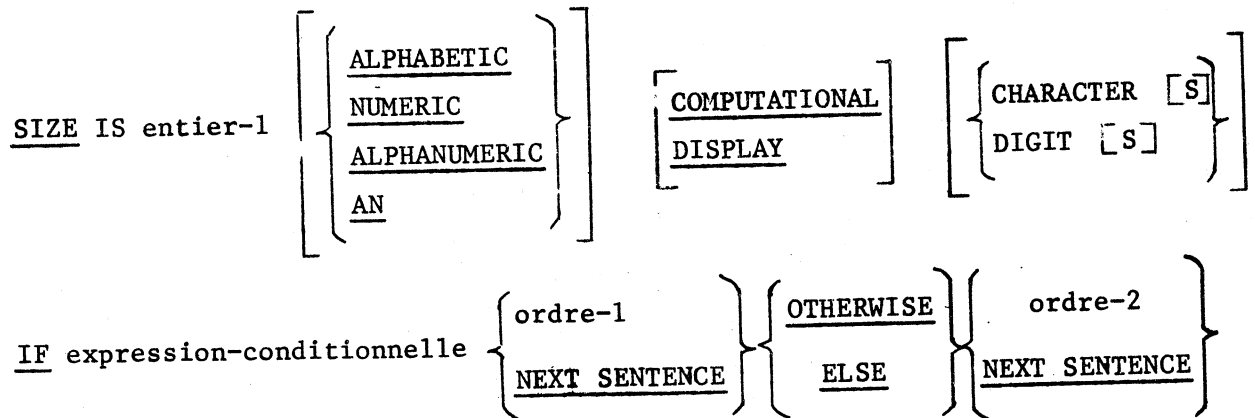
Exemple : $\left\{ \begin{array}{l} \underline{\text{LEFT}} \\ \underline{\text{RIGHT}} \end{array} \right\}$

e) la paire de crochets : [] définit une option c'est-à-dire l'occurrence facultative du contenu des crochets.

f) les 3 points ... dénotent l'occurrence répétée facultative du dernier élément écrit.

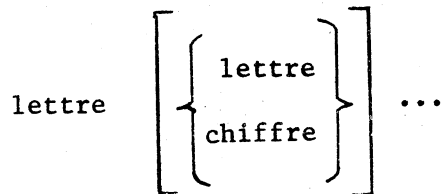
Remarquons que cette dernière notation est souvent utilisée en mathématiques dans un sens analogue pour indiquer la répétition d'éléments semblables (exemple : $\mu_1, \mu_2, \mu_3, \dots$)

Exemples de description syntaxique en notation COBOL.



Avantages et inconvénients de cette notation.

Cette notation est plus concise que celle de Backus spécialement pour les définitions récursives. Par exemple, en utilisant ce symbolisme, la définition d'un identificateur ALGOL devient :



L'écriture des alternatives ligne par ligne facilite la lecture des formats syntaxiques.

Par contre la notation est alourdie par les conventions d'écriture minuscule et majuscule et le soulignement de certains mots.

En définitive, cette notation n'apporte rien de plus que celle de Backus quant à la puissance d'expression du métalangage.

Remarque : cette notation COBOL est utilisée dans la description des systèmes moniteurs pour les formats des cartes systèmes.

Exemple : tiré de la description du système IBSYS-7044

IBJOB program-name $\left[\begin{array}{l} \{ \text{GO} \} \\ \{ \underline{\text{NO GO}} \} \end{array} \right] \left[\begin{array}{l} \{ \text{LOGIC} \\ \text{DLOGIC} \\ \{ \underline{\text{NOLIGIC}} \} \end{array} \right] \left[\begin{array}{l} \{ \text{MAP} \\ \{ \underline{\text{NOMAP}} \} \end{array} \right] \left[\begin{array}{l} \{ \text{FILES} \\ \{ \underline{\text{NOFILES}} \} \end{array} \right]$

Dans cette représentation, les mots soulignés représentent les options par défaut (si rien n'est spécifié, cette alternative sera automatiquement choisie).

3) La notation syntaxique de PL/I. [8]

a) les variables métalinguistiques sont désignées par des mots formés de lettres, de chiffres décimaux et du tiret dont le premier symbole est une lettre.

De plus, si la désignation de la variable métalinguistique comprend à la fois des lettres majuscules et minuscules, le mot doit être constitué de deux parties de mot séparées par un tiret, l'une ne comportant que des lettres majuscules, l'autre que des lettres minuscules.

Exemple : digit
 DO-statement
 file-name

b) à côté des symboles de base proprement dits, il existe des constantes métalinguistiques constituées par des mots écrits avec des lettres majuscules. Une telle constante métalinguistique dénote l'occurrence explicite de ces mots dans la définition.

Exemple : DECLARE identifieur FIXED ;

identifieur est un nom de variable métalinguistique auquel on doit substituer une certaine valeur.

DECLARE, FIXED et ";" sont des constantes métalinguistiques qui se dénotent elles-mêmes.

c) l'opération de concaténation est définie implicitement par le blanc qui sert également de séparateur entre les variables et les constantes métalinguistiques.

d) la paire d'accolades { } est utilisée pour indiquer le groupage des éléments.

Comme dans COBOL, on peut lister les alternatives ligne par ligne à l'intérieur des accolades.

e) la barre verticale peut être utilisée pour séparer les alternatives si l'on veut éviter l'écriture bidimensionnelle.

Exemple : les deux écritures :

identifieur {
 FIXED
 }
 {
 FLOAT
 }

et identifieur {FIXED | FLOAT}

sont strictement équivalentes et peuvent être utilisées indifféremment.

f) la paire de crochets [] dénote une occurrence facultative y compris l'occurrence vide.

Ainsi $[a]$ dénote soit 0, soit 1 occurrence de a

g) les trois points ... dénotent l'occurrence répétée de l'élément qui précède.

On désigne ici par élément soit une variable, soit une constante métalinguistique, soit une suite de variables et de constantes écrites entre accolades ou entre crochets.

Par exemple :

lettre $[\text{lettre} \mid \text{chiffre}] \dots$

définit l'identificateur ALGOL.

- si on utilise le même symbole à la fois comme symbole de base et comme symbole métalinguistique, la dénotation de ce symbole comme symbole de base sera soulignée.

Exemple :

opérande $\{ \varepsilon \mid \perp \mid \neg \}$ opérande

h) deux métaopérateurs spéciaux min et max sont définis avec la signification suivante :

Les symboles min et max sont suivis par des entiers qui jouent le rôle de facteur de répétition : l'élément qui suit doit avoir un minimum et un maximum d'occurrences définies par les valeurs des facteurs de répétition. Ici élément est utilisé avec le sens défini précédemment.

Exemple :

min 2 max 6 {digit | letter}

signifie que l'élément digit ou letter doit figurer au minimum deux fois et au maximum 6 fois.

Lorsque min est utilisé seul, la valeur de max est supposée infinie.
Lorsque max est utilisé seul, la valeur de min est supposée nulle.

Un identificateur ALGOL de longueur 12 ou plus serait défini de la manière suivante :

lettre [max 11 {lettre | chiffre}]

Remarque 2 : Brooker et Morris utilisent une variante de la notation de Backus avec le symbolisme suivant : [9]

- les métacrochets : <et> sont remplacés par les crochets [et] et pour désigner les symboles de base [et] on utilisera l'écriture [[et]] afin d'éviter une confusion entre symboles métalinguistiques et symboles de base.
- la barre d'énumération est remplacée par la virgule (le symbole de base ",," est désigné par [,])
- le signe ::= est remplacé par le signe =
- le métasymbole * désigne une occurrence répétée
Ainsi [B*] signifie [B] ou [B][B] ou etc.
- le métasymbole ? désigne une occurrence facultative
Ainsi [B?] désigne rien ou [B]
Ces deux métasymboles peuvent être combinés
Ainsi [B*?] désigne rien ou [B] ou [B][B] ou...

Exemple de définition des expressions arithmétiques FORTRAN.

[MUL]	= * , /
[ADD]	= + , -
[PRIMAIRE]	= [VARIABLE] , (EXPR)
[FACTEUR]	= [PRIMAIRE] [** PRIMAIRE * ?]
[TERME]	= [FACTEUR] [MUL] [FACTEUR] * ?
[EXPR]	= [ADD?] [TERME] [ADD] [TERME] * ?

Remarque : dans cette définition, le mode des expressions (fixe ou flottant) n'est pas considéré.

Autres extensions de la forme normale de Backus

1) Floyd a proposé l'usage de la paire d'accolades pour spécifier un nombre d'occurrences arbitraires de l'élément contenu à l'intérieur. [10]

Ainsi la forme :

$$\mu \rightarrow \langle x \rangle \{ \langle y \rangle \} \langle z \rangle$$

est une abréviation pour l'ensemble des formules suivantes :

$$\langle \mu \rangle \rightarrow \langle x \rangle \langle z \rangle$$

$$\langle \mu \rangle \rightarrow \langle x \rangle \langle y \rangle \langle z \rangle$$

$$\langle \mu \rangle \rightarrow \langle x \rangle \langle y \rangle \langle y \rangle \langle z \rangle$$

etc...

Si le nombre d'occurrences est $\leq n$ on pourra écrire

$$\langle x \rangle \{ \langle y \rangle \}^n \langle z \rangle$$

Par exemple la définition de la déclaration de tableau en ALGOL s'écrira avec cette notation :

$$\langle \text{décl de tab} \rangle \rightarrow \{ \underline{\text{reel}} \mid \underline{\text{entier}} \mid \underline{\text{Booléen}} \}^1 \underline{\text{tableau}} \{ \langle \text{ident} \rangle, \} \langle \text{ident} \rangle \\ [\{ \langle \text{paire de bornes} \rangle \} \langle \text{paire de bornes} \rangle]$$

L'intérêt d'une telle notation est qu'elle permet de réduire considérablement le nombre de variables intermédiaires.

2) Ershov a proposé les extensions suivantes qui permettent d'augmenter les possibilités de description syntaxique. [11]

- pour exprimer qu'une variable métalinguistique $\langle \lambda \rangle$ par exemple peut prendre une valeur arbitraire mais identique pour toutes ses occurrences dans la partie droite d'une définition, on utilise la notation $\} \lambda \{$

Par exemple :

$\langle x \rangle ::= \} \text{identificateur} \{ \} \text{identificateur} \{$

exprime que $\langle x \rangle$ est définie par la concaténation de deux identificateurs identiques mais dont la désignation est quelconque

De même :

$\langle y \rangle ::= \} \text{identificateur} \{ \langle \text{identificateur} \rangle \} \text{identificateur} \{$

exprime que $\langle y \rangle$ est défini par la concaténation de trois identificateurs, le premier et le troisième devant être identiques.

- pour permettre la substitution de variables métalinguistiques, on définit l'opération suivante :

substitution de $\langle z \rangle$ dans $\langle y \rangle$ pour $\langle x \rangle$

dans laquelle $\langle x \rangle$, $\langle y \rangle$ et $\langle z \rangle$ représentent des variables métalinguistiques.

Le résultat de cette opération est défini de la façon suivante :

Soient X,Y et Z les valeurs des variables métalinguistiques $\langle x \rangle$, $\langle y \rangle$ et $\langle z \rangle$. Dans le mot Y, toutes les occurrences de X doivent être remplacées par Z.

Exemples d'utilisation.

a) Déclaration d'une variable indicée complexe :

<déclaration de variable indicée complexe> ::=
 <ident de tab> [] liste d'ind. { } = <ident de tab> [] liste d'ind }
 + i <ident de tab> [] liste d'ind }

La liste d'indices doit être identique dans les 3 occurrences.

Exemple :

$$a \ [i,j] = b \ [i,j] + i \ c \ [i,j]$$

b) Définitions de listes abrégées :

Exemple de la liste abrégée de variables

<liste abrégée de variable> ::=
 substit de <expr arith> dans }variable { pour }expr d'indice{,...,
 substit de <expr arith> dans }variable { pour }expr d'indice{

En effet, si la liste de variables est une liste de variables indicées, le nom de la variable et les indices constants doivent rester identiques.

Exemple : a [1,2,1],..., a [1,2,n]

qui désigne la liste de variables a [1,2,k] pour k variant de 1 à n

3) Iverson a proposé quelques conventions qui permettent une écriture plus compacte des formules métalinguistiques. [12]

a - les définitions syntaxiques sont numérotées séquentiellement et on utilise le numéro des définitions à la place de leur nom.

b - l'astérisque désigne la relation effectivement définie, les définitions récursives sont donc caractérisées par la présence d'un astérisque.

c - les synonymes ne sont pas répétés.

d - la paire d'accolades sert à désigner les éléments d'un ensemble.

(Ainsi identificateur de variable, variable simple, identificateur de tableau, identificateur de procédure, identificateur d'aiguillage et paramètre formel sont synonymes d'identificateur ; de même expression d'indice, borne supérieure et borne inférieure sont synonymes d'expression arithmétique, instruction procédure et indicateur de fonction sont synonymes).

Exemples :

21	primaire	2		39		36		(25)									
22	facteur	21		*	↑	21											
23	terme	22		*	{	×		/		÷	}	22					
24	expr arith simple	23		{	+		-	}	23		*	{	+		-	}	23
25	expr arith	24		26	24	<u>sinon</u>	*										
26	proposition si															

Conclusion générale.

Toutes les extensions apportées au formalisme initial de Backus n'ont pas sensiblement accru les possibilités de description syntaxique des langages. Si l'on peut dire que la formalisation de la syntaxe est maintenant un fait acquis et représente un progrès considérable sur les modes de description utilisés antérieurement, on ne peut s'empêcher de remarquer qu'elle ne représente qu'une partie assez modeste dans la description complète d'un langage de programmation. Néanmoins, dans la plupart des techniques actuelles de compilation, elle représente le noyau autour duquel s'élabore le schéma d'analyse et de traduction du langage décrit.

C H A P I T R E III

NOTATIONS SEMANTIQUESI - LA NOTATION UTILISEE DANS LE RAPPORT ALGOL.

Alors que la description syntaxique définit les règles d'écriture des programmes, la description sémantique définit les règles d'interprétation et d'évaluation.

Dans le rapport ALGOL, la description syntaxique est complètement formalisée et cette formalisation a permis le développement de techniques perfectionnées d'analyse et de vérification syntaxiques.

Par exemple, l'utilisation des méthodes de précedence permet, étant donné la description syntaxique formalisée d'un langage, de détecter systématiquement les ambiguïtés syntaxiques. Une fois ces ambiguïtés détectées, et localisées, il est aisé de choisir l'interprétation la plus logique et de modifier la description syntaxique.

La description sémantique par contre semble assez loin d'une formalisation acceptable et cela tient à la difficulté même de définir exactement ce qu'on entend par description sémantique d'un langage.

Prenons par exemple le cas des expressions : qu'est ce que la description sémantique des expressions ? C'est l'ensemble des règles qui permettent l'évaluation des expressions. Dans le rapport ALGOL, ces règles sont données au moyen d'un texte explicatif précisant le détail d'exécution des opérations, l'évaluation des primaires, les priorités d'opérateurs. A la fin du texte il est indiqué qu'aucune arithmétique exacte n'est spécifiée et que les

différentes représentations machines peuvent évaluer les expressions arithmétiques de manières différentes.

Alors que les règles de description syntaxique permettent effectivement de préciser si un programme est syntaxiquement correct ou non, il semble que la question de préciser si un programme est sémantiquement correct n'ait pratiquement aucun sens.

Cela tient à ce que la signification d'un programme dépend à la fois du texte du programme lui-même, des données qu'il utilise et d'un mécanisme d'exécution de ce programme.

Utilisation d'un langage minimum dans la description sémantique.

Dans le rapport ALGOL, la sémantique de l'élément de liste de pour de la forme "A pas B jusqu'à C" est définie sous la forme d'un programme ALGOL.

L'effet de l'instruction :

pour V := A pas B jusqu'à C faire INSTRUCTION ;

est décrit par le programme ALGOL suivant :

```
V:=A ;  
L1:si (V-C) × signe (B) < 0 alors allera fin ;  
INSTRUCTION ;  
V:=V+B ;  
allera L1 ;  
fin ;
```

Si cette séquence décrit réellement le mécanisme d'exécution de cette instruction tel qu'il doit être traduit par un compilateur, on peut remarquer que cette traduction est extrêmement coûteuse.

A chaque variation du pas, il faut évaluer :

- 3 fois la variable contrôlée V
- 2 fois le pas B
- 1 fois la valeur finale C

Dans le cas général, la variable contrôlée V peut être une variable indicée comportant des expressions en indice qui devront être évaluées chaque fois, il en est de même pour les expressions B et C qui peuvent contenir des variables indicées et des indicateurs de fonctions.

- si l'on choisit l'interprétation statique de l'instruction pour, alors V, B et C ne sont évalués qu'une seule fois au début de l'exécution de l'instruction. La séquence est la suivante :

```
V := A ;  
V1:= V ;  
V2:= B ;  
V3:= C ;
```

```
L1 : si (V1 - V3) × signe (V2) < 0 alors allera fin ;
```

```
INSTRUCTION ;  
V1:= V1 + V2 ;  
allera L1 ;
```

```
fin :
```

- Si l'on choisit l'interprétation dynamique, on peut tout de même économiser la double évaluation de B à chaque pas et écrire :

```
V := A ;
V2 := B ;
L1 : si (V-C) * signe (V2) < 0 alors allera fin ;
INSTRUCTION ;
V2 := B ;
V := V + V2 ;
allera L1 ;
fin ;
```

Au lieu d'utiliser comme dans le cas précédent les concepts d'instruction d'affectation, d'instruction conditionnelle et d'instruction allera pour la description sémantique de l'instruction pour, on aurait pu utiliser, d'autres concepts, par exemple celui de procédure.

- en choisissant l'interprétation statique on aurait alors la séquence :

```
procédure POUR (V, A, B, C, INSTRUCTION) ; valeur A, B, C ;
réel V, A, B, C ; procédure INSTRUCTION ;
début réel V 1 ;
    V := A ;
    V1 := V ;
    L1 : si (V1 - V3) * signe (V2) < 0 alors
        début
            INSTRUCTION ;
            V1 := V1 + V2 ;
            allera L1 ;
        fin
fin ;
```

Si l'on compare cette séquence et la séquence correspondante n'utilisant pas le concept de procédure, on constate que l'utilisation de ce concept procédure apporte en fait des restrictions supplémentaires (en particulier pour INSTRUCTION).

Ce mécanisme d'abréviation peut s'utiliser pour de nombreuses constructions d'un langage.

Par exemple on peut exprimer l'effet de l'instruction conditionnelle complète au moyen d'instructions si et allera

si EB alors I1 sinon I2 ; I3

est équivalent à

si EB alors allera E1 ;

I1 ;

allera E2 ;

E1 : I2 ;

E2 : I3 ;

Ces équivalences sont très utiles pour l'écriture des compilateurs, le langage minimum étant dans ce cas un langage machine ou un langage intermédiaire.

On constate ainsi que si l'on veut décrire la sémantique d'un langage en utilisant des concepts plus simples de langage, il est indispensable d'avoir la description sémantique complète de tous les concepts de base utilisés.

II - DESCRIPTION SEMANTIQUE ET MECANISME D'EVALUATION.

La signification d'un programme dépend à la fois du texte de ce programme et des données associées et ne peut être valablement définie que par la description d'un mécanisme d'exécution de ce programme.

La description sémantique consiste donc essentiellement à considérer un programme comme une donnée appelée texte utilisant d'autres données dont certaines peuvent être des textes (sous-programmes) et à définir un mécanisme d'évaluation de ces données.

Ainsi par exemple, définir la sémantique du texte $24 + 12$ c'est définir l'algorithme permettant d'additionner ces deux nombres et de trouver la valeur 36. Alors la valeur sémantique de $24 + 12$ est 36 relativement à ce mécanisme précis d'évaluation.

Toutes les descriptions sémantiques utilisent une machine abstraite permettant l'évaluation des programmes, l'action de cette machine étant définie pour chacun des éléments du programme.

On voit qu'il y a une grande analogie entre la sémantique d'un langage machine définie par les réactions de la machine pour chaque opération élémentaire et la sémantique d'un langage de programmation définie par les réactions d'une machine abstraite pour chaque élément de base de ce langage.

III - ETUDE DE QUELQUES MECANISMES DE DESCRIPTION SEMANTIQUE.

A) La notation de Church [13]

1) Le concept de fonction.

Une fonction est une règle de correspondance qui permet par la donnée d'un ou plusieurs arguments d'obtenir une valeur de cette fonction. La fonction est ainsi analogue à une opération qui permet à partir d'un ou plusieurs opérands d'obtenir un résultat d'opération.

Pour chaque fonction, il y a un domaine de définition de l'argument, (ou domaine de définition de la variable indépendante) et le domaine des valeurs prises par la fonction pour toutes les valeurs possibles de l'argument (ou domaine de définition de la variable dépendante).

Les domaines de définition de l'argument ou de la fonction peuvent comporter des fonctions.

Par exemple, l'opération de dérivation d'une fonction, a comme argument une fonction et produit comme résultat une autre fonction.

L'intégrale définie d'une fonction $f(x)$ dans l'intervalle $(0,1)$ à comme argument une fonction et produit un nombre.

En particulier, un des éléments du domaine de définition d'une fonction f peut être la fonction f elle-même : on ne peut plus alors définir une fonction comme une règle de correspondance entre deux domaines de définition préalablement donnés.

Selon Church, l'opération ou règle de correspondance, qui constitue la fonction est préalablement donnée, le domaine des arguments est alors déterminé comme l'ensemble des objets à laquelle l'opération est applicable.

Une fonction f d'argument α sera notée $(f\alpha)$: cette notation n'a pas de sens si α n'appartient pas au domaine des arguments de f .

2) Fonctions de plusieurs variables.

Une fonction de deux variables est regardée comme une fonction de une variable dont les valeurs sont des fonctions d'une variable : une fonction de trois variables est regardée comme une fonction d'une variable dont les valeurs sont des fonctions de deux variables.

Ce procédé permet de définir une fonction de n variables à l'aide des fonctions de une variable.

Si f désigne une fonction de 2 variables, $((fa)b)$ représente la valeur de f pour les arguments a et b , cette notation peut être simplifiée en $(f ab)$ ou $f ab$.

Si f désigne une fonction de 3 variables, $((((fa)b)c)$ pourra s'écrire $f abc$, ou fa désigne une certaine fonction de 2 variables et $f ab$ une fonction de 1 variable.

3) Abstractions fonctionnelles.

Il est important de distinguer soigneusement entre un symbole ou une expression qui désigne une fonction et une expression contenant une variable et pouvant désigner une valeur de la fonction.

Par exemple, considérons l'expression $(x^2 + x)^2$

Lorsque nous disons que $(x^2 + x)^2$ est plus grand que 1000, cette assertion n'a de sens que si nous sous-entendons que x représente un nombre particulier.

Si par contre, nous disons que $(x^2 + x)^2$ est une fonction primitive récursive, notre assertion ne dépend pas de la valeur de x .

La différence entre les 2 exemples réside dans le fait que dans le premier cas l'expression $(x^2 + x)^2$ sert à désigner un nombre entier, et cette désignation est ambigüe, dans le second cas, $(x^2 + x)^2$ sert à désigner une fonction particulière.

Dans le premier cas, nous utiliserons la notation $(x^2 + x)^2$ pour désigner le nombre, dans le second cas nous utiliserons la notation

$(\lambda x (x^2 + x)^2)$ pour désigner la fonction

Remarque : Dans l'expression $(x^2 + x)^2$, une ambiguïté peut subsister sur la signification du signe +, suivant que l'expression est appliquée à des nombres entiers, réels ou complexes. Dans une notation correcte, les trois types d'addition devraient être désignés par des symboles différents par exemple : $+_e$, $+_r$, $+_i$. On devrait de même distinguer entre le carré d'un nombre entier, réel ou complexe.

Soit M une expression contenant une variable x

$(\lambda x M)$ désigne une fonction obtenue à partir de l'expression M par abstraction, le symbole λx est l'opérateur d'abstraction ; ce symbole n'a aucune signification en lui-même.

L'expression :

$(\lambda x (\lambda y M))$ désigne une fonction dont la valeur pour un argument est désignée par $(\lambda y M)$

Cette notation peut être abrégée en $(\lambda x y . M)$

L'expression $(\lambda y (\lambda x M))$ abrégée en $(\lambda y x . M)$ désigne la fonction réciproque de celle désignée par $(\lambda x y . M)$

La fonction de 3 variables $(\lambda x (\lambda y (\lambda z M)))$ peut être écrite en abrégé $(\lambda x y z . M)$.

4) Règles d'écritures.

Les abstractions fonctionnelles peuvent se représenter par des expressions dont la syntaxe est définie par les règles suivantes :

Les symboles de base sont :

- $\lambda ()$
- les lettres qui définissent les variables.

Un mot est une suite finie de symboles de base. Parmi les mots, on peut distinguer des expressions et dans chaque expression, chaque occurrence d'une variable peut être libre ou liée.

Les expressions et les occurrences des variables dans une expression sont définies récursivement :

I - Une variable x est une expression et l'occurrence de la variable x dans cette expression est libre

II - Si F et A sont des expressions, alors (FA) est une expression et l'occurrence d'une variable y dans F est libre ou liée dans (FA) suivant qu'elle est libre ou liée dans F et l'occurrence d'une variable y dans A est libre ou liée dans (FA) suivant qu'elle est libre ou liée dans A .

III - Si M est une expression qui contient au moins une occurrence libre de x , alors $(\lambda x M)$ est une expression et l'occurrence d'une variable y , autre que x , dans $(\lambda x M)$ est libre ou liée dans $(\lambda x M)$ suivant qu'elle est libre ou liée dans M . Toutes les occurrences de x dans $(\lambda x M)$ sont liées.

IV - Un mot est une expression et une occurrence d'une variable dans cette expression est libre ou liée dans les seuls cas qui résultent de l'application des règles I à III.

Nous utiliserons les abréviations suivantes :

$(\lambda x. FA)$ pour $(\lambda x (FA))$

$(\lambda x y. FA)$ pour $(\lambda x (\lambda y (FA)))$

$(\lambda x y z. FA)$ pour $(\lambda x (\lambda y (\lambda z (FA))))$

.....

La suite : $\lambda x y \dots$ sera désignées par partie λ

Toute expression est constituée d'une partie λ suivie d'une expression au sens ordinaire.

Nous pouvons également omettre les parenthèses dans (FA) : pour rétablir les parenthèses, on utilise la règle d'association de gauche à droite. aussi $f x y$ est une abréviation de $((f x) y)$
De la même façon nous pouvons supprimer les parenthèses dans les expressions toutes les fois que la règle d'association de gauche à droite permet de la faire sans ambiguïté.

Ainsi $(\lambda x (\lambda y (\lambda z [[x+y] + z])))$ s'écrira
 $\lambda x y z. x + y + z$

Remarque : Dans ce dernier exemple, [et] désignent les parenthèses d'expression.

Intérêt de la notation de CHURCH.

La notation de CHURCH permet de distinguer entre formes et fonctions

Par exemple $x^2 + y$ est une forme

$\lambda x y. x^2 + y$ est une fonction

L'expression $\lambda u v. u^2 + v$ désigne la même fonction que

$\lambda x y. x^2 + y$

Toute forme peut être transformée en fonction par adjonction d'une partie qui détermine la correspondance entre les variables de la forme et les arguments de la fonction.

Considérons l'intégrale définie

$$\int_0^x x^2 dx$$

Si ϕ représente l'opération d'intégration on peut écrire
 $\phi (0, x, \lambda x. x^2)$ qui exprime que dans l'expression x^2 , x est une variable liée.

De même

$$\sum_{j=1}^n a_{ij} b_j \text{ peut s'écrire } \Sigma (1, n, \lambda j. a(i, j) \times b(j, k))$$

Remarque : En ALGOL, l'appel par nom ne permet pas de transmettre la définition d'une fonction mais seulement une relation fonctionnelle (expression) au moyen du mécanisme de Jensen ; pour transmettre une définition il faudrait pouvoir écrire un bloc à la place d'un paramètre effectif (mécanisme proposé par Samuelson [14]).

Valeurs d'une fonction.

Si l'on veut définir la valeur d'une fonction, on fera suivre la définition de la fonction de la liste des arguments, en suivant l'ordre défini dans la partie λ .

La valeur de $\lambda xy. x^2 + y$ pour $x = 2$ et $y = 3$ s'écrit
 $\lambda xy. x^2 + y (2, 3)$ et donne $2^2 + 3 = 7$
 par contre $\lambda yx. x^2 + y (2, 3)$ donne $3^2 + 2 = 11$
 $\lambda yx. x^2 + y (3, 2)$ donne $2^2 + 3 = 7$

Le langage LISP et la notation de CHURCH.

Le langage LISP utilise la notation de CHURCH pour la définition des fonctions et des fonctionnelles ; les fonctionnelles sont des fonctions qui ont pour arguments des fonctions. La notation de CHURCH est insuffisante pour manipuler le cas particulier des fonctions définies récursivement.

Considérons la définition de la factorielle en notation ALGOL :
entier procédure $f(n)$; valeur n ; entier n ;
 $f(n) :=$ si $n = 0$ alors 1 sinon $n \times f(n-1)$;

La définition de cette fonction pourrait s'écrire en notation de CHURCH $f = \lambda n. \text{ si } n = 0 \text{ alors } 1 \text{ sinon } n \times f (n-1)$

La difficulté réside dans le fait que rien dans cette expression ne permet de préciser que f désigne précisément la fonction définie.

Afin de pouvoir utiliser le nom de la fonction comme une variable liée, LISP définit une convention supplémentaire :

Si E est une expression et e son nom, on écrira Label (e, E) pour exprimer que e est liée dans E

La définition de la factorielle devient alors :

Label ($f, \lambda n. \text{ si } n=0 \text{ alors } 1 \text{ sinon } n \times f (n-1)$)

B) Description syntaxique et sémantique du langage Micro-Algol [15]

a) Syntaxe analytique.

La forme normale de Backus décrit la syntaxe d'ALGOL sous forme synthétique. Elle permet en effet de construire des programmes en utilisant les règles de génération. Ainsi cette description est mieux adaptée à la traduction d'un programme en langage ALGOL qu'à la traduction d'un programme ALGOL en un autre programme.

La syntaxe analytique de Mc Carthy a les propriétés suivantes :

- elle décrit la façon de décomposer un programme (analytique plutôt que synthétique).
- elle est abstraite c'est-à-dire indépendante de la notation utilisée, et elle définit uniquement comment des parties peuvent être reconnues et décomposées.

Considérons le cas d'une expression ALGOL ne contenant que des constantes, des variables, des sommes et des produits.

Soit t un terme. Il est nécessaire d'abord d'identifier ce terme t afin de savoir s'il représente une constante, une variable, une somme ou un produit. Nous devons donc admettre l'existence de 4 prédicats (fonctions booléennes):

$pconst (t)$; $pvar (t)$; $psom (t)$; $pprod (t)$

A chaque terme doit correspondre un prédicat qui prend la valeur vrai et un seul.

Considérons le cas des variables pour lesquelles $pvar (t) = \text{vrai}$; le compilateur doit déterminer si 2 symboles sont 2 occurrences de la même variable : cela signifie que l'on doit définir un prédicat d'égalité sur les variables. Si les variables doivent être utilisées dans un certain ordre (cas de la structure de blocs par exemple) un prédicat définissant la relation d'ordre est nécessaire.

Pour les sommes, le compilateur doit pouvoir séparer les 2 opérandes à gauche et à droite pour lesquels $psom (t) = \text{vrai}$; même chose pour le produit avec multiplicateur et multiplicande : m droit et m gauche.

$$\begin{aligned} \text{terme } (t) &= pconst (t) \vee pvar (t) \vee \\ &psom (t) \wedge \text{terme } (agauche (t)) \wedge \text{terme } (adroit (t)) \vee \\ &pprod (t) \wedge \text{terme } (mdroit (t)) \wedge \text{terme } (mgauche (t)) \end{aligned}$$

Cette définition est l'analogue en ALGOL d'une procédure booléenne définissant une fonction syntaxique. (cf. 2e Partie, chapitre III)

L'opérateur \wedge joue le rôle d'opérateur de concaténation, l'opérateur \vee (ou) permettant la sélection d'un des 4 prédicats qui définissent un terme.

b) Définition du langage Micro-Algol.

C'est un langage très simple où les seules instructions sont l'instruction d'affectation et l'instruction allera ; dans les expressions on peut utiliser l'addition, la soustraction, la multiplication, la division et des expressions conditionnelles utilisant les opérateurs = et >

Exemple : Algorithme de la racine carrée de a.

```

1   racine := 1 ;
2   e: racine := 0.5 × (racine + a/ racine) ;
3   erreur := racine × racine - a ;
4   perreur := si erreur > 0 alors erreur sinon 0 - erreur ;
5   si perreur > 0.00001 alors allera e ;

```

Syntaxe des termes.

A chaque terme on peut associer l'un des 8 prédicats suivants :
 var (t), som (t), diff (t), prod (t)
 quot (t), cond (t), égal (t), super (t)

A chacun de ces prédicats, on peut associer des fonctions syntaxiques qui permettent la décomposition d'une expression.

Pour les 8 prédicats ci-dessus, les fonctions associées sont :

prédicat

fonctions syntaxiques associées

1) som (t)	sgauche (t)	sdroit (t)
2) diff (t)	dgauche (t)	ddroit (t)
3) prod (t)	multiplicande (t)	multiplicateur (t)
4) quot (t)	numérateur (t)	dénominateur (t)
5) cond (t)	proposition (t)	antécédent (t) conséquent (t)
6) égal (t)	égalgauche (t)	égaldroit (t)
7) super (t)	supergauche (t)	superdroit (t)

Exemples :

Si t est égal à "racine + x/racine"
 alors som (t) est vrai avec sgauche (t) = racine et sdroit (t) = x/racine

Si t est égal à :

Si erreur > 0 alors erreur sinon 0 - erreur

cond (t) est vrai avec proposition (t) = erreur > 0

antécédent (t) = erreur

conséquent (t) = 0 - erreur

Syntaxe des instructions.

A chacune des instructions est affecté un prédicat
 affectation (i), allera (i)

1) Au prédicat affectation (i), on associe les deux fonctions
 gauche (i) et droite (i)

2) et au prédicat allera (i), les deux fonctions :
 proposition (i) et destination (i)

Exemples :

si $i = \text{"erreur := racine} \times \text{racine} - x\text{"}$ alors affectation (i) est vrai
 avec "gauche (i) = erreur" et "droite (i) = racine \times racine -x"
 si $i = \text{"si } p \text{ erreur} > 0.00001 \text{ alors } \underline{\text{allera } a}\text{"}$ alors allera (i) est vrai
 avec "proposition (i) = p erreur $>$ 0.00001" et "destination (i) = a"

Syntaxe des programmes.

1) Si π est un programme et n un numéro d'instruction alors instruction (π , n) définit la $n^{\text{ème}}$ instruction du programme π .

Dans l'algorithme de la racine carrée ci-dessus
 Instruction (π , 3) = "erreur := racine \times racine -a"

2) Si e est une étiquette alors numéro (e , π) est le numéro de l'instruction dont l'étiquette est e

Pour l'exemple ci-dessus : numéro (a , π) = 2

3) Le prédicat fin (π , n) est vrai s'il n'existe pas d'instruction de numéro n .

Dans l'exemple ci-dessus fin (π , 6) = vrai

Le vecteur d'état d'un programme.

L'exécution d'un programme est essentiellement caractérisée par un vecteur d'état V qui définit les valeurs de chacune des variables du programme et le numéro de l'instruction à exécuter.

Au vecteur d'état, sont **associées** deux fonctions :

- $c(\text{var}, \xi)$ donne la valeur de la variable désignée par var dans l'état ξ

- a (var, valeur, ξ) définit le nouvel état qui résulte de l'état ξ lorsque la quantité désignée par valeur est affectée à la variable désignée par var.

On suppose que les valeurs des variables sont des nombres entiers.

La variable dont la valeur définit le numéro de l'instruction à exécuter sera désignée par i

Sémantique de micro-Algol.

La sémantique d'un programme écrit en micro-Algol est définie au moyen d'une fonction récursive.

micro (π , ξ)

qui détermine l'état final d'un programme π dont l'état initial est ξ

- au prédicat $\text{const}(t)$ qui prend la valeur vrai si t est une constante est associée la fonction sémantique val (t)

- la valeur d'un t dans l'état ξ est donnée par la fonction sémantique suivante :

valeur (t , ξ) =

si var (t) alors $c(t, \xi)$

sinon si const (t) alors val (t)

sinon si som (t) alors valeur (s gauche (t), ξ) + valeur (s droit (t), ξ)

sinon si diff (t) alors valeur (dgauche (t), ξ) - valeur (ddroit (t), ξ)

sinon si prod (t) alors valeur (multiplicande (t), ξ)

× valeur (multiplicateur (t), ξ)

sinon si quotient(t) alors valeur (numérateur(t), ξ)/valeur (dénominateur(t), ξ)

sinon si cond (t) alors (si valeur (proposition (t), ξ) alors

valeur (antécédent (t), ξ) sinon valeur (conséquent (t), ξ)

sinon si égal (t) alors (valeur (égal gauche (t), ξ) = valeur
(égal droit (t), ξ))

sinon si super (t) alors (valeur (super gauche (t), ξ)
valeur (super droit (t), ξ))

La fonction sémantique micro (π , ξ) qui détermine l'état final d'un programme π à partir de l'état initial est définie par :

micro (π , ξ) = λn . si fin (π , n) alors ξ
sinon (λs . si affectation (s) alors
micro (π , a (i, n+1, a(gauche (s), valeur(droit (s), ξ)) ξ)))
sinon si allera (s) alors :
micro (π , a (i, si valeur (proposition (s), ξ) alors numéro (destination
(s) sinon n + 1, ξ))) (instruction (n, π), c (i, ξ))

Dans l'exemple de l'algorithme de la racine carrée, le vecteur d'état est défini au départ par les valeurs initiales des variables du programme :

racine, carreur, perreur,

Certaines de ces valeurs peuvent être indéterminées dans l'état initial.

C) Mécanisme d'évaluation des programmes et λ -expressions selon LANDIN

[16]

a) La structure opérateur-opérande.

Une fonction peut jouer soit le rôle d'un opérateur soit le rôle d'un opérande.

Exemple 1 : Soit à calculer la valeur de l'expression

$$f(a+b, a-b) + f(a-b, a+b)$$

Pour $a = 33$ et $b = 44$

$f(u, v)$ étant définie par $uv(u+v)$

Utilisant la notation de Church on écrira

$$\lambda a b f . f(a+b, a-b) + f(a-b, a+b) \quad (33, 44, \lambda uv . uv(u+v))$$

opérateur opérande

Dans la liste des opérands le 3e est une λ -expression

Exemple 2 : $f(g(a)) + e(f(b))$

où $f(z) = z^2 + 1$ et $g(z) = z^2 - 1$ s'écrira

$$\lambda fg . f(g(a)) + g(f(b)) \quad (\lambda z . z^2 + 1, \lambda z . z^2 - 1)$$

opérateur opérande

Exemple 3 :

$$u/(u+5) \quad \text{avec } u = a(a+1) \quad \text{et } a = 7-3$$

$$\lambda u . u/(u+5) \quad (\lambda a/a(a+1) \quad (7-3))$$

opérateur opérateur opérande

opérande

Exemple 4 :

$$f(3) + f(4) \quad \text{avec } f(x) = ax(a+x) \quad \text{et } a = 7-3$$

$$\lambda f . f(3) + f(4) \quad (\lambda a . \lambda x . ax(a+x) \quad (7-3))$$

opérateur opérateur opérande

opérande

L'opérande $\lambda a . \lambda x . ax (a+x) (7-3)$ contient une autre λ -expression

" $\lambda x . ax (a+x)$ "

cela signifie que cet opérande produit une fonction :

$$\lambda x . 4x(4+x)$$

L'exemple 4 s'interprète alors comme :

$$\lambda f.f(3)+f(4) (\lambda a . \lambda x . ax(a+x) (7-3)) \Rightarrow \lambda f.f(3)+f(4) (\lambda x.4x (4+x))$$

$$\rightarrow \lambda x.4x(4+x)(3) + \lambda x.4x(4+x) (4) \Rightarrow 12 \times 7 + 16 \times 8 \Rightarrow 212$$

b) Les diverses formes d'expressions.

Les concepts impératifs correspondent aux instructions, à l'ordre séquentiel d'exécution des instructions et aux ruptures de l'ordre séquentiel d'exécution.

Les concepts descriptifs correspondent aux expressions, déclarations structure de blocs, fonctions, règles de correspondance entre définition et appel des fonctions.

Tous les concepts descriptifs peuvent être rendus en terme de λ -expressions ; cela signifie qu'une λ -expression peut exprimer plus que le concept usuel d'expression.

Un langage de programmation comme ALGOL utilise, à côté des expressions ordinaires, des concepts qui se rattachent plus ou moins directement aux expressions, par exemple :

Listes, expressions conditionnelles, définitions récursives.

Ces concepts peuvent être exprimés en termes de λ -expressions.

1) Listes : certains opérandes sont des listes. Dans l'exemple 1 ci-dessus, l'opérande est une liste de 3 éléments (dont le dernier est une λ -expression).

Définition :

Une liste est nulle, ou bien est composée d'une tête $H(\ell)$ et d'une queue $T(\ell)$ qui est une liste.

Par exemple si $\ell = a_1, a_2, a_3 \dots \dots \dots, a_k$

alors

$H(\ell) = a_1$ $T(\ell) = a_2, a_3 \dots \dots \dots, a_k$

La sélection d'un élément peut toujours s'exprimer en termes de H et T

Par exemple, le 3^{ème} élément de la liste ci-dessus est défini par

$$H(T(T(\ell)))$$

A l'aide de H et T, on peut définir d'autres fonctions, par exemple les fonctions :

$$\begin{aligned} 1^{\text{er}}(\ell) &= H(\ell) \\ 2^{\text{e}}(\ell) &= H(T\ell) \\ 3^{\text{e}}(\ell) &= H(T(T(\ell))) \text{ etc} \dots \dots \end{aligned}$$

On peut alors définir les éléments de la liste comme résultat de l'application d'une des fonctions ci-dessus.

Par exemple, le 7^{ème} terme d'une liste ℓ est le résultat de l'application de la fonction 7^e à la liste ℓ .

Fonctions préfixe et construction.

La fonction préfixe (x). A tout élément x, on peut appliquer une fonction qui transforme une liste quelconque en une liste ayant un élément supplémentaire "x" en tête de cette liste.

Cette fonction est préfixe (x)

Exemple : préfixe (y) (L)

opérateur opérande

a pour résultat d'écrire y en tête de L

Désignons par construction (x, l) la fonction qui construit la liste formée de l'élément x suivie de la liste l.

Préfixe (x) est une fonction qui produit une autre fonction appelée construction (x, l) et telle que :

$$\text{Préfixe (x)} = \lambda l. \text{ construction (x, l)}$$

L'exemple précédent devient alors :

$$\text{Préfixe (y) (l)} = \text{construction (y, l)}$$

La fonction préfixe (x) permet alors une écriture systématique des listes et expressions ordinaires.

Exemples :

f (a,b,c,) s'écrit :

$$f(\text{préfixe (a) (préfixe (b) (\text{préfixe (c) liste nulle })))$$

on conviendra d'écrire = a + b comme

$$+ (\text{préfixe (a) préfixe (b) liste nulle })))$$

Si nous utilisons le symbolisme

$$x : L \quad \text{pour préfixe (x) (L)}$$

alors

$$x, y, z \text{ s'écrit } x : (y, z) = x : (y : (z : ()))$$

Considérons l'exemple

$$2^e (2^e (L, x : M, N)) = 1^{er} (M)$$

Le second élément de la liste est lui-même une liste : un contient comme éléments d'autres listes est une structure de liste.

2) Expressions conditionnelles :

Soit l'expression :

$$\underline{\text{si}} \ a < b \ \underline{\text{alors}} \ c \ \underline{\text{sinon}} \ d$$

Considérons "si" comme le nom d'une fonction si (α) qui produit elle-même une autre fonction :

$$\underline{\text{si}} \text{ (vrai)} = 1^{\text{er}} \quad \underline{\text{si}} \text{ (faux)} = 2^{\text{e}}$$

l'expression ci-dessus peut s'écrire

$$\begin{aligned} & \underline{\text{Si}} \text{ (a<b)} \text{ (c,d)} \text{ avec les propriétés suivantes :} \\ \text{si } a < b &= \underline{\text{vrai}} \text{ alors } \underline{\text{si}} \text{ (a<b)} \text{ (c,d)} \text{ est équivalent à } 1^{\text{er}} \text{ (c,d)} = c \\ \text{si } a < b &= \underline{\text{faux}} \text{ alors } \underline{\text{si}} \text{ (a<b)} \text{ (c,d)} \text{ est équivalent à } 2^{\text{e}} \text{ (c,d)} = d \end{aligned}$$

L'intérêt d'écrire les expressions conditionnelles sous cette forme réside dans le fait que cette forme permet l'évaluation séquentielle de l'expression.

Considérons l'expression particulière :

si $a = 0$ alors 1 sinon $1/a$ qui s'écrit :

si ($a=0$) (1, $1/a$) ou encore :

si ($a=0$) (préfixe (1) (préfixe ($1/a$) ()))

L'évaluation de cette expression est fonction de $1/a$ et n'est pas possible si $a=0$. Soit l'expression :

$$\underline{\text{si}} \text{ (a=0)} (\lambda x.1, \lambda x. 1/a) (3)$$

où x et 3 sont arbitraires, cette expression est équivalente à la précédente mais $\frac{1}{a}$ a été remplacé par une λ -expression qui définit une fonction. On peut encore écrire

$$\underline{\text{si}} \text{ (a=0)} (\lambda (). 1, \lambda (). 1/a) ()$$

Toutes les expressions conditionnelles seront écrites de cette façon

Ainsi :

si $a < b$ alors c sinon d devient :

$$\underline{\text{si}} \text{ (a<b)} (\lambda (). c, \lambda (). d) ()$$

3) Définitions récursives :

La définition de la fonction

$$f = \text{si } n = 0 \text{ alors } 1 \text{ sinon } n \times f (n-1)$$

S'écrit :

$$f = \lambda f' . \lambda n. \text{ si } n = 0 \text{ alors } 1 \text{ sinon } n \times f' (n-1) (f)$$

Si l'on utilise f au lieu de f' dans cette dernière expression, alors f est une variable liée et ne constitue pas la même référence que le (f) agissant comme opérande.

Considérons la définition $x = F (x)$; elle signifie que x ne change pas lorsqu'il est appliqué à F . Désignons par Y la fonction qui détermine, pour une fonction donnée, son point fixe.

Alors $x = F (x)$ peut s'écrire $x = YF$ sans référence à x

La définition de $f(n)$ est alors

$$f = Y \lambda f. \lambda n. \text{ si } n = 0 \text{ alors } 1 \text{ sinon } n \times f (n-1)$$

Ainsi l'expression dont la valeur est 6! s'écrit :

$$Y \lambda f. \lambda n. \lambda \text{ si } n = 0 \text{ alors } 1 \text{ sinon } n \times f (n-1) (6)$$

$$Y \lambda f. \lambda n. \text{ si } (n = 0) \text{ , } (1, n \times f (n-1)) \text{ (6)}$$

opérateur opérande

opérateur

opérande

L'évaluation est réalisée comme suit :

$$(n = 0) = \text{faux, si (faux)} (1, n \times f(n-1)), 2^e (1, n \times f(n-1), n \times 1(n-1))$$

$$Y \lambda f. \lambda n. n \times f(n-1) (6) \rightarrow Y \lambda f. 6 \times f(5)$$

$$Y \lambda f. 6 \times f(5) \rightarrow 6 \times Y \lambda f. \lambda n. \text{ si } (n = 0) (1, n \times f (n-1)) (5)$$

etc.....

Landin a donné un algorithme qui définit formellement l'évaluation de telles expressions.

4) Cas d'un bloc en Algol :

Soit le programme Algol

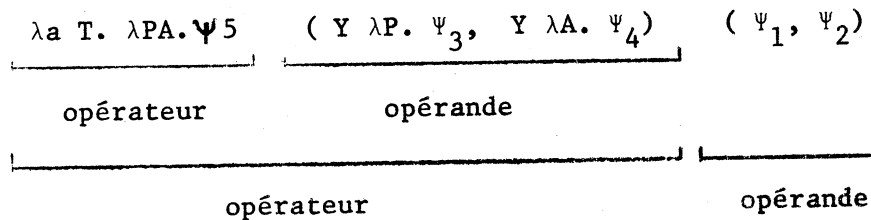
début réel a ; tableau T [] ; procédure P [] ;

Aiguillage A :=..... ;

fin

Les déclarations de type et de tableau ne peuvent faire référence à aucune des autres quantités déclarées dans ce bloc (les bornes du tableau T ne peuvent comporter que des références extérieures au présent bloc,) par contre la procédure P et l'aiguillage A peuvent comporter des références à des éléments de ce bloc, y compris à eux-mêmes.

La partie déclaration d'un bloc doit donc être décomposée en deux groupes distincts : le groupe relatif aux déclarations de type et de tableau est non récursif, par contre le groupe relatif aux déclarations de procédure et d'aiguillage doit être considéré comme récursif :



Ψ_1 et Ψ_2 désignent les valeurs de a et T

Ψ_3 et Ψ_4 les expressions qui représentent le corps du bloc Ψ_5

Landin montre que cette notion d'expression utilisant la notation de Church peut s'appliquer à tous les concepts du langage Algol.

D) Système interprétatif pour la description sémantique

Pour traduire convenablement un assemblage de symboles, c'est-à-dire lui associer une valeur sémantique, il faut d'abord vérifier sa valeur syntaxique (vrai ou faux) et cela quelle que soit la dimension de l'assemblage.

Prenons l'exemple des expressions postfixées.

Nous pouvons définir la syntaxe des expressions postfixées au moyen d'une procédure booléenne définie récursivement.

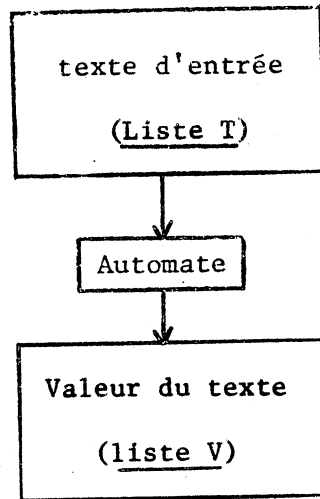
De même, nous pouvons définir la sémantique des expressions au moyen d'un mécanisme d'évaluation récursive. Nous remarquons que pour évaluer une expression postfixée, il n'est pas nécessaire de faire préalablement l'analyse syntaxique complète de cette expression ; il suffit de reconnaître syntaxiquement l'assemblage correspondant à une opération élémentaire puis d'évaluer immédiatement cette opération et ainsi de suite.

Dans un tel processus, description syntaxique et description sémantique sont réellement imbriquées et sont analogues à deux procédures mutuellement récursives.

van Wijngaarden a proposé une méthode de description des langages dans laquelle syntaxe et sémantique sont définies récursivement. [17]

La méthode consiste à définir un langage au moyen d'un automate A qui lit un texte et au fur et à mesure de la lecture produit un autre texte appelé sa valeur.

Globalement l'automate traduit un texte en un autre texte. Le processus peut être schématisé comme suit :



Pour trouver la valeur d'un texte, l'automate explore la liste V dans un sens bien déterminé et tel que l'information la plus récente soit d'abord utilisée.

La liste T (texte d'entrée) est formée de noms séparés par des virgules.

Un nom est défini de la façon suivante :

<métaprimaire>	::= <symbole terminal> <variable> {<nom>} <nom>
<nom simple>	::= <métaprimaire> <nom simple> <métaprimaire>
<nom>	::= <nom simple> <nom>, <nom simple>

Exemples de noms :

A

A dans <lettre>

allera E

B := C \wedge D

<valeur logique> = { <valeur logique> }

Remarque : on suppose que le vocabulaire terminal n'utilise pas les symboles : virgule, guillemets, accolades, sinon il faut évidemment utiliser une autre représentation.

La liste V (valeur du texte) est formée de vérités (propositions vraies) séparées par des virgules.

Une vérité peut avoir l'une des formes suivantes :

nom 1	}	<u>vérités syntaxiques</u>
nom 1 <u>dans</u> nom 2		
vérité \rightarrow nom 1 <u>dans</u> nom 2		
nom 1 = nom 2		
vérité \rightarrow nom 1 = nom 2		
	}	<u>vérités sémantiques</u>

Une vérité est un nom

Remarque : le signe \rightarrow symbolise l'implication logique

Exemples :

vrai

vrai dans <valeur logique>

Procédure <identif> \rightarrow <identif> dans <déclaration procédure>

A = 5

$\omega \rightarrow$ nom = ω

Avant d'attacher une signification à un morceau de texte, l'automate doit vérifier que ce morceau de texte est syntaxiquement correct.

Le mécanisme d'évaluation est donc composé de deux balayages récursifs de la liste V (cela signifie qu'un balayage étant en cours d'exécution, l'automate peut démarrer un nouveau balayage avant que le précédent soit terminé) :

- Un balayage d'applicabilité au cours duquel l'automate cherche une vérité syntaxique applicable au morceau de texte en cours de lecture. Pour effectuer ce balayage, l'automate recherche toutes les partitions du texte d'entrée dans toutes les partitions de la liste V. L'application de cette vérité syntaxique produit une valeur qui est ajoutée au sommet de la liste V : Cette valeur est en effet une vérité.

- Un balayage d'évaluation au cours duquel l'automate cherche une vérité sémantique applicable au morceau de texte pour lequel une vérité syntaxique a été appliquée. De la même façon, l'application de cette vérité sémantique au morceau de texte produit une valeur qui est ajoutée au sommet de la liste V. (Cette valeur est aussi une vérité).

Pour démarrer l'évaluation d'un texte, l'automate doit trouver au départ une première liste de vérités dans V. Les vérités sémantiques suivantes sont incluses dans l'automate et doivent être considérées comme la partie initiale de la liste V :

- 1 valeur <nom simple> = <nom simple> ,
- 2 valeur {<nom>} = <nom> ,
- 3 valeur {<nom> , <nom simple>} = valeur <nom> , valeur <nom simple>

Exemple :

Nous désirons évaluer l'expression booléenne suivante, écrite en langage ALGOL :

si vrai , alors faux \vee vrai \wedge \neg faux sinon vrai

L'automate n'ayant à sa disposition dans la liste V que les 3 vérités précédentes qui forment son mécanisme interne, il faut ajouter à la liste \vee les vérités : v_i, v_{i+1}, \dots, v_n qui définissent la syntaxe et la sémantique des expressions booléennes. Cela permettra à l'automate de faire l'évaluation de l'expression booléenne donnée.

Si l'on donne cette liste de vérités écrite entre guillemets, comme texte d'entrée à l'automate, on ajoutera à la liste V, cette liste de vérités débarassée de ses guillemets : en effet la valeur de ce texte est bien égale à cette liste en vertu de la vérité 2 du mécanisme interne.

$$\underline{\text{valeur}} \quad v_1, v_2, \dots, v_n = v_1, v_2, \dots, v_n$$

Si maintenant à la suite de cette liste entre guillemets, on donne comme texte d'entrée le nom :

si vrai alors faux vrai | faux sinon vrai

alors l'automate pourra déterminer la valeur de cette expression grâce à la liste de vérités ajoutées à V et finalement ajoutera cette valeur à la liste V.

La description des expressions booléennes que nous utilisons ici s'écrit en utilisant la notation du rapport Algol :

1 Syntaxe

<valeur logique> ::= vrai | faux
 <lettre> ::= A | B | C | D
 <identificateur> ::= <lettre>
 <variable bool> ::= <identificateur>
 <primaire bool> ::= <valeur logique> | <variable bool> | (<expr bool>)
 <secondaire bool> ::= <primaire bool> | \neg <primaire bool>
 <facteur bool> ::= <secondaire bool> | <facteur bool> \wedge <secondaire bool>
 <expr bool simple> ::= <facteur bool> | <expr bool simple> \vee <fonction bool>
 <expr bool> ::= <expr bool simple> | si <expr bool> alors <expr bool simple> sinon <expr bool>

2 Sémantique

Une expression booléenne est une règle pour le calcul d'une valeur logique. Dans le cas des expressions booléennes simples, on obtient cette valeur en exécutant les opérations logiques indiquées sur les valeurs logiques effectives des primaires booléens de l'expression.

Pour les valeurs logiques, la valeur logique effective est évidente.

Pour les variables booléennes, il s'agit de la valeur courante.

Pour les expressions booléennes entre parenthèses, on doit exprimer leur valeur récursivement en fonction des valeurs des primaires des 3 catégories.

Pour les expressions booléennes les plus générales contenant si, alors, sinon, une seule parmi les deux expressions booléennes suivant alors et sinon est choisie en fonction de la valeur logique de l'expression booléenne qui suit si. Ce choix est fait de la manière suivante : l'expression booléenne suivant si est évaluée : si sa valeur est vrai, la valeur de l'expression booléenne est alors la valeur de l'expression qui suit alors ; si sa valeur est faux, la valeur de l'expression booléenne est la valeur de l'expression qui suit sinon. De même que pour les expressions booléennes entre parenthèses, ces valeurs doivent être exprimées récursivement.

La signification des opérateurs logiques est donnée par le tableau suivant :

b_1	<u>faux</u>	<u>faux</u>	<u>vrai</u>	<u>vrai</u>
b_2	<u>faux</u>	<u>vrai</u>	<u>faux</u>	<u>vrai</u>
$\neg b_1$	<u>vrai</u>	<u>vrai</u>	<u>faux</u>	<u>faux</u>
$b_1 \wedge b_2$	<u>faux</u>	<u>faux</u>	<u>faux</u>	<u>vrai</u>
$b_1 \vee b_2$	<u>faux</u>	<u>vrai</u>	<u>vrai</u>	<u>vrai</u>

La succession des opérations dans une expression est en général, de la gauche vers la droite, compte tenu des règles de priorité suivantes dans l'ordre décroissant :

- 1) \neg
 2) \wedge
 3) \vee

Les expressions contenues entre parenthèses sont calculées séparément et leurs valeurs utilisées dans la suite des calculs.

La description équivalente exprimée en utilisant une liste de vérités est donnée par le tableau ci-après.

Remarque :

Les vérités 1, 2, 3 sont les vérités initiales incorporées dans l'automate. Les vérités 4, 5, et 43 correspondent aux cas indéfinis (ω est le symbole indéfini).

Par exemple : 3, vrai ne peut être évalué car syntaxiquement incorrect : aucune vérité syntaxique n'est applicable à 3, l'automate appliquera donc la vérité initiale interne.

$$\text{valeur } 3 = 3$$

Revenant au début de la liste V, l'automate applique alors la vérité 43

$$3 = (\omega)$$

Ensuite la vérité 1

$$\text{valeur } (\omega) = \omega$$

Ensuite la vérité 5

$$\omega \rightarrow \omega = (\omega)$$

Ensuite la vérité 1 et la vérité 5 sont appliquées indéfiniment.

Ainsi une fois qu'on a trouvé ω , on trouvera indéfiniment ω : c'est la définition même de l'indéfinition.

Description syntaxique et sémantique des expressions booléennes

- 1 valeur <nom simple> = <nom simple> ,
- 2 valeur ^(<nom>) = <nom> ,
- 3 valeur{ <nom> , <nom simple> } = valeur <nom> ,
valeur <nom simple> ,
- 4 $\omega = \omega'$,
- 5 $\omega \rightarrow \langle \text{nom} \rangle = \omega'$,
- 6 <valeur logique> = (<valeur logique>) ,
- 7 faux dans <valeur logique> ,
- 8 vrai dans <valeur logique> ,
- 9 D dans <lettre> ,
- 10 C dans <lettre> ,
- 11 B dans <lettre> ,
- 12 A dans <lettre> ,
- 13 <lettre> dans <identificateur> ,
- 14 faux \vee faux = faux ,
- 15 faux \wedge vrai ,
- 16 vrai \vee faux ,
- 17 vrai \wedge vrai ,
- 18 faux \wedge faux = faux ,
- 19 faux \wedge vrai = faux ,
- 20 vrai \wedge faux = faux ,
- 21 vrai \wedge vrai ,
- 22 \neg vrai = faux ,
- 23 \neg faux ,
- 24 <expr bool simple> \vee <facteur bool> = valeur <expr bool simple> \vee valeur <facteur bool> ,
- 25 <facteur bool> \wedge <secondaire bool> = valeur <facteur bool> \wedge valeur <secondaire bool> ,
- 26 \neg <primaire bool> = \neg valeur <primaire bool> ,
- 27 (<expr bool>) = <expr bool> ,
- 28 si faux alors <expr bool simple> sinon <expr bool> = <expr bool> ,

Consultant de nouveau la liste, à partir du sommet, l'automate cherche une vérité applicable à vrai, il trouve la vérité 8 : vrai dans <valeur logique>. Il cherche de nouveau une vérité applicable à <valeur logique>, il trouve la vérité 41 qui fournit la valeur vrai dans <primaire bool> qui est ajoutée à la liste V, puis cherche de nouveau une vérité applicable à <primaire bool> (vérité 38) qui fournit la valeur vrai dans <secondaire bool> qui est ajoutée à la liste V, etc...

Finalement, l'application de la vérité 32 fournira la valeur

vrai dans <expr bool>

L'automate reprend alors le balayage initial de l'expression (application de la vérité 31 : il trouve le symbole alors et cherche une vérité applicable à la suite de l'expression :

faux \vee vrai \wedge \neg faux sinon vrai

La première vérité applicable est 33 et le résultat est (après un certain nombre de balayages additionnels)

faux \vee vrai \wedge \neg faux dans <expr bool simple>.

L'automate reprend alors le balayage initial de l'expression (vérité 31) : il trouve le symbole sinon, il cherche une vérité applicable à vrai, et trouvera cette fois près du sommet de la liste (comme résultat de l'application de la vérité 32 au début du balayage).

vrai dans <expr bool>.

Le balayage d'applicabilité est donc fini et le résultat est l'addition dans la liste V de la vérité :

si vrai alors faux \vee vrai \wedge \neg faux sinon vrai
dans <expr bool>.

L'automate cherche maintenant une vérité sémantique applicable à l'expression il trouve la vérité 29 et ajoute à V le résultat.

$$\text{si } \underline{\text{vrai}} \text{ alors } \underline{\text{faux}} \vee \underline{\text{vrai}} \wedge \neg \underline{\text{faux}} \text{ sinon } \underline{\text{vrai}} = \underline{\text{faux}} \vee \underline{\text{vrai}} \wedge \neg \underline{\text{faux}}$$

Il cherche alors une vérité sémantique applicable à $\underline{\text{faux}} \vee \underline{\text{vrai}} \wedge \neg \underline{\text{faux}}$ et trouve la vérité 24.

$$\underline{\text{Faux}} \vee \underline{\text{vrai}} \wedge \neg \underline{\text{faux}} = \underline{\text{valeur faux}} \vee \underline{\text{valeur}} \{ \underline{\text{vrai}} \wedge \neg \underline{\text{faux}} \}$$

La vérité 25 donne :

$$\underline{\text{valeur}} \{ \underline{\text{vrai}} \wedge \neg \underline{\text{faux}} \} = \underline{\text{valeur}} \underline{\text{vrai}} \wedge \underline{\text{valeur}} \{ \neg \underline{\text{faux}} \}$$

La vérité 23 donne :

$$\underline{\text{valeur}} \{ \neg \underline{\text{faux}} \} = \underline{\text{vrai}}$$

La vérité 21 donne :

$$\underline{\text{valeur}} \underline{\text{vrai}} \wedge \underline{\text{valeur}} \underline{\text{vrai}} = \underline{\text{vrai}}$$

et la vérité 19 :

$$\underline{\text{valeur}} \underline{\text{faux}} \vee \underline{\text{valeur}} \underline{\text{vrai}} = \underline{\text{vrai}}$$

Le résultat final est donc l'addition de la vérité :

$$\underline{\text{faux}} \vee \underline{\text{vrai}} \wedge \neg \underline{\text{faux}} = \underline{\text{vrai}}$$

dans la liste

Intérêt de ce mécanisme d'évaluation

Ce mécanisme peut être étendu à tout un langage : cela a été réalisé par J.W.de Bakker pour le langage ALGOL avec quelques restrictions [18].

L'exécution d'un programme ALGOL est réalisée de la façon suivante :

Le texte d'entrée se compose :

- de la définition du langage écrite entre guillemets
- du programme ALGOL à exécuter

La liste \checkmark contiendra :

- La liste des vérités qui définissent le langage.
- La valeur du programme ALGOL (ensemble des résultats intermédiaires écrites sous forme de vérités dans \checkmark).

Ce mécanisme permet d'exprimer simplement des concepts comme celui par exemple de la substitution paramètre effectif-paramètre formel, définie par les deux vérités suivantes :

- (1) $\langle \text{variable} \rangle := \langle \text{expression} \rangle = \{ \langle \text{variable} \rangle = \underline{\text{valeur}} \langle \text{expression} \rangle \}$,
 - (2) $\langle \text{variable 1} \rangle = \langle \text{variable 2} \rangle \rightarrow \langle \text{variable 1} \rangle := \langle \text{variable 2} \rangle := \langle \text{expr} \rangle$
- qui précisent qu'avant d'effectuer la valeur d'une expression à une variable on cherche s'il y a deux variables identiques.

La substitution paramètre effectif x-paramètre formel X peut être représentée par :

$$x := \langle X \rangle$$

Supposons $x := a + b \times c$

" Lorsque cette affectation est réalisée, elle produit $x = X$ en vertu de (1) qui est ajoutée à la liste V

Pour chaque occurrence du paramètre formel X on évaluera donc :

$X := a + b \times c$ en vertu de (2)

Supposons, qu'il y ait une autre substitution.

$X := \langle Y \rangle$ qui lorsqu'elle sera évaluée produira $X = Y$ qui sera ajoutée à V

A chaque occurrence de Y, on évaluera

$$Y := a + b \times c$$

On voit que les substitutions ne sont pas textuellement exécutées mais qu'elle laisse dans la liste V des vérités ($x = X$, $X = Y$) qui permettront l'évaluation de l'expression correspondante.

Conclusion générale.

Les notations sémantiques n'ont pas encore dépassé le stade expérimental. Aucune des notations proposées jusqu'ici n'apportent une solution satisfaisante aux problèmes de la description sémantique. Comme pour la description syntaxique, la principale difficulté tient à ce qu'on s'adresse à deux catégories bien différentes de personnes : les programmeurs qui ont à utiliser le langage pour résoudre leur problème et les réalisateurs de compilateurs qui doivent mettre le langage en oeuvre sur des machines données. Bien que l'on recherche activement des méthodes tendant à rendre automatique la fabrication des compilateurs, les difficultés restent très grandes à l'heure actuelle étant donné que la plus grosse partie du travail dans la réalisation d'un compilateur consiste à définir un langage objet et à établir une correspondance entre les éléments d'un langage source donné et du langage objet associé et que cette phase est difficilement automatisable dans l'état des connaissances actuelles.

C H A P I T R E IV

NOTATIONS PARENTHESÉESI - PROPRIETES DES NOTATIONS PARENTHESÉES.

La plupart des notations utilisées dans les langages de programmation sont des notations parenthésées.

Ces notations possèdent un certain nombre de propriétés remarquables relatives à l'emboîtement des parenthèses et à la recherche des paires les plus intérieures sur lesquelles sont basées la compilation des expressions. L'un des buts de la compilation est en effet le passage de la forme parenthésée, permettant l'imbrication des opérations à une forme sans parenthèses permettant leur évaluation séquentielle sur un ordinateur.

En général, il existe dans un langage plusieurs sortes de parenthèses : typographiquement, elles ont souvent la même représentation alors que fonctionnellement elles sont très différentes : ainsi les parenthèses d'expression sont fonctionnellement différentes des parenthèses de fonction.

Dans Algol 60, par exemple, on trouve les catégories suivantes :

()	expression
()	fonction procédure
()	instruction procédure

[]	indicateur d'aiguillage
[]	déclaration de tableau
[]	variable indicée dans une expression
[]	variable indicée en partie gauche d'une affectation simple
[]	" " " " multiple
[]	" " en partie paramètre effectif
[]	" " contrôlée par une instruc- tion <u>pour</u>

<u>début</u>	<u>fin</u>	parenthèses de bloc
<u>début</u>	<u>fin</u>	parenthèses d'instruction composée

De plus, certains séparateurs peuvent jouer le rôle d'une parenthèse gauche, d'une parenthèse droite ou les deux simultanément.

Par exemple dans l'expression conditionnelle ALGOL

si E1 alors E2 sinon E3 ;

le délimiteur si joue le rôle d'une parenthèse gauche pour E1

le délimiteur alors joue le rôle {

- 1) d'une parenthèse droite pour E1
- 2) d'une parenthèse gauche pour E2

le délimiteur sinon joue le rôle {

- 1) d'une parenthèse droite pour E2
- 2) d'une parenthèse gauche pour E3

Le délimiteur ; joue le rôle d'une parenthèse droite pour E3

1) Cas où il existe une seule espèce de parenthèses.

a) Poids d'un élément.

Dans l'écriture des expressions nous pouvons considérer trois sortes d'éléments :

- les parenthèses gauches
- les parenthèses droites
- les autres éléments : opérateurs, variables et constantes

A chaque élément δ d'une expression, on associe une fonction $P(\delta)$ désignant le poids de l'élément δ et définie de la façon suivante :

- si δ est une parenthèse gauche : $P(\delta) = + 1$
- si δ est une parenthèse droite : $P(\delta) = - 1$
- si δ est un autre élément : $P(\delta) = 0$

b) Fonction de distribution d'une suite d'éléments.

Considérons une suite d'éléments numérotés de 1 à n

A l'élément de rang i est associée la valeur de la fonction $P(\delta_i)$ définie précédemment.

Définition :

La fonction de distribution d'une suite de n éléments est définie par :

$$S(n) = \sum_{i=1}^{i=n} P(\delta_i)$$

c) Couplage des parenthèses.

Un ensemble de n éléments comportant des parenthèses est dit couplé si le nombre de parenthèses gauches est égal au nombre de parenthèses droites.

Proposition 1 Le couplage de n éléments s'exprime par la condition nécessaire et suffisante :

$$S(n) = 0$$

En effet soit k le nombre de parenthèses gauches contenues dans l'ensemble de n éléments, la somme des poids de ces k parenthèses gauches est égal à k , par définition il y a aussi k parenthèses droites dont la somme des poids est égale à $-k$ la somme des $n - 2k$ éléments restants est égal à 0 donc $S(n) = k - k + 0 = 0$.

Réciproquement supposons $S(n) = 0$, soit k le nombre de parenthèses gauches et k' le nombre de parenthèses droites.

La somme des poids des k parenthèses gauches est égal à k
 " " " k' " droites " " à $-k'$
 " " des $n - (k + k')$ éléments restants est égal à 0

La fonction de distribution des n éléments est égale à

$$k - k'$$

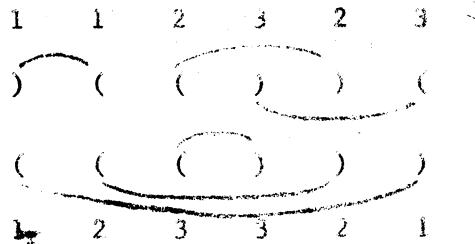
Par hypothèse sa valeur est égale à 0 donc

$$k - k' = 0 \text{ d'où } k = k'$$

d) Exemples de couplages.

Si un ensemble d'éléments est couplé, alors à toute parenthèse gauche on peut associer une parenthèse droite et une seule (en général de plusieurs façons différentes).

Dans les exemples suivants, les numéros définissent l'association des parenthèses gauches et droites :



Séparation de deux paires de parenthèses.

Deux paires de parenthèses sont dites séparées ou non emboîtées si elles correspondent au couplage suivant :



e) Couplage correct des parenthèses.

Un ensemble de p paires de parenthèses est correctement couplé s'il satisfait aux deux conditions suivantes :

- une parenthèse gauche est toujours associée avec une parenthèse droite située à sa droite.
- il n'existe aucun couplage de deux paires de parenthèse définissant deux paires non emboîtées.

Proposition 2 La première condition est équivalente à la relation :

$$\forall i = 1, \dots, n \quad S(i) > 0$$

En effet dire qu'une parenthèse gauche est toujours associée à une parenthèse droite située à sa droite, c'est-à-dire que dans une suite de i éléments le nombre de parenthèses gauches est toujours supérieur ou égal au nombre de parenthèses droites puisque l'occurrence d'une parenthèse droite (de poids -1) est associée à l'occurrence antérieure d'une parenthèse gauche (de poids $+1$)

Réciproquement si $S(i)$ est ≥ 0 pour toute valeur de i alors le premier élément d'une expression ne peut pas être une parenthèse droite (de poids -1), ni le premier élément d'une fin d'expression dont la partie antérieure ne contient que des paires correctement couplées.

Proposition 3 La deuxième condition est équivalente à la suivante :

Une parenthèse droite est toujours associée à la dernière parenthèse gauche rencontrée.

En effet cette règle d'association rend impossible l'occurrence de deux paires de parenthèses non emboîtées.

f) Emboitement des paires de parenthèses - Recherche des paires les plus intérieures.

Proposition 4 Un couplage correct de p paires de parenthèses contient au moins une paire intérieure à toutes les autres, c'est-à-dire qui ne contient aucune autre paire.

En effet considérons une suite de n éléments contenant p paires de parenthèses correctement couplées, alors

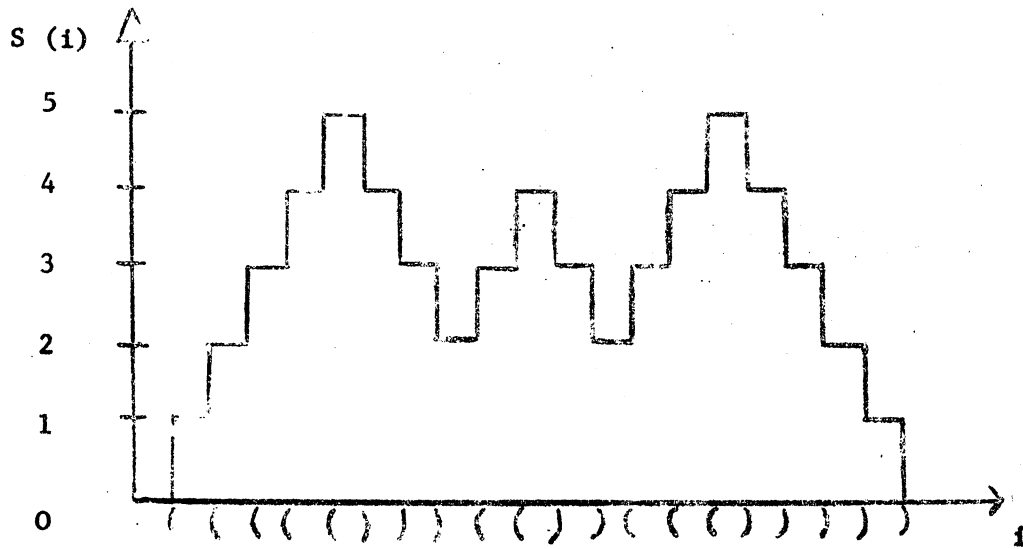
$S(n) = 0$ et pour $\forall i = 1, \dots, n$ on a $S(i) \geq 0$

D'autre part $S(0) = 0$ (par définition)

La fonction $S(i)$ qui reste positive ou nulle dans l'intervalle $(0, n)$ et s'annule pour $i=0$ et $i=n$ passe par un maximum au moins dans cet intervalle, ce maximum est associé à la parenthèse gauche de la paire la plus intérieure.

Si la fonction passe par plusieurs maxima, le nombre de paires les plus intérieures est égal au nombre de maxima de la fonction $S(i)$.

Exemple :



2) Cas où il existe des parenthèses de diverses espèces.

Il est nécessaire de définir une fonction de distribution pour chaque espèce de parenthèse de la façon suivante :

La fonction de distribution S_k relative aux parenthèses de k^e espèce est égale à :

$$S_k(n) = \sum_{i=1}^n P_k(\delta_i)$$

avec $P_k = +1$ si δ est une parenthèse gauche de k^e espèce

$P_k = -1$ si δ " droite " "

$P_k = 0$ si δ est un autre élément

Le couplage des parenthèses de diverses espèces s'exprime alors par la condition :

$$S_1(n) = S_2(n) = \dots = S_k(n) = 0$$

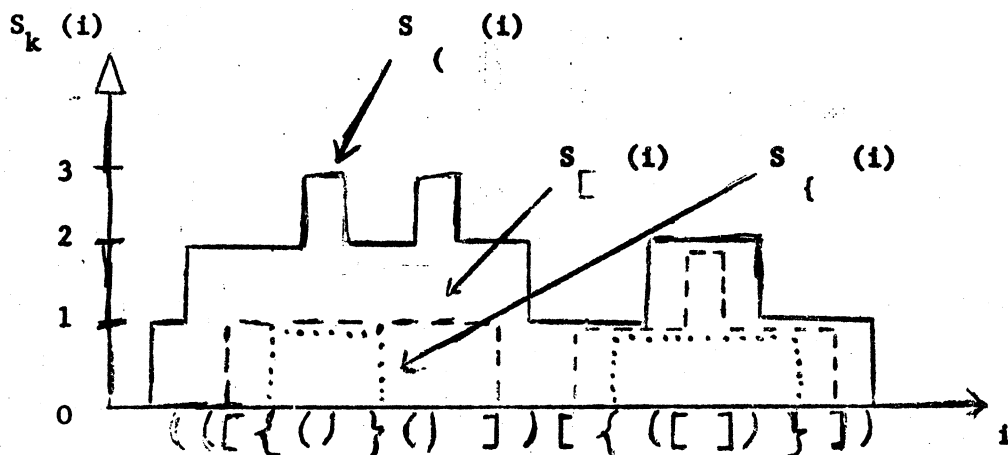
et le couplage correct des parenthèses de diverses espèces par les trois conditions supplémentaires:

1 - $\forall i = 1, 2, \dots, n \quad S_1(i) \geq 0, S_2(i) \geq 0, \dots, S_k(i) > 0$

2 - Une parenthèse droite de k^e espèce est toujours associée à la dernière parenthèse gauche de k^e espèce.

3 - Soit $S_\lambda, S_\mu, S_\gamma$ les fonctions de distribution associées aux parenthèses d'espèce λ, μ, γ , chaque fois que S_λ , par exemple, repasse en décroissant par une valeur v , S_μ et S_γ doivent reprendre les valeurs qu'elles avaient lorsque S_λ est passée en croissant par la valeur v . Cette dernière condition évite des couplages incorrects du type suivant : $(\quad)(\quad)$

Exemple :



3) Cas où il existe une hiérarchie des couplages.

En ALGOL, on peut trouver des expressions donc des parenthèses d'expressions à l'intérieur des parenthèses d'instructions composées ou de blocs (début et fin) mais l'inverse n'est pas possible.

On peut exprimer cette condition par le fait que chaque fois que la fonction de distribution associée aux parenthèses d'instructions décroît, la fonction de distribution des parenthèses d'expression doit être nulle.

4) Cas où il existe des délimiteurs ayant une double fonction de délimitation.

C'est le cas des délimiteurs si, alors et sinon dans l'écriture des expressions en ALGOL.

Au lieu d'une fonction on associe à ces séparateurs un vecteur à 2 composantes.

Initialement la valeur des deux composantes est nulle

A chaque occurrence d'un délimiteur si, la valeur de la première composante augmente de 1

A chaque occurrence d'un délimiteur alors, la valeur de la première composante diminue de 1 et celle de la seconde augmente de 1

A chaque occurrence d'un délimiteur sinon, la valeur de la deuxième composante diminue de 1

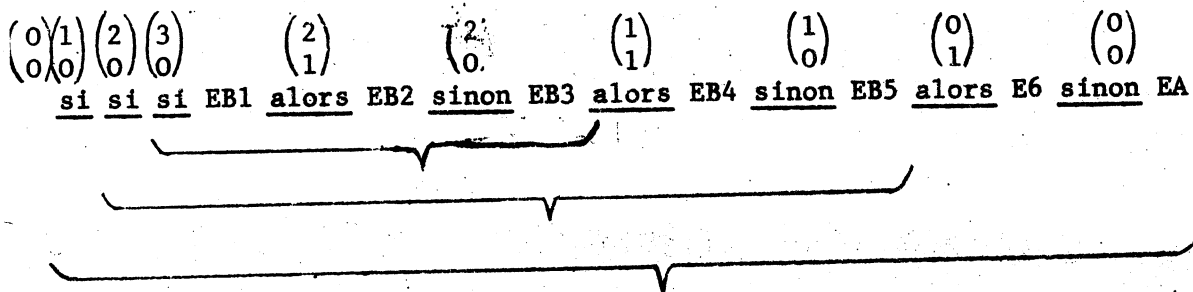
Le couplage correct s'exprime par les deux conditions

- la valeur d'une composante n'est jamais négative
- à la fin d'une expression, la valeur des 2 composantes est nulle

Remarque : Il n'y a pas de restrictions sur la possibilité d'expressions conditionnelles entre si et alors, entre alors et sinon, après sinon.

Exemples :

a) Imbrication des expressions conditionnelles entre si et alors



b) Imbrication des expressions conditionnelles entre alors et sinon

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 0 \\ 2 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

si EB1 alors si EB2 alors E3 sinon E4 sinon E5

On peut remarquer qu'en ALGOL 60, on ne peut pas écrire une expression conditionnelle entre alors et sinon si elle n'est pas transformée en expression simple par une mise entre parenthèses.

Cette restriction s'exprime par la condition supplémentaire :

- la valeur de la deuxième composante est toujours ≤ 1

c) Imbrication des expressions conditionnelles entre sinon et la fin de l'expression

Etant donné que la valeur des 2 composantes est nulle à chaque occurrence d'un sinon, tout se passe comme si l'on était dans la position initiale.

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

si EB1 alors E2 sinon E3 si EB4 alors E5 sinon E6

5) Généralisation.

La même méthode peut être appliquée aux délimiteurs jouant un rôle double :

- la virgule qui est équivalente aux deux parenthèses droite et gauche de la même espèce que celles qui ouvrent et ferment la liste d'éléments correspondants.
- les délimiteurs pas, jusqu'à, tant que qui délimitent deux expressions dans un élément de liste de pour.
- le délimiteur faire qui joue le rôle de parenthèse droite vis à vis du délimiteur pour pour la liste de pour et le rôle de parenthèses gauche d'instruction pour l'instruction qui suit le délimiteur faire.

II - ANALYSE DES NIVEAUX DE PARENTHÈSES.

Sous-expression : une paire de parenthèses et son contenu définissent une sous-expression.

A l'intérieur d'une sous-expression donnée, il y a donc normalement d'autres sous-expressions emboîtées dans celle-ci, leur nombre étant égal au nombre de paires de parenthèses contenues dans la sous-expression donnée.

Une expression est, en général, une sous-expression incomplète dans laquelle il manque les parenthèses gauche et droite qui délimitent normalement une sous-expression.

Si l'on convient de délimiter une expression à gauche et à droite par une paire de parenthèses implicites, on peut définir une expression comme une sous-expression qui n'est contenue dans aucune autre sous-expression.

Degré d'imbrication d'une sous-expression.

Le degré d'imbrication d'une sous-expression est égal au nombre de sous-expressions qui la contiennent augmenté de 1.

Cette valeur initiale de 1 est due au fait que l'expression complète bien que non écrite obligatoirement entre parenthèses doit être considérée comme une sous-expression qui contient toutes les autres.

Proposition 5 Le degré d'imbrication d'une sous-expression est égal à la valeur de la fonction de distribution associée à la parenthèse gauche qui définit le premier élément de la sous-expression.

En effet, la valeur de la fonction de distribution associée à une parenthèse gauche est égal au nombre de parenthèses gauches qui la précèdent non encore couplées avec les parenthèses droites correspondantes. Cette valeur est donc bien égale au nombre de sous-expressions qui contiennent la sous-expression qui débute à cette parenthèse gauche augmentée de 1.

Dans une expression où les parenthèses sont convenablement couplées on peut donc associer à toute parenthèse gauche un nombre entier égal à la valeur de la fonction de distribution de cet élément et qui représente le degré d'imbrication de la sous-expression correspondante.

Hauteur d'une sous-expression.

On parle également de hauteur d'une sous-expression comme synonyme de degré d'imbrication. Ce terme hauteur provient de la représentation graphique où le degré d'imbrication d'une sous-expression est égal à la hauteur de la parenthèse gauche qui en constitue le premier élément.

Relation entre la hauteur et le couplage des parenthèses.

Proposition 6 Soit k la valeur de la fonction de distribution d'une parenthèse gauche ($k > 1$), la parenthèse droite qui lui est associée est la première située à sa droite pour laquelle la fonction de distribution associée prend la valeur $k - 1$.

Remarquons d'abord que la valeur de la fonction de distribution d'une parenthèse gauche est égale à la valeur de la fonction de distribution de la parenthèse (gauche ou droite) qui précède augmentée de 1. (Pour une parenthèse droite, c'est la valeur de la parenthèse qui précède diminuée de 1).

Soit maintenant k la valeur de la fonction de distribution associée à une parenthèse, deux cas sont à considérer.

- si le premier élément de poids différent de 0 rencontré à droite est une parenthèse droite, la valeur de la fonction de distribution pour cet élément est égale à $k-1$ et les deux parenthèses forment une paire.

- il y a p paires de parenthèses situées à droite, ces paires se couplent une par une, la dernière parenthèse droite de la p^e paire étant affectée d'un nombre égal à k . On retombe alors sur le cas précédent.

Rang d'une sous-expression.

Plusieurs sous-expressions peuvent avoir la même hauteur, la hauteur n'est donc pas suffisante pour définir complètement une sous-expression à l'intérieur d'une expression.

En effet, les parenthèses gauches correspondant à des valeurs égales de la fonction de distribution et les parenthèses droites associées délimitent des sous-expressions de même hauteur.

Définition :

Le rang d'une sous-expression de hauteur h est égal au rang de la sous-expression de hauteur h qui la précède augmentée de 1, la numérotation étant réalisée de la gauche vers la droite.

(Le rang d'une expression complète est égal à 1).

Le nombre entier qui définit le rang est, comme la hauteur, une caractéristique essentielle de toute parenthèse gauche.

Toute sous-expression est ainsi caractérisée à l'intérieur de l'expression qui la contient par les deux nombres entiers associée à sa parenthèse gauche : hauteur et rang.

Exemple :

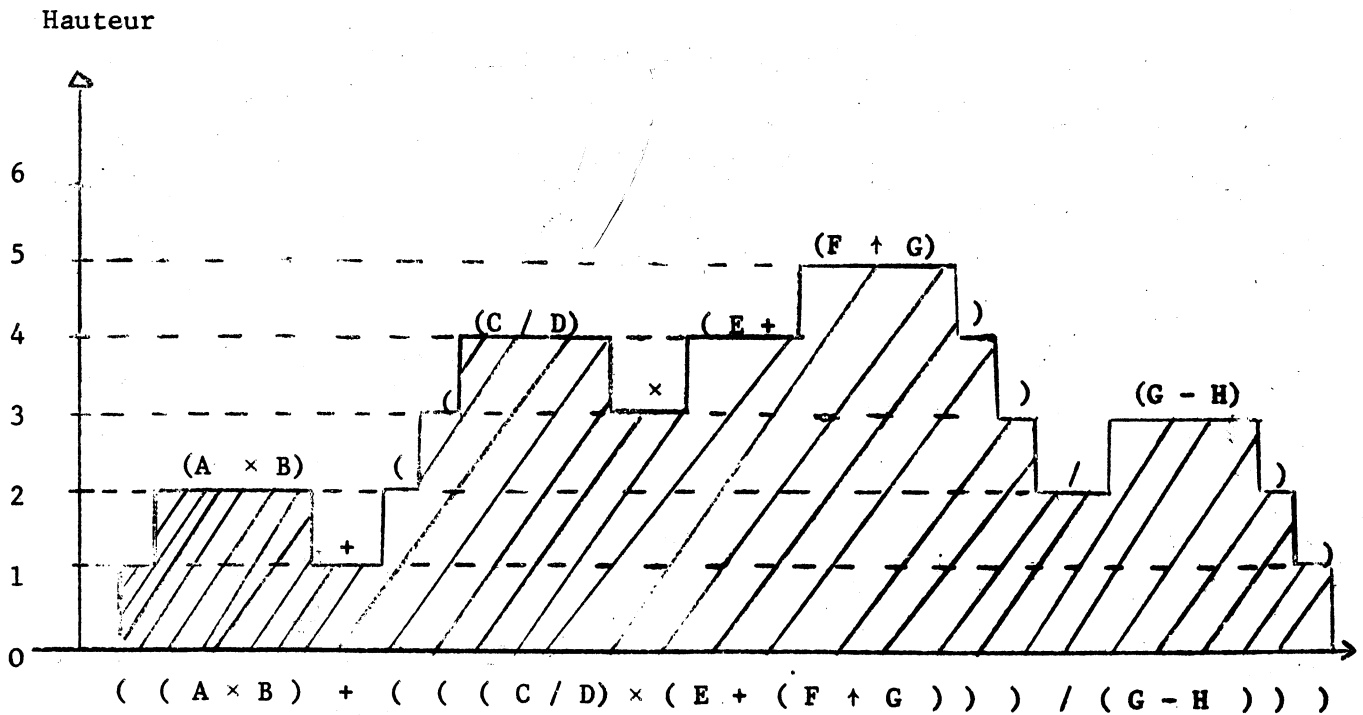
$$((A \times B) + (((C/D) \times (E + (F+G)) / (G-H))))$$

Hauteur	12	234	4	5	3
Rang	11	211	2	1	2

$(A \times B)$ est la sous-expression de hauteur 2 et de rang 1

$(E + (F + G))$ est la sous-expression de hauteur 4 et de rang 2

Représentation graphique.



La hauteur est directement indiquée par l'ordonnée des parenthèses gauches. Pour savoir s'il y a plusieurs sous-expressions de même rang correspondant à une hauteur donnée, il suffit d'observer s'il y a plusieurs paires de parenthèses correspondant à ce niveau.

C H A P I T R E I

DEFINITIONS ET RESULTATS ELEMENTAIRESI - INTRODUCTION.

La programmation au niveau du langage machine est caractérisée par trois aspects fondamentaux :

- l'extension de la notion d'opérande par l'introduction du concept de variable qui permet l'utilisation d'un même algorithme pour des valeurs différentes des variables.

- l'extension de la notion d'opérateur par l'introduction du concept de sous-programme qui permet la définition de nouveaux opérateurs.

- La propriété d'auto-modification, caractéristique des machines à programme enregistré, qui permet de considérer un programme comme un opérateur défini dynamiquement.

La programmation au niveau des langages symboliques est caractérisée par d'autres aspects :

- Notation fonctionnelle : un programme ou un sous-programme est un opérateur, son exécution est une opération d'où l'idée de considérer l'exécution d'un programme comme l'évaluation d'une fonction. Ce point de vue, courant dans les langages usuels, est important dans les systèmes à accès multiple où le programme et les données sont considérées comme des files sur lesquelles on peut effectuer des opérations.

Les langages de commande utilisent tous cette particularité.

Exemple : $\lambda := \text{charger} (\alpha + \text{algol} (\beta + \gamma + \text{extraire} (500, 700, \mu)))$
 où $\lambda, \alpha, \beta, \gamma$ et μ sont des files ; $\text{charger}, \text{algol},$
 extraire et $:=$ sont des opérateurs.

A l'heure actuelle les langages de commande n'est pas encore atteint ce degré de complication mais la tendance dans cette direction est nettement amorcée.

- Phase de liaison.

C'est l'instant où un attribut lié à une quantité est définitivement figé. Par exemple la syntaxe des langages actuels est fixée une fois pour toutes dans la définition du langage. Dans les techniques de programmation machine un grand progrès à été réalisé dans la phase de liaison liée à l'implantation des programmes dans la mémoire centrale, d'abord absolue, puis relative, puis relative avec la possibilité de zones communes mais toujours préalablement à l'exécution dans les systèmes à traitement séquentiel. Dans les systèmes à accès multiple, cette implantation est entièrement dynamique c'est-à-dire fonction de l'exécution du programme et peut varier au cours de celle-ci.

Un autre exemple est celui des procédures ALGOL pour lesquelles la définition ne peut plus changer à l'intérieur du bloc. Dans une généralisation d'ALGOL, van Wijngaarden a proposé l'introduction de déclarations dynamiques de procédures [19].

Par exemple :

```

new P, ..., P := (new s, s := if x > 0 then y else z,
                  s × (s+1), i := i+1)
..., u := 3 × P, ...
P := 3.14, ...
u := 3 × P, ...

```

Dans ce dernier cas, il n'y a plus de distinction entre expressions et instructions et entre les deux types d'affectation (affectation de valeur, affectation de nom).

- Interférence entre évaluation numérique et évaluation symbolique.

Dès l'origine les langages symboliques ont été liés à des domaines d'application distincts.

- Evaluation numérique : c'est le cas de tous les langages dits scientifiques : FORTRAN, ALGOL et leurs dérivés, spécialement conçus pour les applications proprement numériques. Un cas particulier important est celui des langages de gestion (COBOL et ses homologues) où la structure des données est plus importante que leur traitement.

- Evaluation symbolique : c'est le cas des langages de listes (IPL V, LISP) spécialement conçus pour la manipulation des quantités symboliques.

Alors que dans l'évaluation numérique on a tendance à utiliser au maximum des processus itératifs (en Analyse numérique on s'efforce au maximum d'étudier et d'utiliser des algorithmes de calcul basés sur l'itération), dans l'évaluation symbolique, il est naturel d'utiliser des processus récursifs beaucoup mieux adaptés à la structure des informations manipulées.

Les dernières années ont montré en tous cas que cette distinction ne manque pas d'être assez artificielle. Dans LISP 1.5 on trouve les "program features" qui donnent au langage des possibilités d'évaluation numérique. Dans un certain nombre de compilateurs ALGOL on trouve des extensions qui permettent l'évaluation symbolique. Parmi les plus perfectionnés, on peut citer : "FORMULA ALGOL" développé au "Carnegie Institute of Technology" [20], LISPALGOL développé à l'Université de Grenoble [21]

Nous entrons maintenant dans la période où cette interférence des deux aspects va être réalisée non plus au niveau des extensions à un langage de base (LISP en ALGOL) mais au niveau des définitions.

Mentionnons en particulier les efforts actuels pour les langages
PL/1 + "list processing facilities" [22]
ALGOL avec le concept de "records" [23]
LISP2 [24]

2 - SUBSTITUTIONS STATIQUES ET SUBSTITUTIONS DYNAMIQUES.

Considérons une certaine donnée que nous appelons texte, qui peut représenter soit un programme (texte à exécuter) soit une donnée proprement dite (texte à lire).

Une substitution statique est une opération qui permet de produire un texte résultat (texte objet) à partir d'un texte donné (texte source), le mécanisme de substitution étant indépendant de l'exécution du texte source si ce dernier est un texte exécutable.

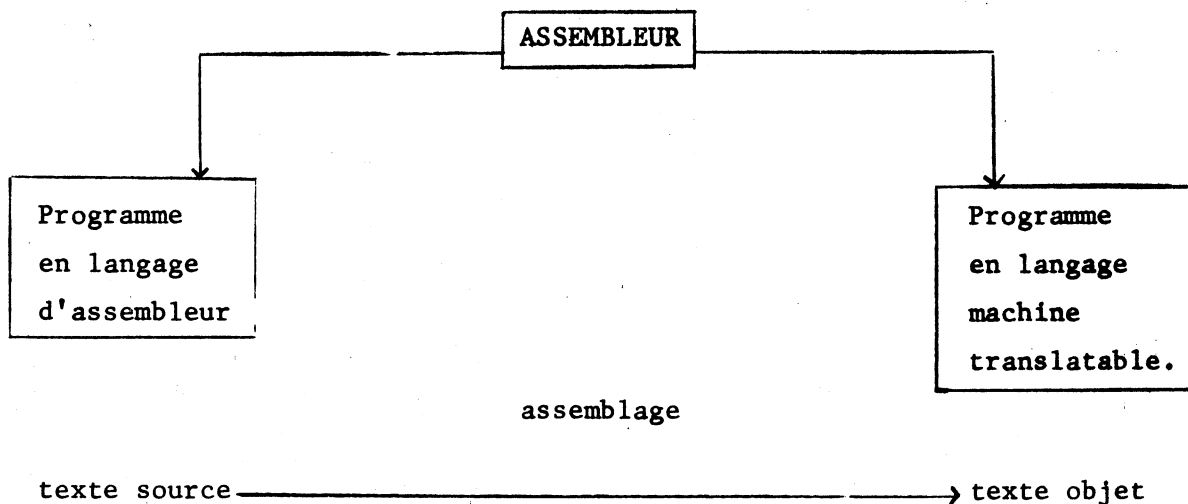
Une substitution dynamique est une opération qui permet de produire un texte résultat à partir de l'exécution d'un texte source.

On peut dire également que dans une substitution statique le mécanisme de substitution est défini indépendamment du texte source ; dans une substitution dynamique au contraire le mécanisme est défini par le texte source.

2.1 - Exemples de substitutions statiques.

2.1.1. Assemblage

Un programme rédigé en langage machine symbolique (langage d'assembleur) est transformé en langage machine avec adresses relatives (langage machine translatable) au moyen d'une substitution statique dite assemblage.



Par exemple l'instruction :

LOOP CLA ALPHA

est transformée en :

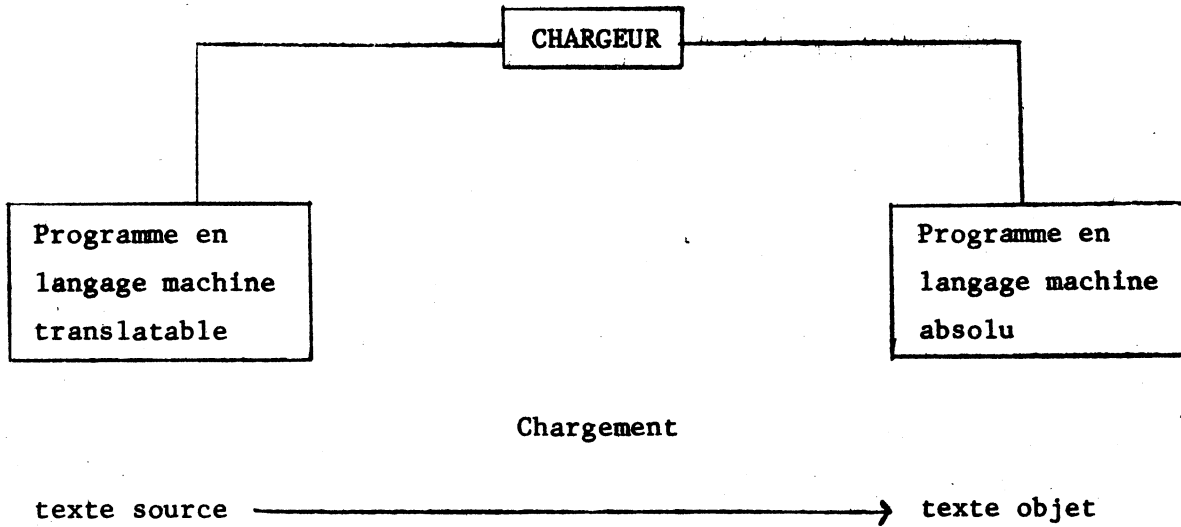
0003 050000 000144

Substitution totale et substitution partielle.

La substitution peut être complète ou partielle suivant que tous les éléments du texte source ou seulement une partie d'entre eux sont remplacés. Dans l'exemple précédent la substitution est totale.

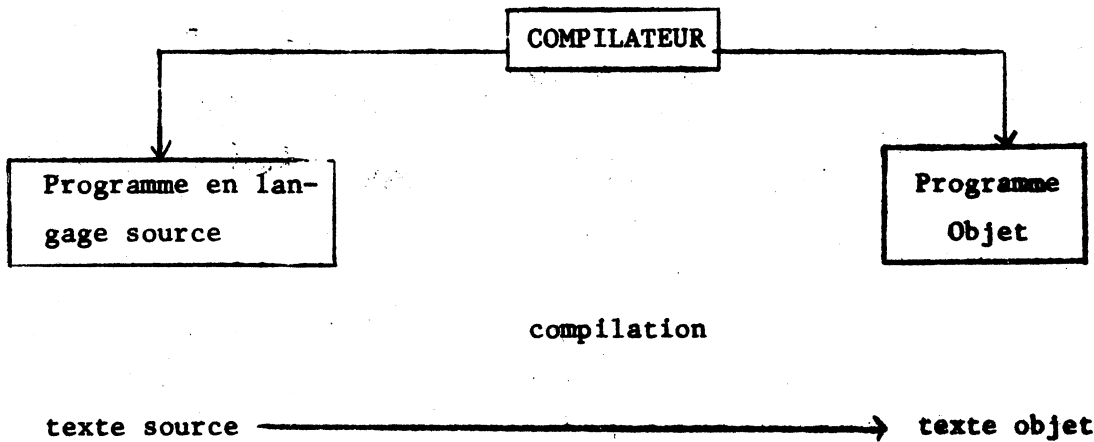
2.1.2. Chargement

Pour des raisons de flexibilité d'implantation, l'assemblage produit des adresses machines relatives. Juste avant l'exécution d'un programme et compte tenu de la place disponible on effectue le chargement c'est-à-dire la transformation des adresses machine relatives en adresses machine absolues.



2.1.3. Compilation

La compilation est un processus de substitution complexe permettant de produire un texte objet (programme objet) à partir d'un texte source (programme source), la complexité provenant de la différence de niveau considérable des langages source et objet.

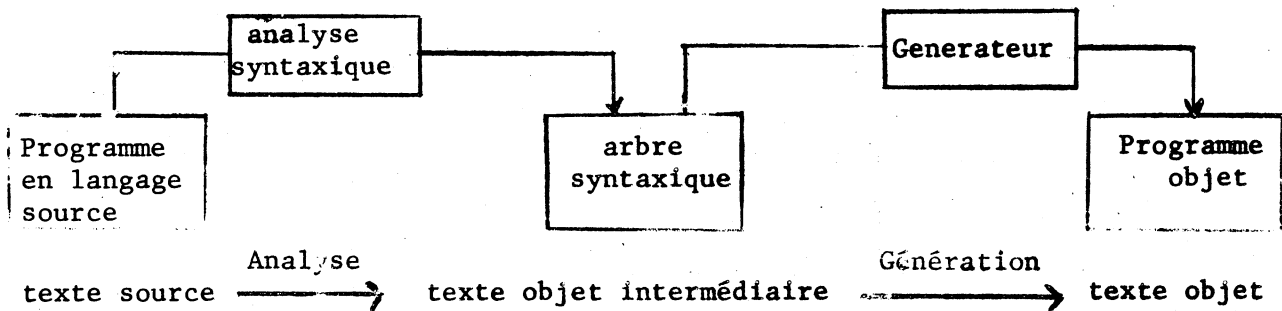


Produit de substitutions statiques.

Il peut être nécessaire ou commode de réaliser une substitution statique complexe en plusieurs phases, le texte objet de la phase précédente devenant le texte source de la phase suivante. La substitution globale est alors équivalente au produit des substitutions élémentaires successives.

Exemples : Dans l'assemblage, il y a en général deux phases distinctes : construction du dictionnaire des références symboliques, assemblage proprement dit.

Dans la compilation, on peut avoir une phase d'éditons avec phase d'analyse syntaxique et une phase de génération.



2.2 - Exemples de substitutions dynamiques.

2.2.1. Interprétation

L'exemple le plus simple d'interprétation est celui du langage machine. Un programme écrit en langage machine est entièrement interprété c'est-à-dire que chaque instruction est en fait un appel de sous-programme câblé à l'intérieur de l'unité centrale. L'interprétation s'effectue en deux phases :

- détermination des arguments
- exécution du sous-programme associé

Alors que dans une substitution statique, les éléments du texte source ne sont pas considérés comme exécutables, dans une substitution dynamique les éléments du texte source sont en fait considérés explicitement comme des éléments exécutables et c'est le résultat produit par leur exécution qui définit la substitution.

- Si l'on définit un sous-programme fermé comme une séquence d'instructions susceptible d'être appelée n'importe où à l'intérieur d'un programme, l'appel d'un sous-programme implique la substitution dynamique de la séquence d'instructions correspondantes. On peut par ce procédé étendre à volonté le code d'instruction d'une machine : c'est la base de toute la programmation.

Dans le cas d'un sous-programme fermé non récursif, la substitution dynamique permet seulement une économie de place considérable par rapport à la substitution statique (correspondant à un sous-programme de type ouvert). Dans le cas d'un sous-programme récursif la substitution dynamique est obligatoire étant donné que le nombre de substitutions (nombre d'appels) dépend de l'exécution du sous-programme.

Appel par valeur.

La différence entre une constante et une variable réside précisément dans la substitution dynamique nécessaire pour définir la valeur instantanée d'une variable. On remarque également que l'évaluation d'une variable peut entraîner l'exécution d'un programme complexe (variable indicée, indicateur de fonction).

Appel par nom.

L'occurrence d'une variable en partie gauche d'une instruction d'affectation implique une substitution dynamique d'adresse. Un paramètre appelé par nom nécessite une substitution dynamique de l'expression associée.

La détermination d'une valeur ou d'une adresse peut impliquer un grand nombre de substitutions successives.

2.2.2. Macro-génération

Un assembleur est rarement limité aux seules possibilités de substitutions statiques. La plupart des assembleurs disposent de pseudo-instructions c'est-à-dire de morceaux de texte exécutables permettant un mélange de substitutions statiques et de substitutions dynamiques.

Toute pseudo-instruction est en fait un appel de sous-programme exécuté au moment de l'assemblage et qui modifie le processus d'assemblage. Parmi ces pseudo-instructions figurent notamment les macro-instructions qui permettent des substitutions dynamiques de morceaux de texte pendant l'assemblage et les pseudo-instructions conditionnelles qui permettent des substitutions conditionnelles.

Le processus d'assemblage devient alors tout-à-fait analogue au processus d'évaluation d'un programme ALGOL par exemple. Les "macro déclarations" correspondent aux déclarations de procédure et les "appels de macro" aux instructions procédures. Au lieu de produire des valeurs du type arithmétique ou booléen, on produit des valeurs qui sont des morceaux de texte.

Par exemple étant donné la macro-définition :

Q POLY	MACRO	COEFF, LOOP, DEG, T, OP
	AXT	DEG, T
	LDQ	COEFF
LOOP	FMP	GAMMA
	OP	COEFF+DEG+1, T
	STO	TEMP
	LDQ	TEMP
	TIX	LOOP, T, 1
	ENDM	QPOLY

l'appel de macro

X015	Q POLY	C1-4, FIRST, 5, 4, FAD
------	--------	------------------------

produit la séquence :

X015	BSS	0
	AXT	5,4
	LDQ	C1-4
FIRST	FMP	GAMMA
	FAD	C1-4+5+1,4
	STO	TEMP
	LDQ	TEMP
	TIX	FIRST,4,1

Les macro-définitions peuvent être emboîtées comme les déclarations de procédure. On peut ainsi définir de nouvelles macros ou redéfinir des macros déjà définies à l'aide d'une seule instruction.

Par exemple, étant donné la définition :

	MAC1	MACRO	MAC2,ALPHA,BETA,GAMMA,DELTA
	MAC2	MACRO	ALPHA
		BETA	A
		GAMMA	B
		DELTA	C
		ENDM	MAC2
		ENDM	MAC1
l'appel		MAC1	ABC, (A,B,C),CLA,ADD,STO
generera :		MACRO	A,B,C
	ABC	CLA	A
		ADD	B
		STO	C
		ENDM	ABC

On trouvera l'utilisation des possibilités de macro-assemblage dans le chapitre relatif à l'évaluation sémantique.

Les possibilités de macro assemblage ont été généralisées dans le "macro generator" de Strachey. Dans un tel système le processus de substitution n'est plus lié à un format particulier de texte. [25]

Une macro définition s'écrit :

§ DEF, nom, texte ;

et un appel de macro

§ nom d'appel, liste d'arguments ;

Un nom d'appel peut être un nom ou un appel de macro et de la même façon un argument peut être un nom ou un appel de macro : à l'intérieur d'un texte, on peut trouver des noms d'appel n'importe où.

L'exemple précédent relatif à Q POLY s'écrit :

macro-définition :

§ DEF,QPOLY, < AXT~3,~4 LDQ~1,~2 FMP GAMMA
~5~1+~3+1,~4 STO TEMP LDQ TEMPSTIX~2,~4, 1> ;

appel de macro :

§ QPOLY,C1-4,FIRST,5,4,FAD ;

L'exemple relatif à MAC 1 s'écrit :

macro-définition :

§ DEF,MAC1,< \$DEF,~1,~2<~1>~3<~2>~4<~3> ; >;

appel de macro :

§ MAC1,ABC,CLA,ADD,STO ;

et produit le résultat :

\$DEF,ABC,CLA<~1>ADD<~2>STO<~3> ;

L'appel de macro

§ ABC,X,Y,Z ;

produit alors :

CLA X ADD Y STO Z

2.2.3. Allocation dynamique

2.2.3.1. Allocation dynamique des quantités

Certains aspects dynamiques du langage ALGOL en particulier les tableaux à bornes variables et les procédures récursives nécessitent une allocation dynamique de la mémoire pour les diverses quantités : variables explicitement déclarées dans le programme, variables intermédiaires liées à l'évaluation des expressions ou de certaines instructions (blocs, instructions POUR, procédures).

Cette correspondance entre adresses symboliques et adresses absolues est en fait une substitution dynamique d'adresse : à chaque adresse symbolique est substituée une adresse absolue qui dépend de l'état d'exécution du programme.

Cette allocation dynamique des quantités est réalisée par un interpréteur dont la fonction essentielle est la gestion de la pile d'évaluation.

2.2.3.2. Allocation dynamique des programmes

En monoprogrammation, l'implantation des programmes c'est-à-dire la zone de mémoire qu'ils occupent pendant leur exécution est définie statiquement par l'intermédiaire de l'opération de chargement. Cela signifie que l'allocation de mémoire est faite indépendamment de l'exécution des programmes et préalablement à celle-ci.

En multiprogrammation, plusieurs programmes sont simultanément en mémoire et l'implantation des programmes change en permanence.

La mise en oeuvre de la multiprogrammation exige en effet que plusieurs programmes soient simultanément en mémoire centrale et qu'ils y demeurent tant qu'ils ne font pas appel à des opérations avec les organes périphériques ou qu'une certaine tranche de temps n'est pas entièrement écoulée. En pratique cela signifie qu'un programme devra être chargé plusieurs fois en mémoire centrale avant que son exécution ne soit terminée et il est très improbable qu'il soit chargé à la même place à chaque reprise.

Cette organisation nécessite une allocation dynamique des programmes réalisée par un interpréteur qui est une partie du système superviseur. La correspondance entre l'adresse symbolique d'une instruction et son adresse physique est réalisée à chaque activation du programme en mémoire centrale. L'allocation des données suit les mêmes règles et peut nécessiter deux niveaux de substitution dans le cas de quantités implantées dynamiquement (pile de quantités implantées dans des zones variables pendant l'exécution).

3 - SYSTEMES INTERPRETEURS ET SYSTEMES COMPILATEURS.

Ce rapide tour d'horizon des processus de substitution fait clairement apparaître la différence entre systèmes interpréteurs et systèmes compilateurs. Les substitutions statiques sont toujours réalisables par des systèmes compilateurs tandis que les substitutions dynamiques nécessitent des systèmes interpréteurs.

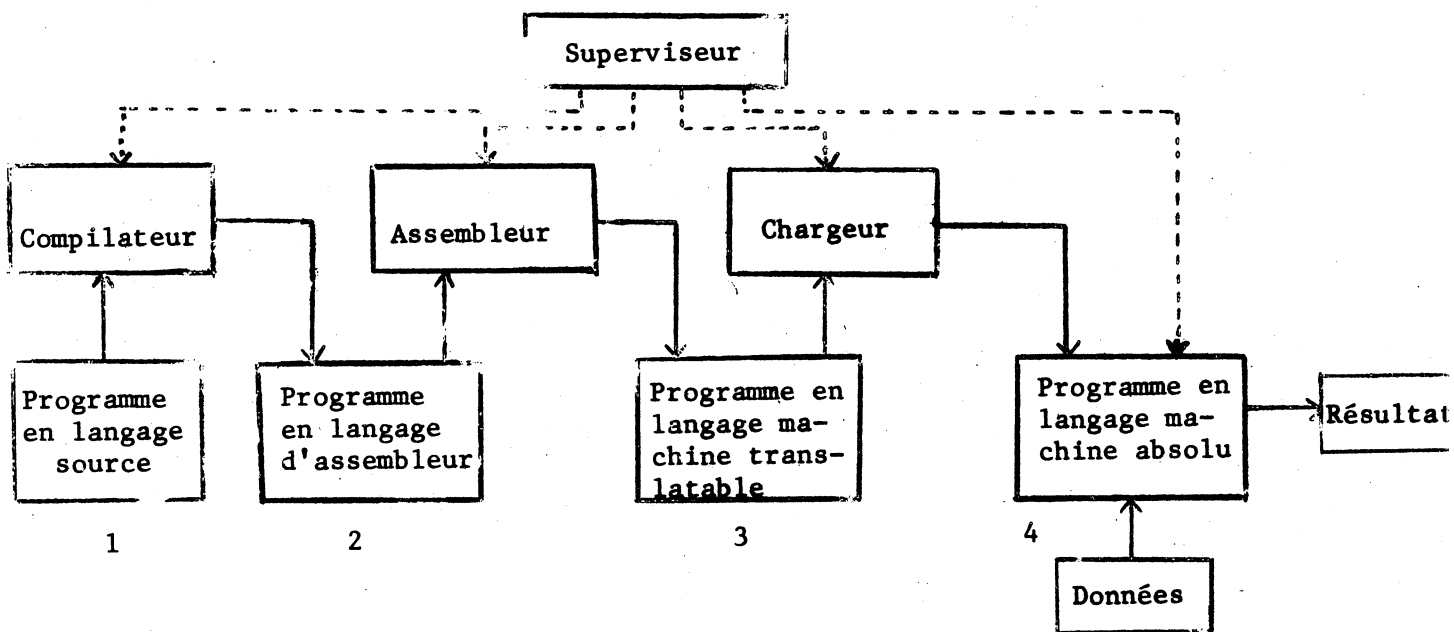
La plupart du temps d'ailleurs, on a des systèmes mixtes dans lesquels interprétation et compilation interfèrent.

- Dans un macro assembleur, on a simultanément substitution statique des adresses et interprétation au niveau des pseudo-instructions.

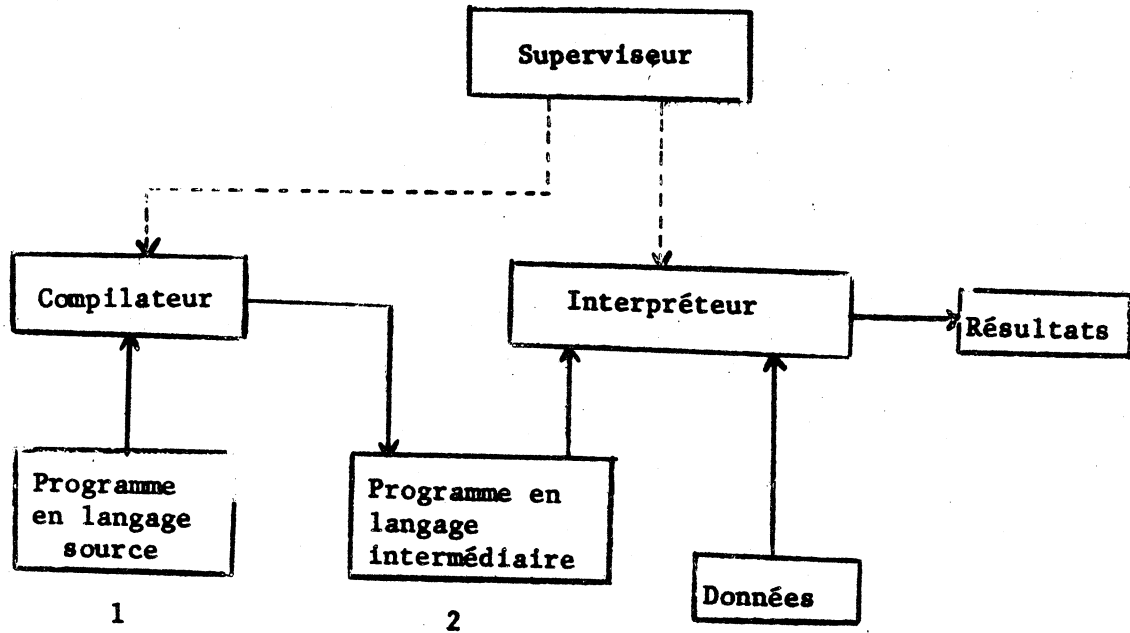
De même, dans un chargeur, on a simultanément substitution statique des adresses et interprétation des pseudo-instructions de chargement.

Un cas analogue est celui des systèmes superviseurs qui sont des interpréteurs exécutant des instructions spéciales appelées commandes; l'exécution d'une commande peut entraîner l'exécution d'opérations complexes telles qu'assemblage, chargement, compilation, interprétation.

- Schéma d'un processus de compilation et d'exécution d'un programme utilisant assembleur et chargeur.



- Schéma d'un processus de compilation et d'exécution d'un programme utilisant un interpréteur.



Remarque 1 : Si le superviseur n'existe pas en tant que programme, sa fonction est alors réalisée par l'opérateur humain qui doit commander et contrôler les diverses phases relatives à l'exécution des programmes.

Structures imbriquées.

Ce sont des structures dont la définition est récursive.

Elles ont une grande importance dans les techniques de programmation où l'un des principaux problèmes est justement l'analyse, la réduction et l'évaluation de telles structures.

3.1 - Expressions complètement parenthésées.

3.1.1. Propriétés élémentaires

Rappelons brièvement les résultats du chapitre précédent :

Les parenthèses sont associées par paires. En attribuant le poids + 1 à toute parenthèse gauche, le poids -1 à toute parenthèse droite, le poids 0 à tous les autres éléments on associe un poids $p(E_i)$ à tout élément E_i de rang i

$$\text{Posons } P(k) = \sum_{i=1}^{k-1} p(E_i)$$

On a les propriétés suivantes :

- pour tout i compris entre 1 et k $P(k) \neq 0$
- pour une expression correctement parenthésée comportant n éléments $P(n) = 0$

Il en résulte que l'on a au moins un maximum qui correspond à la paire la plus intérieure. (Il peut y avoir plusieurs maximum relatifs).

Cela conduit à deux catégories de techniques pour analyser une telle structure : techniques itératives et techniques récursives.

Les techniques itératives ou globales consistent à rechercher successivement les paires les plus intérieures de l'expression et à les réduire une par une. Elles nécessitent un balayage préalable de l'expression pour déterminer le maximum de $P(k)$ qui est par définition associé à la parenthèse gauche la plus

intérieure. La paire la plus intérieure est alors réduite et un nouveau balayage est effectué pour déterminer la parenthèse gauche associée à la valeur du maximum diminuée de 1 et ainsi de suite.

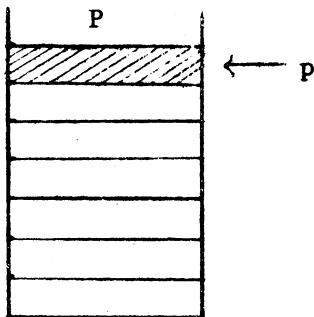
Il peut y avoir plusieurs paires intérieures associées au maximum de $P(k)$, ces paires sont réduites successivement de gauche à droite au cours d'un même balayage.

Les techniques récursives ou séquentielles consistent à réduire séquentiellement les paires de parenthèses au cours d'un seul balayage de gauche à droite autrement dit à effectuer une réduction pour chaque maximum relatif de $P(k)$.

Ces techniques nécessitent l'utilisation de piles de mémoire permettant le stockage des paires incomplètes rencontrées au cours du balayage.

3.1.2. Technique des piles de mémoire.

Une pile est un ensemble de mémoires consécutives dont la dernière seulement, appelée sommet de la pile est accessible par l'intermédiaire d'un index dont la valeur courante définit la hauteur de la pile,



Une pile de mémoires est essentiellement un dispositif de stockage permettant de mettre temporairement en réserve des éléments d'information accessibles dans un certain ordre et réutilisés ultérieurement dans l'ordre strictement inverse.

On remarquera que la pile définit une relation d'ordre entre les éléments. En effet, considérons 2 éléments a et b rangés dans la pile, si a est plus haut que b, cela signifie que a redevient accessible avant b.

Dans la technique usuelle des mémoires adressées, il n'existe pas de relation d'ordre entre les éléments mais seulement entre les adresses des éléments (chaque élément est disponible à tout instant par l'intermédiaire de son adresse).

Les 3 opérations fondamentales que l'on peut effectuer sur une pile sont les suivantes : (nous désignons par $P[p]$ la mémoire sommet de la pile et par p l'index de la dernière mémoire utilisée).

- 1) Entrée d'un élément : $p := p+1 ; P_p := \text{élément}$
- 2) Sortie d'un élément vers S : $S := P_p ; p := p-1$
- 3) Effacement d'un élément : $p := p-1$

Remarque : pour l'opération d'effacement, il n'est pas nécessaire d'effacer physiquement la mémoire sommet de la pile, il suffit de diminuer d'une unité la valeur de l'index p.

3.1.3. Evaluation directe

3.1.3.1. Méthode itérative [26]

Une expression peut être évaluée directement en utilisant l'une des deux techniques précédentes.

L'évaluation itérative nécessite que l'expression soit complètement enregistrée et que les résultats intermédiaires de l'évaluation soient recopiés à l'intérieur de l'expression.

Le mécanisme d'évaluation est défini par l'algorithme suivant :

L'expression à évaluer est désignée par

$E_1 E_2 \dots E_n \omega$ (ω délimiteur de fin d'expression)

DEBUT COMMENTAIRE EVALUATION D UNE EXPRESSION COMPLETEMENT PARENTHESÉE. METHODE ITERATIVE ;
ENTIER I, J, K, P, P_{MAX}, OMEGA, PARGAUCHE, PARDROITE ;
ENTIER TABLEAU E[1:100] ;

BOOLEEN PROCEDURE OPERANDE(X) ;
DEBUT COMMENTAIRE CETTE PROCEDURE NE PREND LA VALEUR VRAI QUE SI X EST UN OPERANDE ;
 ° ° °
FIN ;

BOOLEEN PROCEDURE OPERATEUR(X) ;
DEBUT COMMENTAIRE CETTE PROCEDURE NE PREND LA VALEUR VRAI QUE SI X EST UN OPERATEUR ;
 ° ° °
FIN ;

ENTIER PROCEDURE OP(X, Y, Z) ;
DEBUT COMMENTAIRE CETTE PROCEDURE DONNE LA VALEUR DU RESULTAT DE L OPERATION DEFINIE PAR L OPERATEUR X ET LES OPERANDES Y ET Z ;
 ° ° °
FIN ;

° ° °

PHASE A: P_{MAX} := P := I := 0 ;
POUR I := 1 * 1 TANTQUE E[I] ≠ OMEGA FAIRE
DEBUT
 SI E[I] = PARGAUCHE ALORS
 DEBUT
 P := P + 1 ;
 SI P > P_{MAX} ALORS P_{MAX} := P
 FIN ;
 SI E[I] = PARDROITE ALORS
 DEBUT
 P := P - 1 ;
 SI P < 0 ALORS
 ERREUR1 : ECRIRE(" TROP DE PARENTHESES DROITES. ")
 FIN
 FIN ;
 SI P ≠ 0 ALORS
 ERREUR2 : ECRIRE(" TROP DE PARENTHESES GAUCHES ") ;

PHASE B:
POUR P_{MAX} := P_{MAX} PAS -1 JUSQUA 0 FAIRE
DEBUT
 P := I := 0 ;

PHASE C:
POUR I := 1 * 1 TANTQUE E[I] ≠ OMEGA FAIRE
 SI E[I] = PARGAUCHE ALORS

```

DEBUT
P:=P+1 ;
SI P=PMAX ALORS
  DEBUT
  SI OPERANDE(E[I+1]) ET OPERATEUR(E[I+2]) ET
  OPERANDE(E[I+3]) ET E[I+4]=PARDROITE
  ALORS E[I]:=OP(E[I+2],E[I+1],E[I+3])
  SINON ERREUR3: Ecrire( " EXPRESSION INCORRECTE " ) ;
  K:=I ;
  POUR K:=K+1 TANTQUE E[K+4] ≠ OMEGA FAIRE
    DEBUT E[K]:=E[K+4] ; J:=K FIN ;
  E[J+1]:=OMEGA ; P:=P-1
  FIN LA SOUS EXPRESSION DE NIVEAU PMAX LA PLUS A GAUCHE
  A ETE EVALUEE, ET EST REMPLACÉE PAR SA VALEUR. LA
  PARTIE DROITE DE L'EXPRESSION EST RECOPÉE AVANT
  DE POURSUIVRE LE BALAYAGE ;
  FIN
SINON SI E[I] =PARDROITE ALORS P:=P-1 ;

```

FIN
FIN

3.1.3.2. Méthode récursive

Dans l'évaluation récursive, on recopie l'expression à évaluer dans une pile jusqu'à l'occurrence d'une parenthèse droite. On peut alors effectuer l'opération définie à l'intérieur de la paire associée dont les éléments sont effacés. Le résultat de l'opération est placé au sommet de la pile et l'évaluation continue.

DEBUT COMMENTAIRE EVALUATION D UNE EXPRESSION COMPLETEMENT
PARENTHESEE. METHODE RECURSIVE ;
ENTIER I, J, OMEGA, PARGAUCHE, PARDROITE ;
ENTIER TABLEAU E [1:100], P [1:10] ;

BOOLEEN PROCEDURE OPERATEUR(X) ;
DEBUT COMMENTAIRE CETTE PROCEDURE NE PREND LA VALEUR VRAI QUE
SI X EST UN OPERATEUR ;
 ° ° °
FIN ;

BOOLEEN PROCEDURE OPERANDE(X) ;
DEBUT COMMENTAIRE CETTE PROCEDURE NE PREND LA VALEUR VRAI QUE
SI X EST UN OPERANDE ;
 ° ° °
FIN ;

ENTIER PROCEDURE OP(X, Y, Z) ;
DEBUT COMMENTAIRE CETTE PROCEDURE DONNE LA VALEUR DU RESULTAT DE
L OPERATION DEFINIE PAR L OPERATEUR X ET LES OPERANDES Y ET Z ;
 ° ° °
FIN ;

° ° °

I := J := 0 ;
POUR I := I + 1 TANTQUE E [I] ≠ OMEGA FAIRE
SI E [I] = PARDROITE ALORS
DEBUT
SI OPERANDE(P [J]) ET OPERATEUR(P [J-1]) ET
OPERANDE(P [J-2]) ET P [J-3] = PARGAUCHE
ALORS P [J-3] := OP(P [J-1], P [J-2], P [J])
SINON ALLERA ERREUR2 ;
J := J-3 ;
FIN
SINON DEBUT J := J+1 ; P [J] := E [I] FIN ;
SI J ≠ 1 ALORS
ERREUR1 : ECRIRE (" ERREUR DE PARENTHESAGE ") ;
ALLERA FIN ;
ERREUR2 : ECRIRE (" EXPRESSION INCORRECTE ") ;
FIN :
FIN

3.1.4. Compilation

Dans les deux méthodes précédentes on peut remplacer l'évaluation directe par la génération d'un programme machine.

Dans cette génération il faut faire apparaître explicitement les adresses de mémoires de travail qui servent à ranger les résultats intermédiaires. Dans l'évaluation directe, ces mémoires de travail sont définies à l'intérieur de l'expression (évaluation itérative) ou dans la pile (évaluation récursive).

On remarque que le grand avantage de la méthode récursive réside dans une utilisation optimum de la zone de travail qui à chaque instant ne contient que les résultats intermédiaires des opérations exécutées et les éléments des opérations non encore complètement définies. Les recopies ne sont donc jamais nécessaires, les éléments se trouvant rangés dans la pile dans l'ordre strictement inverse de leur réutilisation.

Dans la compilation la gestion des mémoires de travail peut être réalisée de deux façons différentes :

- en utilisant une forme du programme objet dont l'exécution sera réalisée par l'intermédiaire d'une pile (notation post-fixée).

- en simulant l'exécution d'une expression au moment de la compilation pour définir les adresses de mémoires utilisées pour le rangement des résultats intermédiaires.

3.2 - Expressions sans parenthèses. Notations préfixées et post-fixées.

3.2.1. La notation préfixée

C'est une variante de la notation fonctionnelle.

En notation fonctionnelle une opération F sur deux variables a et b s'écrit $F(a,b)$: par exemple l'addition $a+b$ s'écrit $+(a,b)$: on peut remarquer que les parenthèses et la virgule ne sont pas strictement indispensables et avec des conventions convenables on peut écrire $+ ab$

Un logicien polonais, Jan LUKASIEWICZ, découvrit que si l'on écrit systématiquement les signes d'opération devant les opérands au lieu de les écrire entre les opérands comme dans la notation usuelle, les parenthèses deviennent inutiles pour définir l'ordre des opérations.

La notation préfixée est définie formellement par les règles suivantes :

1) Toute variable ou constante est une expression

2) - si θ_1 est un opérateur unaire et α une expression alors $\theta_1 \alpha$ est une expression.

- si θ_2 est un opérateur binaire, α et β deux expressions alors $\theta_2 \alpha \beta$ est une expression.

....

- si θ_n est un opérateur n-aire et $\alpha, \beta, \dots, \pi$ un ensemble de n expressions, alors $\theta_n \alpha \beta \dots \pi$ est une expression.

3) Il n'y a pas d'autres expressions

Remarque : cette notation tire son nom de la position de l'opérateur qui préfixe systématiquement les opérands qui lui sont associés. Elle est également connue sous le nom de notation polonaise.

Exemple 1 :

Identité de la déduction

p, q, r étant des propositions et \Rightarrow l'opérateur d'implication, l'identité s'écrit :

$$(p \Rightarrow q) \Rightarrow ((q \Rightarrow r) \Rightarrow (p \Rightarrow r))$$

et devient en notation préfixée

$$\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{\Rightarrow \Rightarrow p q}_{4} \Rightarrow}_{2} \Rightarrow}_{1} q r}_{3} \Rightarrow}_{5} p r}$$

Exemple 2 :

$$\begin{array}{c} (A+B) \times C + (D/(E+F)) \\ \times + AB + C/D + EF \\ \underbrace{\quad}_{+} \quad \underbrace{\quad}_{1} \\ \underbrace{\quad}_{2} \\ \underbrace{\quad}_{3} \\ \underbrace{\quad}_{5} \end{array}$$

Propriété caractéristique : à la différence de la notation algébrique usuelle qui est ambiguë lorsque l'ordre des opérations n'est pas explicitement défini, la notation préfixée définit complètement l'association des opérations.

Par exemple :

A + B + C + D peut signifier

$((A+B)+C)+D$	qui s'écrit en notation préfixée :	$+++ABCD$
$((A+B)+(C+D))$	"	$++AB+CD$
$((A+(B+C))+D)$	"	$++A+BCD$
$(A+((B+C)+D))$	"	$+A++BCD$
$(A+(B+(C+D)))$	"	$+A+B+CD$

Si l'on donne une règle d'associativité de gauche à droite pour l'écriture des expressions cela revient à choisir la première forme.

Remarque : la forme préfixée ne change pas l'ordre des opérandes car cela supposerait une commutativité des opérations

ainsi $a+b$ s'écrit $+ab$

et $b+a$ " $+ba$

3.2.2. La notation postfixée

Comme nous le verrons un peu plus loin, la transformation d'une expression en forme préfixée nécessite un balayage de cette expression de droite à gauche. Comme la plupart des notations utilisent un sens d'écriture et d'interprétation de gauche à droite, on utilise de préférence dans la compilation la forme postfixée dans laquelle les opérateurs sont écrits à droite des opérandes au lieu d'être écrits à gauche comme dans la forme préfixée.

La notation postfixée est définie formellement par les règles suivantes :

- 1) Toute variable ou constante est une expression
- 2) Si θ_1 est un opérateur unaire et α une expression alors $\alpha \theta_1$ est une expression

Si θ_2 est un opérateur binaire et α et β des expressions alors $\alpha\beta\theta_2$ est une expression

.....

Si θ_n est un opérateur n-aire et $\alpha, \beta, \dots, \pi$ un ensemble de n expression alors $\alpha\beta \dots \pi\theta_n$ est une expression

3) Il n'y a pas d'autres expressions

Propriété caractéristique

A tout élément E_i de rang i on associe un poids $p(E_i)$ égal à 1 pour les variables et les constantes, à 0 pour les opérateurs unaires, à -1 pour les opérateurs binaires, etc..., à $-(n-1)$ pour les opérateurs n-aires.

$$\text{Posons } P(k) = \sum_{i=1}^{i=k} p(E_i)$$

Pour toute expression postfixée, $P(k)$ a les propriétés suivantes :

- pour tout i compris entre 1 et k, $P(k) > 0$.
- pour une expression postfixée bien formée comportant n éléments :

$$P(n) = 1$$

Vérifier qu'une expression postfixée est bien formée revient donc à vérifier que la fonction $P(k)$ reste positive lorsqu'on balaye l'expression de gauche à droite et que sa valeur finale est égale à 1. L'intérêt de la notation postfixée réside dans la possibilité d'évaluer séquentiellement à l'aide d'une pile toute expression postfixée. Une expression postfixée peut être considérée comme un programme comportant deux sortes d'instructions:

- la lecture d'un opérande provoque son transfert au sommet de la pile.
- la lecture d'un opérateur provoque l'exécution de l'opération correspondante, les deux opérandes étant pris au sommet de la pile et le résultat rangé au sommet de la pile (après effacement des deux opérandes).

L'évaluation des expressions postfixées avec un évaluateur à pile est décrite par l'algorithme suivant :

DEBUT COMMENTAIRE EVALUATION D UNE EXPRESSION POSTFIXEE ;
ENTIER I, J, OMEGA ; ENTIER TABLEAU E[1:100], P[1:10] ;

BOOLEEN PROCEDURE OPERATEUR(X) ;
DEBUT COMMENTAIRE CETTE PROCEDURE NE PREND LA VALEUR VRAI QUE SI
X EST UN OPERATEUR ;

°°°
FIN ;

BOOLEEN PROCEDURE OPERANDE(X) ;
DEBUT COMMENTAIRE CETTE PROCEDURE NE PREND LA VALEUR VRAI QUE SI
X EST UN OPERANDE ;

°°°
FIN ;

ENTIER PROCEDURE OP(X, Y, Z) ;
DEBUT COMMENTAIRE CETTE PROCEDURE DONNE LA VALEUR DU RESULTAT
DE L OPERATION DEFINIE PAR L OPERATEUR X ET LES OPERANDES Y, Z ;

°°°
FIN ;

°°°°

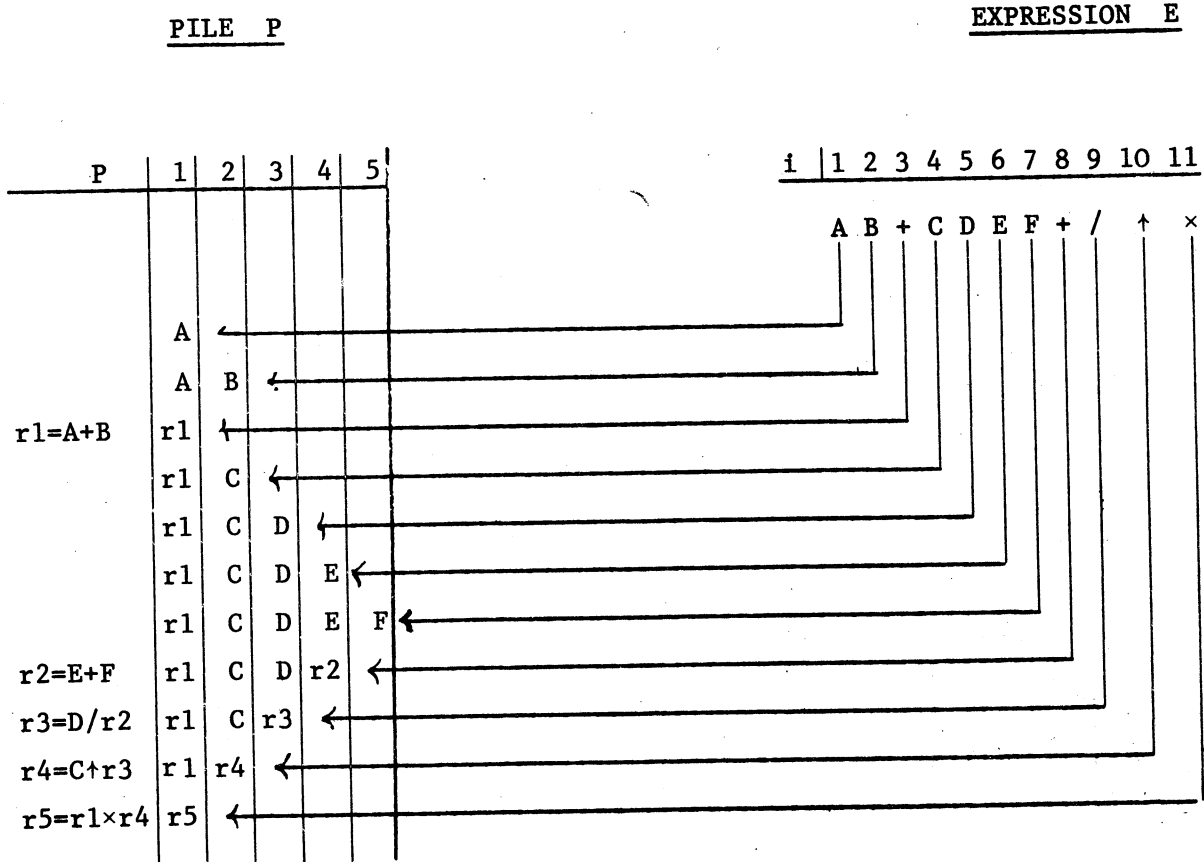
I:=0 ; J:=1 ;
A: I:=I+1 ;
SI E[I]=OMEGA ALORS ALLERA FIN ;
SI OPERANDE(E[I]) ALORS
DEBUT P[J]:=E[I] ; J:=J+1 ; ALLERA A FIN ;
SI OPERATEUR(E[I]) ALORS
DEBUT P[J-2]:=OP(E[I], P[J-2], P[J-1]) ;
J:=J-1 ; ALLERA A
FIN ;

FIN ;
FIN

L'expression

AB+CDEF+/ \uparrow x

sera évaluée comme suit



3.2.3. Transformation d'une expression complètement parenthésée en expression postfixée.

L'algorithme suivant transforme une expression complètement parenthésée $E_1 E_2 \dots E_n \omega$, en une expression postfixée, $S_1 S_2 \dots S_n$ avec vérification de la propriété caractéristique. La fonction poids est représentée par SIGMA.

DEBUT COMMENTAIRE TRANSFORMATION D UNE EXPRESSION COMPLETEMENT
PARENTHESEE EN EXPRESSION POSTFIXEE ;
ENTIER I, J, K, SIGMA, PARGAUCHE, PARDROITE, OMEGA, UN, BIN ;
ENTIER TABLEAU E, S[1:100], P[1:10] ;

BOOLEEN PROCEDURE OPERATEUR(X) ;
DEBUT ... FIN ;

BOOLEEN PROCEDURE OPERANDE(X) ;
DEBUT ... FIN ;

ENTIER PROCEDURE NOP(F) ;
DEBUT COMMENTAIRE CETTE PROCEDURE DONNE LE NOMBRE D OPERANDES
ASSOCIES A F, PAR EXEMPLE UN, BIN ;
FIN ;

...
...
...

I:=J:=K:=SIGMA:=0 ;
A: I:=I+1 ;
SI E[I]=OMEGA ALORS
ALLERA SI SIGMA=1 ALORS FIN SINON ERREUR ;
SI OPERANDE(E[I]) ALORS
DEBUT
SIGMA:=SIGMA+1 ;
K:=K+1 ; S[K]:=E[I] ;
ALLERA A
FIN ;
SI OPERATEUR(E[I]) ALORS
DEBUT
J:=J+1 ; P[J]:=E[I] ;
ALLERA A
FIN ;
SI E[I]=PARGAUCHE ALORS ALLERA A ;
SI E[I]=PARDROITE ALORS
DEBUT
SI NOP(P[J])=UN ALORS
ALLERA SI SIGMA < 0 ALORS ERREUR SINON B ;
SI NOP(P[J])=BIN ALORS
DEBUT
SI SIGMA < 1 ALORS ALLERA ERREUR ;
SIGMA:=SIGMA-1 ;
B: K:=K+1 ; S[K]:=P[J] ;
J:=J-1
FIN ;
ALLERA A
FIN ;
ERREUR: ECRIRE(" ERREUR ") ;
FIN ;
FIN

3.3 - Expressions non complètement parenthésées.

3.3.1. Portée des opérateurs.

Un opérateur unaire a une portée à droite qui est définie par l'opérande associé à cet opérateur.

Un opérateur binaire a une portée à gauche et une portée à droite définies par les opérandes associés.

Une fonction, $F(a,b,c)$ par exemple, n'a qu'une seule portée à droite définie par la liste d'arguments.

Si l'on écrit les opérateurs binaires sous forme fonctionnelle, on n'a qu'une seule portée à droite pour ces opérateurs.

Exemple :

$+ (a,b)$ est équivalent à $a+b$

L'opérande ou les opérandes associés à un opérateur peuvent être des expressions entre parenthèses ou non.

Exemples :

$((A + B) \times (C - (D/E)))$

(en notation complètement parenthésée)

$A + B \times C \uparrow D$

avec une priorité d'opérateurs : priorité (\uparrow) > priorité (\times) > priorité $(+)$

est équivalent à :

$(A + (B \times (C \uparrow D)))$

(en notation complètement parenthésée)

Remarque : les opérations unaires sont écrites sous forme fonctionnelle.

3.3.2. Transformation des expressions non complètement parenthésées

3.3.2.1. Passage à la forme complètement parenthésée

Méthode utilisée dans les premiers compilateurs Fortran [27]

Il y a 3 niveaux de priorités qui sont dans l'ordre décroissant :

0 * *	exponentiation
1 * , /	multiplication, division
2 + , -	addition, soustraction

et une règle d'associativité de gauche à droite pour les opérateurs additifs et multiplicatifs de même priorité, et de droite à gauche pour l'opérateur d'exponentiation.

On fait apparaître explicitement les niveaux de priorité à l'intérieur de l'expression selon les règles suivantes :

α et β étant des opérands simples ou complexes (sous-expressions)

$\alpha * * \beta$	reste inchangé
$\alpha * \beta$	est transformé en $(\alpha) * (\beta)$
$\alpha + \beta$	" " $((\alpha)) + ((\beta))$

De cette façon, la priorité des opérateurs est représentée par le nombre de paires de parenthèses de chaque côté de l'opérateur.

Exemple :

l'expression

$A + B * * C * (E + F)$

est transformée en :

$((A)) + ((B * * C) * (((E)) + ((F))))$

Transformation séquentielle de l'expression :

Soit

$$\alpha E_1 \text{-----} E_i \text{-----} E_n \omega$$

une expression à transformer, encadrée par les délimiteurs α et ω .

Nous désignons par :

φ la partie de l'expression déjà transformée (initialement $\varphi = \alpha$)

σ un élément E_i désignant un opérande simple ou une parenthèse gauche.

ψ la partie de l'expression non encore transformée qui suit un élément.

Les règles de transformation sont les suivantes :

$$R1 : \varphi \sigma \psi \rightarrow \varphi ((\sigma \psi \text{ si le dernier élément de } \varphi \text{ est } \alpha \text{ ou } ($$

$$R2 : \varphi \pm \sigma \psi \rightarrow \begin{cases} (\varphi_{\pm} ((\sigma \psi \text{ si le dernier élément de } \varphi \text{ est } \alpha \text{ ou } (\\ (\varphi)) \pm ((\sigma \psi \text{ dans les autres cas} \end{cases}$$

$$R3 : \varphi * \sigma \psi \rightarrow (\varphi) * (\sigma \psi$$

$$R4 : (\varphi) * * \sigma \psi \rightarrow (\varphi) * * \sigma \psi$$

$$R5 : \varphi) \psi \rightarrow \varphi))) \psi$$

$$R6 : \varphi \omega \rightarrow))) \omega$$

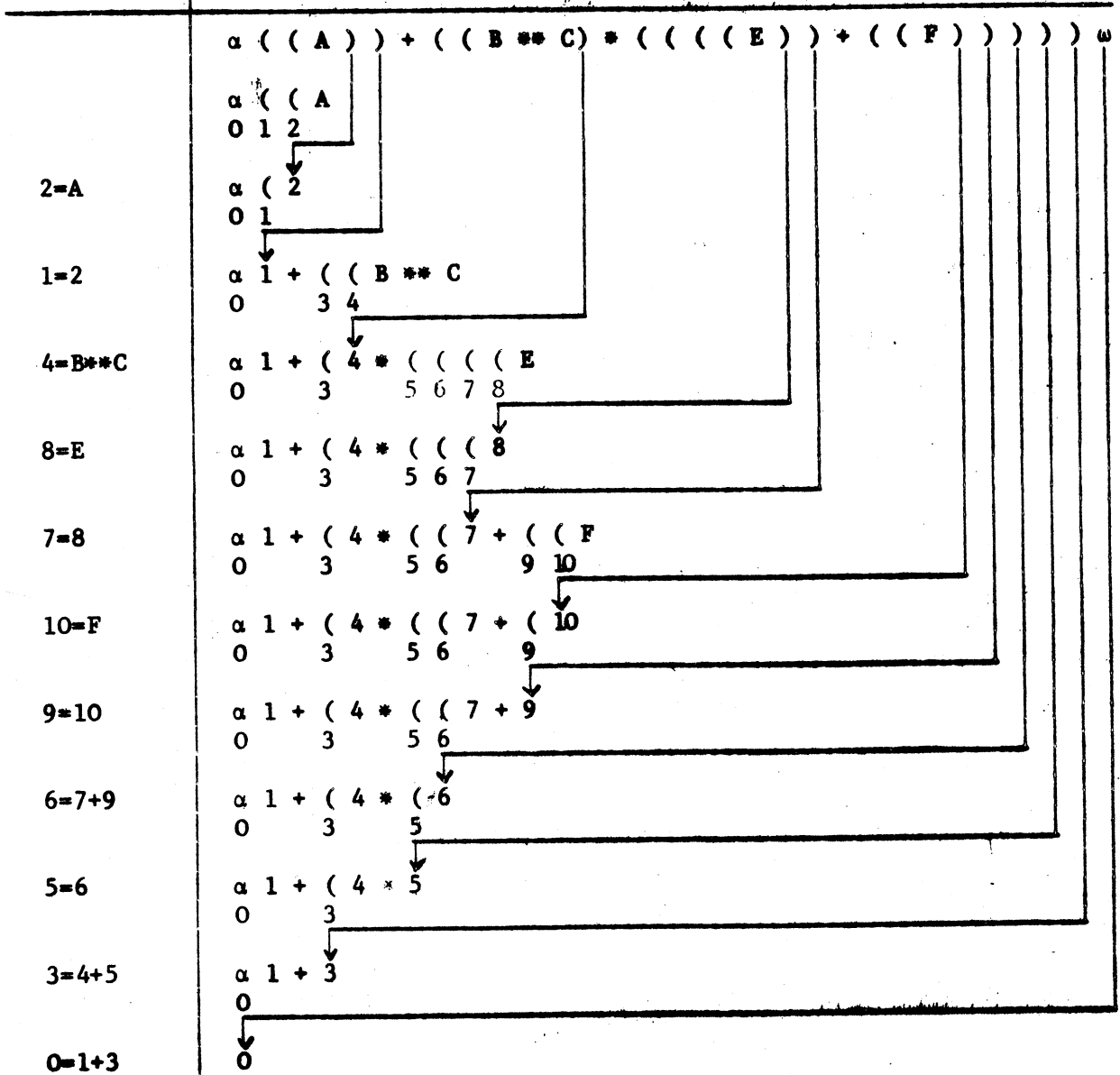
Nous remarquons que cette transformation ne change pas le couplage correct des parenthèses. Le premier cas de R2 correspond à l'opérateur additif unaire (+,-).

Réduction séquentielle de l'expression parenthésée.

On utilise une pile pour recopier les éléments de l'expression jusqu'à la première parenthèse droite. Les parenthèses gauches sont numérotées dans l'ordre croissant à partir de 1. A l'occurrence d'une parenthèse droite, la pile est vidée jusqu'à la rencontre d'une parenthèse gauche. On génère alors l'opération gauche et le numéro associé avec la parenthèse gauche remplace cette dernière dans la pile.

Exemple :

Génération Transformation de l'expression dans la pile P



Simplification de l'expression réduite.

Les opérations à un seul opérande provenant de l'insertion des parenthèses supplémentaires peuvent être éliminées après une substitution convenable.

Sur l'exemple précédent, après substitution et classement des opérations dans l'ordre décroissant on obtient :

$$\begin{aligned} 5 &= E + F \\ 4 &= B * * C \\ 3 &= 4 * 5 \\ 0 &= 1 + 3 \end{aligned}$$

qui décrit correctement les opérations nécessaires pour l'évaluation de l'expression donnée.

3.3.2.2. Passage à la forme postfixéeMéthode des priorités d'opérateurs

Considérons l'expression

$$\alpha \theta_1 \beta \theta_2 \gamma$$

ou α , β , et γ sont des opérands et θ_1 et θ_2 des opérateurs.

si l'on a priorité $(\theta_1) \geq$ priorité (θ_2) (1)

alors l'expression ci-dessus est équivalente à :

$$(\alpha \theta_1 \beta) \theta_2 \gamma$$

sinon l'expression est équivalente à :

$$\alpha \theta_1 (\beta \theta_2 \gamma)$$

Remarque : cette définition tient compte de la priorité de gauche à droite dans le cas de 2 opérateurs successifs de même priorité. Dans le premier cas, la forme postfixée associée est :

$$\alpha \beta \theta_1 \quad \gamma \theta_2$$

et dans le second cas :

$$\alpha \beta \gamma \theta_2 \quad \theta_1$$

Comme dans le cas de la notation complètement parenthésée les opérands seront recopiés directement dans la chaîne postfixée et les opérateurs transférés dans la pile, mais cette fois à chaque occurrence d'un opérande il faut comparer la priorité du dernier opérateur rencontré, qui est au sommet de la pile, avec l'opérateur suivant de l'expression à transformer.

Si la relation est vérifiée il faut transférer l'opérateur placé au sommet de la pile vers la chaîne postfixée et l'effacer de la pile : on recommence alors la comparaison de la priorité du nouvel opérateur au sommet de la pile avec l'opérateur suivant dans l'expression à transformer.

Si la relation n'est pas vérifiée on passe au traitement de l'élément suivant de l'expression.

Les opérateurs usuels ont une priorité bien définie qui est dans l'ordre décroissant la suivante :

$$\begin{array}{l} \uparrow \\ \times, / \\ +, - \end{array}$$

L'expression $A + B \times C \uparrow D$ sera ainsi transformée en $A B C D \uparrow \times +$

Priorité des parenthèses

Lorsqu'une opérande est une expression entre parenthèses, toutes les opérations définies à l'intérieur de la paire de parenthèses ont par définition une priorité supérieure à celle dont l'expression entre parenthèses est un opérande.

Il est commode de considérer les parenthèses comme des délimiteurs au même titre que les opérateurs arithmétiques et de leur affecter une priorité.

En attribuant à la parenthèse gauche une priorité inférieure à celle des opérateurs arithmétiques, les opérateurs précédents rangés dans la pile sont automatiquement protégés. La parenthèse gauche est rangée dans la pile comme un opérateur ordinaire et permettra le jeu normal des priorités pour les opérateurs contenus dans la paire de parenthèses associées. De manière à revenir à l'état existant avant l'occurrence de la parenthèse gauche, la parenthèse droite doit avoir une priorité inférieure à celles des opérateurs arithmétiques mais supérieure à celle de la parenthèse gauche. La lecture d'une parenthèse droite a pour effet de vérifier que le sommet de la pile est une parenthèse gauche et d'effacer cette parenthèse gauche, la transformation de l'opérande entre parenthèses étant alors terminée.

Les symboles $_$ et $_$ qui délimitent le début et la fin d'une expression jouent le même rôle que les parenthèses gauche et droite et sont affectés des mêmes priorités.

On a alors le tableau de priorité suivant :

(, $_$	0
) , $_$	1
+ , -	2
× , /	3
↑	4

L'algorithme de transformation en forme postfixée est décrit par le programme suivant :

DEBUT COMMENTAIRE TRANSFORMATION D UNE EXPRESSION PARENTHESEEE
 EN EXPRESSION POSTFIXEE, PAR PRIORITE DES OPERATEURS ;
ENTIER I, J, K, SIGMA, UN, BIN, OMEGA, PARGAUCHE, PARDROITE ;
ENTIER TABLEAU E, S(I:100), P(I:10) ;

BOOLEEN PROCEDURE OPERATEUR(X) ;
DEBUT ... FIN ;

BOOLEEN PROCEDURE OPERANDE(X) ;
DEBUT ... FIN ;

ENTIER PROCEDURE NOP(F) ;
DEBUT COMMENTAIRE CETTE PROCEDURE DONNE LE NOMBRE
 D OPERANDES ASSOCIES A F, PAR EXEMPLE UN, BIN ;
 ...
FIN ;

ENTIER PROCEDURE PRIORITE(X) ;
DEBUT COMMENTAIRE CETTE PROCEDURE DONNE LA PRIORITE DE
 L OPERATEUR X ;
 ...
FIN ;

...
 ...

I:=J:=K:=SIGMA:=0 ;
 A: I:=I+1 ;
SI E[I]=OMEGA ALORS ALLERA SI SIGMA=1 ALORS FIN
SINON ERREUR1 ;
SI OPERANDE(E[I]) ALORS
DEBUT
 SIGMA:=SIGMA+1 ;
 K:=K+1 ; S[K]:=E[I] ;
ALLERA COMPARAISON
FIN ;
SI OPERATEUR(E[I]) OU E[I]=PARGAUCHE ALORS
DEBUT
 J:=J+1 ; P[J]:=E[I] ;
ALLERA A
FIN ;
SI E[I]=PARDROITE ALORS
DEBUT
SI P[J] ≠ PARGAUCHE ALORS ALLERA ERREUR2 ;
 J:=J-1 ;
ALLERA COMPARAISON
FIN ;
 ERREUR3: ECRIRE(" ELEMENT INCONNU ") ; ALLERA FIN ;
 ERREUR2: ECRIRE(" ERREUR DE PARENTHESSES ") ; ALLERA FIN ;
 ERREUR1: ECRIRE(" EXPRESSION INCOMPLETE ") ; ALLERA FIN ;

COMPARAISON: SI J=0 ALORS ALLERA A ;
SI PRIORITE(P[J]) > PRIORITE(E[I+1]) ALORS

DEBUT

SI NOP(P[J])=UN ALORS
ALLERA SI SIGMA < 0 ALORS ERREUR1

SINON B ;

SI NOP(P[J])=BIN ALORS

DEBUT

SI SIGMA < 1 ALORS ALLERA ERREUR1 ;

SIGMA:=SIGMA-1 ;

B: K:=K+1 ; S[K]:=P[J] ;

J:=J-1

FIN ;

ALLERA COMPARAISON

FIN ;

ALLERA A ;

IN :

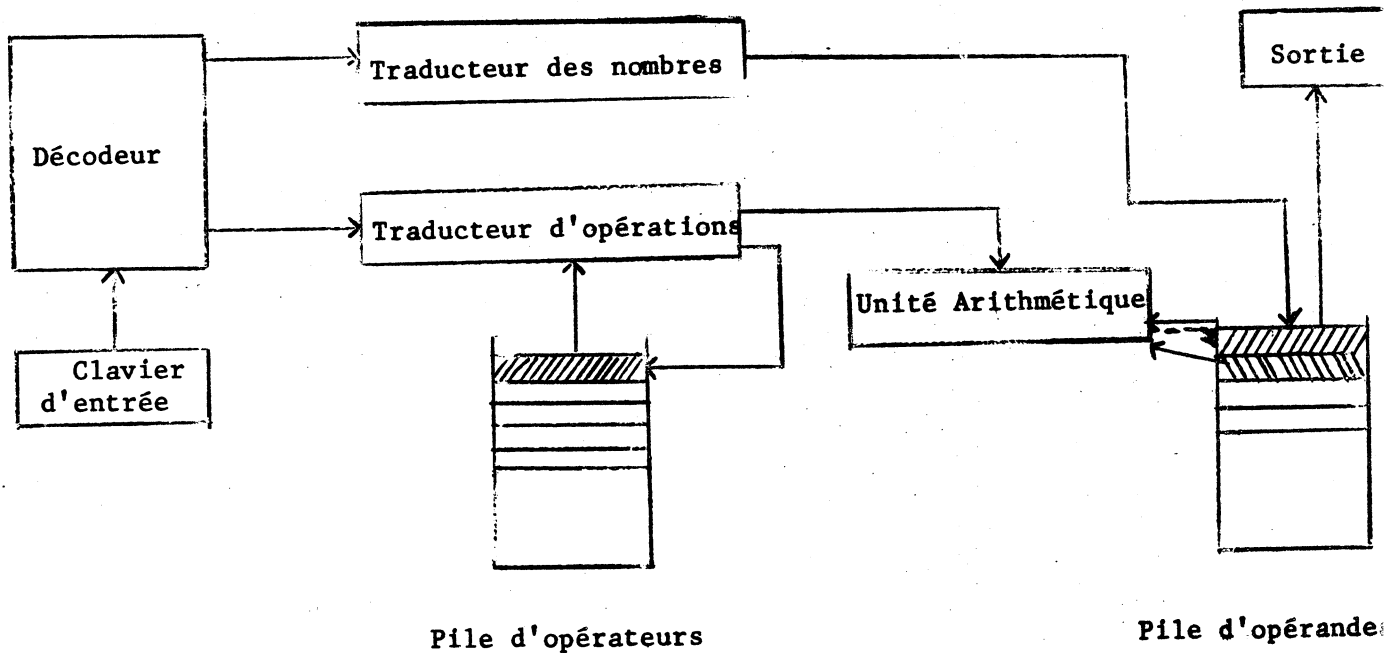
IN

3.3.3. Evaluation directe

Une expression non complètement parenthésée peut être évaluée directement en utilisant deux piles. En fait cette méthode d'évaluation n'est qu'une variante de la transformation postfixée.

Les opérandes sont copiés dans une pile d'opérandes au lieu d'être recopiés dans la chaîne postfixée ; le transfert éventuel de l'opérateur dans la chaîne postfixée est remplacé par l'exécution immédiate de l'opération correspondante.

BAUER ET SAMELSON ont construit une machine qui permet l'évaluation des expressions selon ce principe. [28]



Les expressions ne comportent que les opérateurs additifs et multiplicatifs avec priorité de l'opérateur additif sur l'opérateur multiplicatif, et les opérandes ne peuvent être que des nombres.

Les expressions sont entrées par le clavier. Les éléments (opérateurs binaires, nombres, parenthèses) sont transmis au décodeur qui envoie les nombres vers la pile d'opérandes et les opérateurs et parenthèses vers la pile d'opérations.

Le fonctionnement du traducteur d'opérations suppose que l'on a défini un ordre de priorité pour les opérateurs :

Tout opérateur transmis par le décodeur est comparé à l'opérateur rangé au sommet de la pile d'opérateurs.

Si l'opérateur de la pile a un ordre de priorité inférieur ou égal à celui de l'opérateur provenant du décodeur, cet opérateur de la pile est transmis à l'unité arithmétique afin d'exécuter l'opération correspondante (qui trouve ses opérandes au sommet de la pile d'opérandes et renvoie le résultat sur la pile). Cette comparaison et l'exécution d'une opération sont répétées tant que l'ordre de priorité de l'opérateur de la pile est inférieur ou égal à celui de l'opérateur provenant du décodeur.

Lorsque la comparaison n'est plus vérifiée, l'opérateur provenant du décodeur est envoyé sur la pile d'opérateurs.

Dans le cas où l'opérateur provenant du décodeur est une parenthèse droite, il n'est pas envoyé vers la pile d'opérateurs mais provoque l'effacement de la parenthèse gauche qui doit se trouver au sommet de la pile d'opérateurs. A la fin de l'évaluation, la valeur finale de l'expression se trouve au sommet de la pile d'opérandes.

3.3.4. Compilation

La compilation peut consister en un passage à la forme postfixée, dans ce cas le problème des mémoires de travail associé à l'évaluation est automatiquement résolu.

Si l'on génère un programme objet pour un calculateur ne comportant ou ne simulant pas un évaluateur à pile, l'optimisation des mémoires de travail est réalisée au moyen d'une transformation qui n'est autre qu'une simulation du processus d'évaluation directe.

L'algorithme de transformation dans un langage à 3 adresses est décrit par le programme suivant :

- m est l'adresse de la mémoire de travail utilisée
- P1 est la pile opérandes et P2 la pile opérateurs.

DEBUT COMMENTAIRE GENERATION DANS UN LANGAGE A TROIS ADRESSES ;
TIER I, K1, K2, S, OMEGA, PARGAUCHE, PARDROITE, M, UN, BIN ;
TIER TABLEAU E[0:100], P1, P2[1:20] ;

MODULEEN PROCEDURE OPERATEUR(X) ;
DEBUT ... FIN ;

MODULEEN PROCEDURE OPERANDE(X) ;
DEBUT ... FIN ;

MODULEEN PROCEDURE NOP(X) ;
DEBUT ... FIN ;

MODULEEN PROCEDURE PRIORITE(X) ;
DEBUT COMMENTAIRE CETTE PROCEDURE DONNE LA
 PRIORITE DE L OPERATEUR X ;
 ...
FIN ;

MODULEEN PROCEDURE TYPE(OPERANDE) ;
DEBUT COMMENTAIRE CETTE PROCEDURE PREND LA VALEUR M
 LORSQUE OPERANDE EST UN RESULTAT INTERMEDIAIRE ;
 ...
FIN ;

MODULEEN PROCEDURE CODE(OPERATEUR, OPERANDE1, OPERANDE2, RESULTAT) ;
DEBUT COMMENTAIRE CETTE PROCEDURE GENERE UN ORDRE A
 TROIS ADRESSES ;
 ...
FIN ;

I := K1 := K2 := S := 0 ;

I := I + 1 ;

SI E[I] = OMEGA ALORS
ALLERA SI K1 ≠ 1 OU K2 ≠ 0 ALORS ERREUR1
SINON FIN ;

SI OPERANDE(E[I]) ALORS
DEBUT
 K1 := K1 + 1 ; P1[K1] := E[I] ;
ALLERA COMPARAISON
FIN ;

SI OPERATEUR(E[I]) OU E[I] = PARGAUCHE ALORS
DEBUT
 K2 := K2 + 1 ; P2[K2] := E[I] ;
ALLERA A
FIN ;

SI E[I] = PARDROITE ALORS
DEBUT
SI P2[K2] ≠ PARGAUCHE ALORS

ALLERA ERREUR 2 ;


```

    K2:=K2-1 ;
    ALLERA COMPARAISON ;
    FIN ;
ERREUR3: ECRIRE( " ELEMENT INCONNU " ) ; ALLERA FIN ;
ERREUR2: ECRIRE( " ERREUR DE PARENTHESES " ) ; ALLERA FIN ;
ERREUR1: ECRIRE( " ERREUR SYNTAXIQUE " ) ; ALLERA FIN ;

COMPARAISON: SI K2=0 ALORS ALLERA A ;
    SI PRIORITE(P2[K2]) > PRIORITE(E[I+1]) ALORS
    DEBUT
        SI NOP(P2[K2])=UN ALORS
            DEBUT
                SI TYPE(P1[K1]) ≠ M ALORS S:=S+1 ;
                CODE(P2[K2],P1[K1],0,S) ;
                K2:=K2-1 ;
                P1[K1]:=M+S ;
                ALLERA COMPARAISON
                FIN ;
            SI NOP(P2[K2])=BIN ALORS
                DEBUT
                    SI TYPE(P1[K1])=M ET TYPE(P1[K1-1])=M
                    ALORS S:=S-1 ;
                    SI TYPE(P1[K1]) ≠ M
                    ET TYPE(P1[K1-1]) ≠ M ALORS S:=S+1 ;
                    CODE(P2[K2],P1[K1-1],P1[K1],S) ;
                    K2:=K2-1 ; K1:=K1-1 ;
                    P1[K1]:=M+S ;
                    ALLERA COMPARAISON
                    FIN ;
                FIN ;
            ALLERA A ;
    FIN ;
FIN ;
FIN

```

Remarque : Dans cet algorithme de transformation, la simulation de la zone de travail est réalisée en attachant à chaque opérande la qualité de variable symbolique ou de résultat intermédiaire.

Une telle simulation n'est plus possible si l'expression est évaluée à l'intérieur d'un sous-programme récursif, étant donné qu'il faut alors attacher à chaque résultat intermédiaire le niveau actuel de récursion.

Il est très rare que l'on connaisse à priori le nombre de récursions et le nombre de résultats intermédiaires attachés à chaque niveau : c'est pour cette raison que l'on préfère une transformation basée sur les propriétés de la notation postfixée. Sur les machines ne disposant pas d'évaluateur à pile, on simule le fonctionnement de cet évaluateur au moyen d'un interpréteur.

3.4 - Fonctions - Variables indicées - Expressions conditionnelles.

Les fonctions possèdent les mêmes propriétés d'imbrication que les expressions, en ce sens que chaque argument peut être lui-même une expression contenant d'autres fonctions.

Exemple : $F(A, G(B, H(C,D), E), I)$

Les variables indicées sont en fait un cas particulier de fonction et peuvent être traitées de façon analogue.

Les fonctions présentent la particularité de faire intervenir un délimiteur particulier, la virgule, qui joue le rôle de deux délimiteurs distincts.

Une virgule sépare deux expressions et est syntaxiquement équivalente à une paire de parenthèses opposées ")" et "(".

L'écriture $F(X,Y,Z)$ est équivalente à :

$F(X)(Y)(Z)$

En Algol, les expressions conditionnelles présentent la même particularité avec les délimiteurs alors et sinon qui jouent le rôle de deux parenthèses opposées.

L'expression conditionnelle

si EB alors E1 sinon E2

peut également s'écrire sous forme fonctionnelle

si (EB, E1, E2)

Cette solution semble meilleure que la notation utilisée dans le langage PL/I.

Dans ce langage l'équivalent de

if A then B else C peut s'écrire de deux façons :

- a) $A * B + (\neg A) * C$
- b) IF A THEN X=B ELSE X=C

La première forme présente deux inconvénients :

- l'expression A est évaluée deux fois
- du fait de cette double évaluation, quelques expressions ALGOL ne peuvent pas être transposées directement par exemple :

if B > 0 then SQRT(B) else SQRT(-B)

qui conduirait à écrire :

$(B > 0) * \text{SQRT}(B) + (\neg(B > 0)) * \text{SQRT}(-B)$

ou SQRT(-B) n'est pas défini dans le second produit

La deuxième forme nécessite l'introduction d'une variable fictive (X) et ne peut être utilisée dans les expressions conditionnelles utilisées comme paramètres effectifs, expressions en indice, bornes de tableaux, et rend difficile l'écriture d'expressions conditionnelles imbriquées.

Méthode de la double priorité

Dans ce qui suit nous ne considérons que les transformations relatives à la forme postfixée.

Le cas de la virgule et des délimiteurs alors et sinon est traité en définissant pour ces délimiteurs deux priorités distinctes :

- une priorité normale égale à 1 lorsque le délimiteur est considéré comme un élément de l'expression à transformer dans laquelle il joue le rôle d'une parenthèse droite.
- une priorité de pile égale à 0 lorsque le délimiteur est rangé dans la pile d'opérateurs, il joue dans ce cas le rôle d'une parenthèse gauche vis à vis des éléments de l'expression.

Forme postfixée des fonctions, des variables indicées et des expressions conditionnelles.

La fonction $F(X+Y, I/J, K+L \times M)$ sera transformée en

$$F X Y + I J / K L M \times + , , \dagger$$

Le nombre de virgules définit le nombre d'arguments moins un et le symbole \dagger est un opérateur permettant l'appel de la fonction.

La variable indicée $T [A+3, B-C]$ sera transformée en

$$T A 3 + B C - , \odot$$

Le symbole \odot est un opérateur permettant l'indexage.

L'expression conditionnelle

si A < B alors X+Y sinon si B=C alors X/Y sinon X-Y

sera représentée par :

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21

A B < si(8) X Y + alors(21) B C = si(16) X Y / alors(20) X Y - sinon sinon

Le délimiteur si est traduit par un saut conditionnel de manière à éviter l'évaluation de la première expression, le délimiteur alors est équivalent à un saut inconditionnel qui renvoie à la fin de l'expression lorsque l'on a évalué la première expression.

A cause de la transformation séquentielle, lorsque le délimiteur si est désempilé et transféré dans la chaîne postfixée, on doit le remplacer dans la pile par son adresse dans la chaîne postfixée. Ceci permet de le compléter avec l'adresse de saut à la détection du délimiteur alors qui lui est associé. La même chose se produit avec les délimiteurs alors et sinon.

Il convient de remarquer que l'imbrication possible des expressions conditionnelles nécessite une pile pour les adresses associées aux délimiteurs si et alors étant donné que l'on peut commencer la transformation d'une nouvelle expression conditionnelle avant que la précédente soit terminée.

3.5 - Instructions composées.

Les délimiteurs begin et end jouent le même rôle pour les instructions que les parenthèses "(" et ")" pour les expressions et permettent l'écriture et l'imbrication des instructions composées.

Ces délimiteurs begin et end ont la même priorité que les parenthèses d'expressions correspondantes.

Dans la compilation la principale différence entre instructions composées et blocs (autre les déclarations) concerne la portée des étiquettes.

3.6 - Structure de blocs.

La structure de blocs est une structure imbriquée qui présente une certaine analogie avec celle des expressions.

En effet, la structure de blocs est caractérisée par la durée de vie imbriquée des quantités : les identificateurs déclarés à l'intérieur d'un bloc sont locaux à ce bloc et n'ont pas d'existence à l'extérieur.

L'allocation dynamique des quantités consiste à considérer l'espace de mémoire utilisable comme une pile complexe : à l'entrée d'un bloc on réserve dans cette pile l'espace nécessaire au stockage des quantités locales, à la sortie de ce bloc, cet espace est libéré. A un instant donné de l'exécution d'un programme, on trouvera dans la pile uniquement les espaces de travail associés aux blocs effectivement commencés et non encore terminés. Ce système d'allocation optimise automatiquement l'utilisation de l'espace de mémoire utilisable : en effet deux blocs de même niveau utiliseront des espaces de travail ayant la même origine dans la pile. Cette pile d'évaluation des blocs est complexe dans le sens que cette pile ne contient pas uniquement des éléments accessibles par le sommet dans l'ordre strictement inverse de leur rangement. La structure de blocs nécessite que ce concept soit appliqué aux zones de travail, ces zones de travail comprenant en plus des emplacements réservés aux quantités locales une pile simple par l'évaluation des expressions.

3.7 - Structure de sous-programmes.

Une structure de sous-programmes est caractérisée par l'imbrication des

exécutions des sous-programmes qui la composent : chaque sous-programme pouvant appeler d'autres sous-programmes pendant son exécution sans limitation de niveau. Ce concept est si important en programmation que le mécanisme du rangement de l'adresse de retour à l'intérieur d'un registre attaché au sous-programme appelé est câblé sur la plupart des calculateurs.

Une telle solution n'est plus valable dans le cas des sous-programmes récursifs étant donné que chaque nouvel appel détruit automatiquement l'adresse de retour du niveau précédent.

En tenant compte de la nature imbriquée d'une telle structure, la solution consiste à ranger l'adresse de retour au sommet d'une pile extérieure à la structure de sous-programmes.

A l'entrée d'un sous-programme, l'adresse de retour est rangée au sommet de cette pile ; à la sortie l'adresse de retour est prélevée au sommet de cette pile. Si la pile des adresses de retour est câblée, le rangement et le prélèvement de l'adresse de retour à l'entrée et à la sortie d'un sous-programme peuvent être réalisés automatiquement au moyen d'instructions machines.

Si la pile n'est pas câblée, l'entrée et la sortie des sous-programmes sont des pseudo-instructions exécutées par l'interpréteur qui gère la pile des adresses de retour.

Un autre point important est relatif aux espaces de travail nécessaire à l'exécution des sous-programmes. Pour une structure de sous-programmes non récursifs, les espaces de travail peuvent être propres à chacun des sous-programmes. Cela n'est plus possible avec des sous-programmes récursifs pour lesquels les espaces de travail doivent être définis à l'extérieur et correctement imbriqués.

On peut alors remarquer que même pour des sous-programmes non récursifs un tel schéma optimise automatiquement l'utilisation de l'espace de travail relatif à l'exécution d'une structure de sous-programmes.

En effet, si tous les sous-programmes travaillent sur une zone commune extérieure à chacun d'eux, l'utilisation de cette zone de travail se trouve automatiquement optimisée étant donné que dans une structure de sous-programmes, deux sous-programmes ne sont jamais exécutés simultanément, excepté le cas particulier des sous-programmes réentrants.

DIJKSTRA [29] a montré qu'une telle organisation est particulièrement intéressante pour ALGOL dans laquelle se superposent une structure statique de blocs et une structure dynamique de sous-programmes. Une seule zone de travail organisée en pile complexe permet l'évaluation des programmes.

CHAPITRE II

MACROEVALUATION1 - STRUCTURE DU PROCESSUS DE COMPILATION.

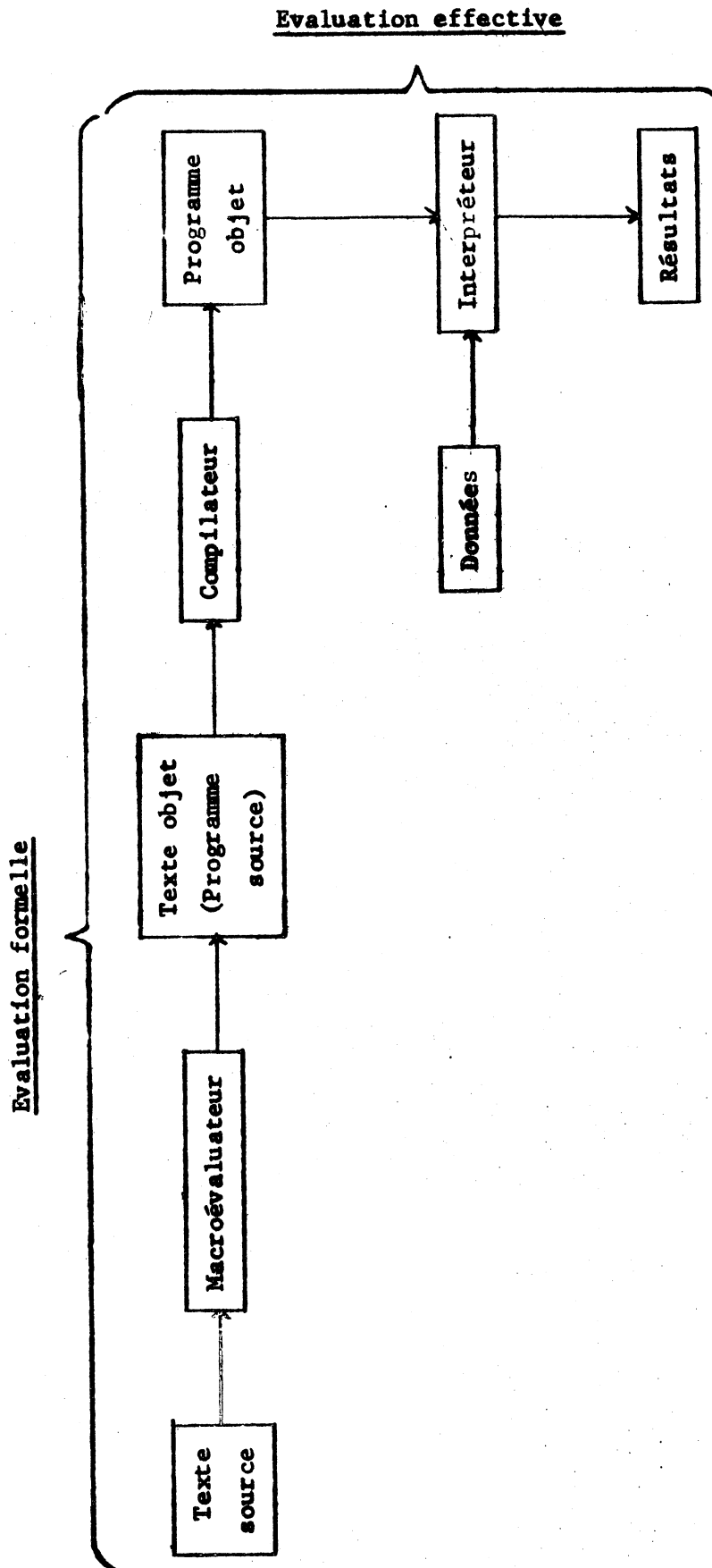
La compilation d'un programme comporte trois phases distinctes :

a) macroévaluation pendant laquelle une partie du texte source est considérée comme une donnée modifiable au moyen des macroinstructions. Le résultat de la macroévaluation est un texte objet dans lequel toutes les macrodénominations ont disparu et qui constitue le programme source proprement dit.

b) évaluation syntaxique pendant laquelle on réalise l'analyse syntaxique complète d'un programme source. Le programme source est transformé en un arbre syntaxique en utilisant les règles syntaxiques du langage source.

c) évaluation sémantique qui permet la transformation d'un programme source dont l'évaluation syntaxique est correcte en un programme objet sémantiquement équivalent. L'évaluation sémantique nécessite donc une description du langage objet et les règles de correspondance entre éléments du langage source et éléments du langage objet. Ces trois phases constituent l'évaluation formelle d'un programme source dont le résultat est un programme objet.

Pour la plupart des langages algorithmiques actuels, caractérisés par de nombreuses propriétés dynamiques, l'évaluation effective d'un programme objet nécessite la mise en oeuvre d'un mécanisme d'interprétation pendant l'exécution.



La réalisation de tels programmes d'interprétation représente une partie importante du travail nécessaire à la mise en oeuvre d'un compilateur.

Le schéma général du processus de compilation peut se représenter de la façon suivante : (II - 2).

2 - MACROEVALUATION.

On peut définir la macroévaluation comme la transformation préliminaire d'un texte source contenant des éléments dont la signification n'est pas définie au niveau du langage source en un texte source proprement dit dont tous les éléments sont définis au niveau du langage source.

La macrosubstitution qui est utilisée depuis de nombreuses années pour les langages d'assembleur (macro-assemblage) commence seulement à être étudiée pour les langages algorithmiques.

La plupart des langages algorithmiques utilisent des macro-dénotations rudimentaires pour l'insertion de commentaires à l'intérieur des programmes source.

En Algol 60, par exemple, les commentaires peuvent être considérés comme des macrodénotations dont la définition est donnée une fois pour toute.

Les règles de substitution sont les suivantes :

La suite de symboles de base

; comment < toute suite de symboles de base ne contenant pas >;
begin comment < toute suite ne contenant pas >;
end < toute séquence ne contenant pas end ou ; ou else>
) < chaîne de lettres > : (
 (dans une liste de paramètres formels ou effectifs)

est équivalente à

;
begin
end
 ,

Même avec des règles de substitution aussi simples on peut avoir des ambiguïtés si l'on ne précise pas l'ordre d'application des règles de substitution.

Par exemple le texte

end begin comment A ; B ;

peut être transformé dans l'une des deux formes :

end begin comment A ; B ;

begin (2^e règle)

end ; (3^e règle)

end begin comment A ; B ;

end (3^e règle)

end ; B ;

Dans le rapport révisé d'Algol 60, on précise que la première structure de commentaire rencontrée, en lisant de la gauche vers la droite a priorité pour la substitution sur d'autres structures contenues dans la suite. Cette règle conduit à la deuxième forme pour l'exemple ci-dessus.

Cet exemple de macrosubstitution est très particulier étant donné qu'il consiste seulement à supprimer des suites de caractères dans les trois premières règles et à une équivalence simple dans la quatrième.

Le principal intérêt des macrosubstitutions réside dans les possibilités de changer la syntaxe d'un langage au moyen des macrodéfinitions.

L'utilisation des macrosubstitutions soulève deux problèmes importants

Le macrolangage doit-il avoir la même syntaxe que le langage de base ?

Cette propriété semble souhaitable aussi bien pour les programmeurs que pour les constructeurs de compilateurs qui peuvent utiliser les mêmes concepts pour le macrolangage et le langage de base dans l'écriture et la compilation des programmes. Toutefois, il convient de remarquer que le macrolangage nécessite quelques symboles de base supplémentaires pour permettre une discrimination des éléments appartenant au macrolangage et au langage de base.

Le macroévaluateur doit-il se réduire à un programme de traitement élémentaire des symboles ou être un véritable analyseur syntaxique capable de réaliser les substitutions au niveau des éléments syntaxiques ?

Nous allons décrire très brièvement trois systèmes différents de macro-substitutions afin de donner une idée des diverses approches possibles.

a) Macrosubstitution de PL/1 [8]

Le macrolangage est un sous-ensemble du langage de base sauf pour les instructions "ACTIVATE", "DEACTIVATE", "INCLUDE" et le signe spécial "%" (utilisé pour préfixer les macroinstructions). Les macroinstructions peuvent apparaître n'importe où à l'intérieur du texte source.

Fonctionnement du macroprocesseur.

Le fonctionnement du macroprocesseur est défini par l'algorithme de transformation du texte source en texte objet :

le texte source est balayé séquentiellement caractère par caractères :

- s'il n'y a pas de macroinstructions le texte objet est identique au texte source.

- Lorsqu'on rencontre une macroinstruction, celle-ci est immédiatement exécutée. L'exécution d'une macroinstruction peut modifier le balayage séquentiel de deux façons :

- l'exécution d'une macroinstruction peut obliger le macroévaluateur à continuer le balayage à partir d'un autre point à l'intérieur du texte source.

- l'exécution d'une macroinstruction peut indiquer que pour toutes les occurrences d'un certain identificateur du texte source, la valeur de cet identificateur et non l'identificateur lui-même doit être insérée dans le texte objet. Les macroinstructions ne sont jamais insérées dans le texte objet mais remplacées par un blanc. La macrosubstitution est terminée lorsqu'on essaie de balayer au delà du dernier caractère du texte source.

Macroinstructions de PL/1.

La macroinstruction de déclaration permet la définition des macrovariables et des noms de macroprocédures.

La macroinstruction d'affectation permet d'évaluer une macro-expression et d'affecter sa valeur à une macrovariable.

Les macroinstructions d'activation et de désactivation permettent de contrôler la portée des macrovariables.

La macroinstruction "aller a" permet de continuer le balayage au point défini par une macroétiquette.

La macroinstruction vide sert à insérer des macroétiquettes à l'intérieur du texte source.

La macroinstruction "si" permet de contrôler le balayage en fonction de la valeur d'une macroexpression.



La macroinstruction de groupe permet de contrôler le balayage des parties répétitives du texte source.

La macroinstruction d'inclusion sert à incorporer des chaînes de texte externes à l'intérieur du texte source en cours de traitement.

Enfin, les macroinstructions procédure (déclaration et appel) permettent l'utilisation du concept de procédure au niveau du macrolangage.

Remarque importante : il n'y a pas de possibilité récursive dans l'écriture des macros, c'est-à-dire qu'une macroinstruction ne peut jamais contenir une autre macroinstruction.

Exemples de transformations.

<u>Textes source</u>	<u>Macrosubstitution</u>	<u>Textes objet</u>
<pre>% DECLARE I FIXED ; % DO I = 1 TO 10 ; Z(I) = X(I) + Y(I) ; % END</pre>		<pre>Z(1) = X(1) + Y(1) ; Z(2) = X(2) + Y(2) ; Z(10) = X(10) + Y(10) ;</pre>
<pre>% DECLARE I FIXED, T CHARACTER ; % DEACTIVATE I ; % I = 15 ; % T = 'A(I)' ; S = I * T * 3 ; % I = I + 5 ; % ACTIVATE I ; % DEACTIVATE T ; R = I * T * 2 ;</pre>		<pre>S = I * A(I) * 3 ; R = 20 * T * 2 ;</pre>


```

% DECLARE A CHARACTER,
  VALUE ENTRY (CHARACTER, FIXED) RETURNS CHARACTER ;
DECLARE (Z(10), Q) FIXED ;
% A = 'Z' ;
% VALUE : PROCEDURE (ARG1, ARG2) CHARACTER ;
  DECLARE ARG1 CHARACTER, ARG2 FIXED ;
  RETURN (ARG1 || '(' ARG2 || ')') ;
% END VALUE ;
Q = 6 + VALUE (A, 3) ;

```

} DECLARE (Z(10), Q) FIXED ;
} Q = 6 + Z(3) ;

b) Macroalgot [30]

H. Leroy a proposé l'introduction d'une macrosubstitution en Algol au niveau de l'ensemble des entités syntaxiques. La syntaxe de macroalgot est alors identique à celle d'algol exception faite de quelques macrodénotations.

La sémantique de macroalgot est définie par un algorithme de transformation d'un macroprogramme source en un macroprogramme objet. Après un certain nombre de macrosubstitutions un macroprogramme objet peut représenter un programme en Algol de base.

La différence essentielle avec la macrosubstitution de PL/1 réside dans la récurtivité de la macrosubstitution : alors que dans PL/1 un texte source produit toujours un texte objet ne contenant pas de macrodénotations, dans macroalgot, un texte objet peut contenir des macrodénotations et doit être transformé jusqu'à la disparition complète de toute macrodénotation afin de pouvoir être compilé.

Nous donnons maintenant un bref résumé des possibilités de macroalgot.

On peut définir quelques-unes des variables déclarées dans un bloc comme des macrovariables au moyen de la dénotation macro. Ceci signifie qu'on leur affectera des valeurs pendant la macrosubstitution et qu'elles n'apparaîtront plus dans le texte objet.

De la même façon, on peut définir des macroprocédures, ce qui signifie que chaque appel d'une de ces procédures sera remplacé dans le texte objet par une copie du corps de procédure avec substitution des paramètres effectifs aux paramètres formels et que ces identificateurs de procédure n'apparaîtront plus dans le texte objet.

On peut définir une position d'indice comme une position de macro-
indice au moyen d'une macrodénotation associée avec la paire de bornes correspon-
dante. Par exemple, un tableau à deux dimensions déclaré avec un macroindice sera
remplacé dans le texte objet par un ensemble de tableaux à 1 dimension, le nombre
de tableaux étant égal au nombre d'éléments correspondant au macroindice.

Une instruction conditionnelle peut contenir une macrodénotation pour
indiquer qu'elle doit être remplacée par une des deux instructions qui la compo-
sent.

Si la variable contrôlée d'une instruction "pour" est une macrovariable, l'instruc-
tion "pour" est remplacée par une instruction composée constituée par les instruc-
tions générées provenant de l'instruction qui suit faire.

Etant donné que le macroévaluateur doit pouvoir affecter des valeurs
aux macrovariables, il doit nécessairement pouvoir exécuter des instructions
d'affectation. En fait, n'importe quelle catégorie d'instructions doit pouvoir
être exécutée par le macroprocesseur. On utilise pour cela la macrodénotation
valeur.

Enfin une macrodénotation est nécessaire pour indiquer que la macro-
substitution d'un élément syntaxique se réduit à une copie dans le texte objet.

Il y a donc 3 macrodénotations distinctes :

macro qui permet de dénoter les macroidentificateurs, les positions de
macroindices et les macroinstructions conditionnelles.

value pour indiquer une évaluation ou une exécution complète.
quote pour indiquer la copie littérale d'un élément syntaxique.

Exemples :

Instruction aller a

Syntaxe

<instruction aller a> ::= goto <expression de désignation> |
value goto <expression de désignation>

Sémantique

Pour une instruction "aller a" de la première forme, il y a macro-évaluation de l'expression de désignation et l'instruction objet est une instruction "aller a" avec l'expression de désignation objet résultant de la substitution.

Pour une instruction "allera" de la deuxième forme, il y a évaluation de l'expression de désignation, l'instruction objet est vide et la macro-substitution continue au point du texte source défini par l'étiquette (valeur de l'expression de désignation).

c) Redéfinition des opérateurs : ("Overloading")

Une des tendances importantes des nouveaux langages de programmation actuels est la possibilité de définir dynamiquement de nouveaux types de quantités. Les structures dynamiques de PL/1 et les enregistrements d'Algol X sont des exemples de cette nouvelle tendance.

Ayant défini de nouveaux types de quantités, le problème se pose alors de définir des opérations pour ces nouveaux types en utilisant les mêmes symboles opératoires que pour les quantités de type élémentaire.

On peut choisir de réaliser ces définitions en utilisant le concept de procédure qui permet un mécanisme simple de définition dynamique, la portée de la définition étant celle du bloc dans lequel la procédure associée est déclarée.

La définition dynamique au moyen de procédures présente néanmoins l'inconvénient d'une faible efficacité au niveau de l'exécution : le concept de procédure est très puissant et très élégant mais son utilisation en toute généralité coûte très cher en temps d'exécution.

Une autre façon de définir dynamiquement des opérations consiste à utiliser la macrosubstitution qui présente dans ce cas deux avantages :

- on peut utiliser les mêmes opérateurs pour définir des opérations sur des quantités de types différents : l'opérateur '+' par exemple peut être redéfini pour l'addition de quantités de type complexe, sa définition courante correspondant à l'addition des entiers ou des réels. (Cette propriété est couramment utilisée au niveau des macroassembleurs pour redéfinir, par exemple, des opérations machine).

- on évite l'utilisation du mécanisme des procédures étant donné que la substitution est réalisée préalablement à la compilation.

Exemples de notations syntaxiques pour la redéfinition des opérateurs

Cas de l'addition de quantité de type complexe

(1) en utilisant le concept de référence [31]

```
référence (complex) overload (u+v) ; référence (complex) u, v ;
complex (real part (u) + real part (v) ,
        imag part (u) + imag part (v) ) ;
```

(2) en utilisant une extension d'Algol 60 [32]

complex means array [1:2] ;

complex a + complex b := complex complex^{(a} [1] + b [1], a [2] + b [2])' ;

d) Macroextensions syntaxiques [33]

Une autre possibilité consiste à décrire séparément la macrostructure qui définit la syntaxe du texte source et la macrodéfinition qui correspond à la sémantique exprimée en termes d'entités plus simples.

La différence essentielle avec le macroassemblage au niveau du langage machine symbolique tient ici à la double substitution syntaxique et sémantique alors que dans le premier cas on a seulement substitution formelle au niveau d'un appel de procédure.

Par exemple en MAP la macrodéfinition suivante :

```
MACRO  ALPHA (X, Y, Z)
      CLA  X
      ADD  Y
      STO  Z
      END  MACRO
```

associée à la macroinstruction

```
ALPHA (A, B, C)
```

créera le texte

```
CLA  A
ADD  B
STO  C
```

au moyen d'une simple substitution paramètre effectif, paramètre formel.

Il n'y a pas création d'une structure syntaxique nouvelle étant donné que l'appel des macros est toujours réalisé sous forme fonctionnelle.

L'idée de macrostructure consiste à définir syntaxiquement une nouvelle entité en termes d'entités plus simples.

Par exemple la macrostructure :

macro pour variable := expression pas expression jusqua expression faire instruction

définit une nouvelle unité syntaxique (instruction "pour") en fonction de constantes métalinguistiques (symboles de base : pour, pas, jusqua, faire et de variables métalinguistiques (variable, expression et instruction).

La macrodéfinition associée à la macrostructure précédente est alors :
(sémantique de l'instruction "pour" de la forme pas - jusqu'à en Algol 60)

```

définition début  ~1 := ~2
                L1 : si (~1 - ~4) × signe (~3) < 0 alors
                    début
                        ~5 ;
                        ~1 := ~1 + ~3 ;
                        allera L1
                    fin
                fin

```

Les variables métalinguistiques de la macrostructure jouent le rôle de liste de paramètres formels ; au lieu d'être séparés par des virgules ils sont séparés par des symboles de base. Etant donné la nature particulière de ces paramètres formels (variables métalinguistiques), ils sont désignés dans la macrodéfinition associée au moyen des symboles ~1 à ~5.

Ainsi ~ 2 , ~ 3 , ~ 4 sont syntaxiquement équivalents à "expression" mais correspondent à des expressions différentes dans la macrodéfinition (valeur initiale, pas, valeur finale).

Dans la phase de macroévaluation, lorsqu'on rencontre une macroinstruction telle que :

pour q := r pas s jusqua t faire A[q] := B[q],

il y a d'abord macroévaluation syntaxique pour vérifier que cette macroinstruction est syntaxiquement correcte c'est-à-dire dans ce cas que :

q est une variable
r, s et t sont des expressions
A[q] := B[q] est une instruction

puis macroévaluation sémantique qui produit par substitution :

```

debut
  q := r
  L1 : si (q-t) × signe (s) < alors
    debut
      A[q] := B[q] ;
      q := q+s ;
      allera L1
    fin
  fin

```

L'intérêt d'une telle macroextension syntaxique réside dans la possibilité de définir dynamiquement de nouvelles entités syntaxiques autrement que par le moyen des déclarations de procédure.

La sémantique de l'instruction "pour" précédente pourrait se définir en Algol de la façon suivante :

```

procédure POUR(V, A, B, C, INST) ; valeur A, B, C ;
      reel V, A, B, C ; procédure INST ;
debut
      V := A ;
      L1 : si (V-C) × signe (B) < alors
          debut
              INST ;
              V := -V + B ;
              allera L1
          fin
      fin ;

```

Dans ce cas, on ne définit pas de nouvelle entité syntaxique (celle de procédure existant déjà).

La macroinstruction précédente est alors remplacée par l'appel de procédure POUR(q, r, s, t, AFFECT)

avec quelque part la déclaration de procédure

```

procédure AFFECT ; A[q] := B[q] ;

```

Les macroextensions présentent en définitive deux avantages importants :

- création de nouvelles entités syntaxiques utilisées dynamiquement, ce qui permet une grande souplesse d'expression au niveau du programme.
- programme objet beaucoup plus performant étant donné l'utilisation beaucoup plus limitée du concept de procédure dont la mise en oeuvre coûte chère dans l'évaluation des programmes objet : les transmissions de valeurs sont remplacées par des copies de formes.

C H A P I T R E III

EVALUATION SYNTAXIQUE

Une des originalités du rapport ALGOL est certainement l'utilisation d'un formalisme simple et concis pour la description syntaxique. Il convient de remarquer toutefois que ce formalisme décrit essentiellement un mécanisme de génération des programmes à partir :

- de symboles terminaux (symboles de base)
- de symboles non terminaux (variables métalinguistiques)
- d'un ensemble de règles de dérivation (formules métalinguistiques)
- d'un symbole non terminal particulier (axiome) désigné en général par "programme"

Pour construire (générer) syntaxiquement un programme il suffit de partir du symbole non terminal <programme> et d'appliquer un certain nombre de fois les règles de dérivation jusqu'à obtenir une chaîne ne comportant que des symboles de base.

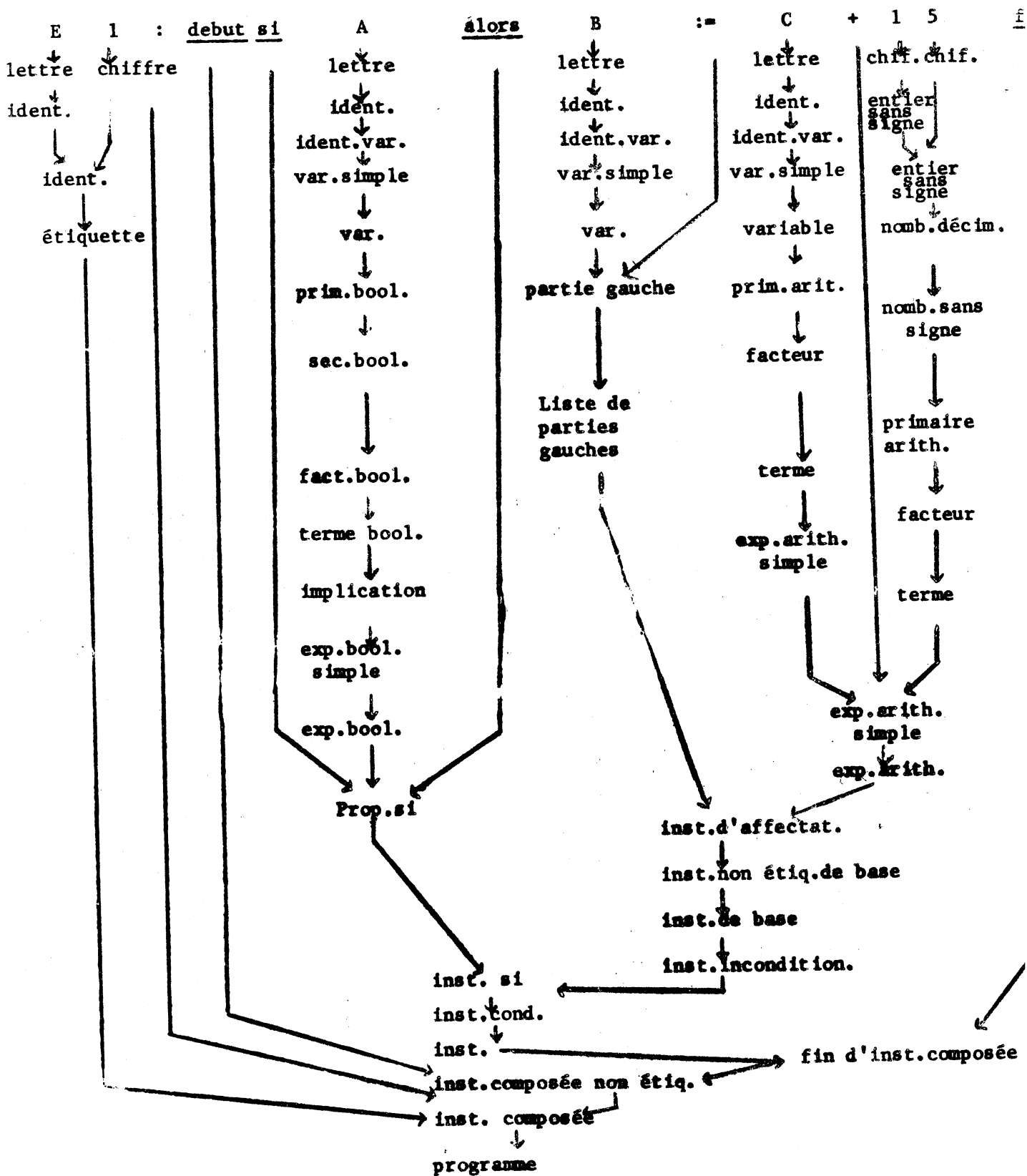
Dans la compilation, où l'on a des programmes déjà construits, on est essentiellement intéressé par un mécanisme inverse, à savoir un mécanisme de reconnaissance ou d'analyse.

Exemple : considérons en ALGOL, la suite de symboles de base

E1 : début si A alors B := C + 15 fin

Effectuer l'analyse syntaxique de cette suite de symboles consiste à construire l'arbre de décomposition syntaxique associé à cette suite.

En utilisant les règles syntaxiques d'ALGOL 60, on obtient pour cet exemple



Ce schéma met clairement en évidence la dualité génération - reconnaissance : alors que dans la génération on part de <programme> pour aboutir à une suite de symboles terminaux, en construisant l'arbre en partant de la base pour aller vers les extrémités, dans la reconnaissance on construit l'arbre en sens inverse partant des extrémités pour aboutir à la base.

1 - ANALYSE LEXICOGRAPHIQUE.

Sur l'exemple précédent on peut remarquer assez facilement qu'il est long et fastidieux de réaliser l'analyse syntaxique à un niveau trop bas : ainsi les lettres et les chiffres ne jouent aucun rôle en eux-mêmes mais servent uniquement à former des identificateurs et des nombres. Il est donc naturel d'introduire la notion d'unité syntaxique qui est constituée par un identificateur, un délimiteur, un nombre sans signe, une valeur logique, une chaîne, un nombre déterminé de caractères pour des procédures en code. La plupart des algorithmes d'analyse syntaxique sont utilisés au niveau des unités syntaxiques ainsi définies.

Règles liées de la description syntaxique.

Un certain nombre de règles d'écriture échappent actuellement à toute formalisation. Dans le cas d'ALGOL on peut citer par exemple les règles concernant la structure de blocs (déclarations), la correspondance entre paramètres formels et effectifs des procédures quant à leur nombre, et leur nature, la correspondance entre expressions en indice d'une variable indicée et paires de bornes dans la déclaration de tableau associée quant à leur nombre et leur concordance.

L'exemple ci-dessus bien que conduisant à une analyse syntaxique correcte n'est cependant pas un programme correct au sens ALGOL (absence de déclarations pour A,B,C et d'affectation de valeurs à A et C).

Il y a donc une différence importante entre l'analyse syntaxique réalisée au moyen des règles de syntaxe dites libres qui sont effectivement formalisables et l'évaluation syntaxique complète qui nécessite que toutes les règles de syntaxe, libres ou liées, aient été correctement utilisées dans l'écriture des programmes source.

Alors qu'il est en général assez facile de tenir compte des règles concernant la structure de blocs dans l'analyse lexicographique ou syntaxique, la compilation séquentielle d'un programme a comme conséquence la compilation indépendante et séparée des déclarations et des instructions. Cette particularité rend généralement impossible la vérification des règles de correspondance concernant les paramètres effectifs et formels, les variables indicées et les déclarations de tableaux pendant la compilation. Ces vérifications sont néanmoins possibles pendant l'exécution.

Analyse lexicographique et édition.

L'analyse lexicographique peut être réalisée dans une phase préliminaire indépendante ou au contraire au moyen d'un sous-programme appelé à l'intérieur de la phase d'analyse syntaxique toutes les fois qu'on a besoin d'une nouvelle unité syntaxique.

On réalise en général dans une phase "édition" un certain nombre d'opérations préliminaires permettant une transformation des programmes source qui facilite les traitements ultérieurs.

L'édition réalise les opérations suivantes :

- lecture des caractères du programme source en tenant compte des conventions de représentation machine particulières et des supports d'entrée
- reconnaissance et délimitation des unités syntaxiques
- construction des tables de nombres réels et entiers qui seront incorporées directement dans le programme objet

- rangement des chaînes et des corps de procédure en code
- constitution de la table des identificateurs qui permet l'analyse de la structure de blocs. Le cas des étiquettes qui ne sont pas déclarées en tête de bloc ne constitue plus un cas particulier une fois que les étiquettes sont rangées dans cette table.
- élimination des commentaires de diverses natures : après les symboles de base commentaire et fin et à l'intérieur des listes de paramètres effectifs ou formels.

A la fin de l'édition, chaque unité syntaxique est représentée à l'aide d'un descripteur indiquant :

- sa nature (nombre, identificateur, délimiteur, etc).
- sa valeur (adresse dans une table, numéro d'ordre dans la liste des délimiteurs, ...)

Lorsque l'édition est réalisée dans une phase préliminaire le résultat est donc la suite de descripteurs associée aux unités syntaxiques du programme source. (Si l'édition est réalisée par un sous-programme de l'analyse syntaxique, le résultat est alors le descripteur associé à l'unité syntaxique reconnue).

Utilisation de table d'états pour l'analyse lexicographique.

L'analyse lexicographique peut être réalisée au moyen de tables d'états. Par exemple, l'unité syntaxique "identificateur" est définie par les règles suivantes :

$I \rightarrow l \mid I l \mid I d$

l lettre

d chiffre

I Identificateur

La reconnaissance de cette entité peut être faite à l'aide de la table d'états suivante :

	l	d
S1	S2	erreur
S2	S2	S2

S1 est l'état initial,

S2 l'état final.

De la même façon, l'unité syntaxique "nombre" est définie par les règles :

$N \rightarrow V \mid sV$

N nombre

$V \rightarrow D \mid E \mid DE$

V nombre sans signe

$D \rightarrow J \mid F \mid JF$

D nombre décimal

$E \rightarrow eI$

E partie exposant

$F \rightarrow pJ$

J entier sans signe

$I \rightarrow J \mid sJ$

F fraction décimale

$J \rightarrow dJ \mid d$

I entier

s signe (+ ou -)

e 10 (base 10)

p . (point décimal)

d chiffre

et peut être reconnue au moyen de la table d'états suivante :

	s	d	p	e
S1	S2	S3	S4	S6
S2	erreur	S3	S4	S6
S3	erreur	S3	S4	S6
S4	erreur	S5	erreur	erreur
S5	erreur	S5	erreur	S6
S6	S7	S8	erreur	erreur
S7	erreur	S8	erreur	erreur
S8	erreur	S8	erreur	erreur

S1 est l'état initial,

S8 l'état final.

La production de ces tables d'états peut être réalisée automatiquement au moyen de deux transformations successives :

a) transformation des règles "context free" en règles "forme standard", suivie d'une série de transformations itératives des règles "forme standard" en règles "forme standard" équivalente jusqu'à l'obtention d'une grammaire d'états finis dans laquelle toutes les règles sont de la forme suivante : [33]

$A \rightarrow a$
ou $A \rightarrow a B$

b) transformation de la table d'états non déterministe ainsi obtenue en une table d'états déterministe [34].

Il convient toutefois de remarquer que la première transformation peut ne pas se terminer alors qu'il existe un langage d'états finis équivalent. Un exemple bien connu est le suivant :

$A \rightarrow x A x$
 $A \rightarrow x$

pour lequel la première transformation ne se termine pas, bien que le langage défini soit analysable au moyen d'un tel processus.


```

DEBUT COMMENTAIRE PROGRAMME EDITEUR ;
ENTIER M, NUMERO, NBENTIER, NBIDEN, SYMBOLE DE BASE, ENTIER, IDENTIFIER,
CODE CROCHET DROIT, CODE VIRGULE, CODE PAR DROITE, CODE 2 POINTS,
CODE POINT VIRG, CODE 2 PTS EGAL, VEGAL, V2POINTS, VAPOSTROPHE, VPOINT,
VESPACE, VPARGAUCHE, NUMERO DE FIN, NUMERO DE COMMENTAIRE ;
COMMENTAIRE SYMBOLE DE BASE, ENTIER, IDENTIFIER SONT LA PARTIE
TYPE DU DESCRIPTEUR DE L UNITE SYNTAXIQUE CODEE DANS SU. POUR LES
SYMBOLES DE BASE LA PARTIE ARGUMENT DU DESCRIPTEUR EST DONNEE PAR
CODE CROCHET DROIT, ET LES TABLEAUX CODE ET TB CODE. VEGAL, ...
REPRESENTENT LA VALEUR DES CODES DES CARACTERES =, ... ;
ENTIER I, CAR, N ;
ENTIER TABLEAU TABLE IDEN[1:50], TABLE NOMBRE[1:20],
TABLE SYMBOLE DE BASE, TB CODE[1:37], CODE[1:6], SU[1:100] ;
PROCEDURE CARSUIVANT ;
INSYMBOL(1, " 0123456789ABCDEFGHIJKLMNPOQRSTUVWXYZ'+-/(, :*. ) SPA"
CAR) ;
BOOLEEN PROCEDURE ALPHANUM ; ALPHANUM:=CAR < 36 ;
BOOLEEN PROCEDURE NUMERIQUE ; NUMERIQUE:=CAR < 10 ;
BOOLEEN PROCEDURE ALPHABETIQUE ; ... ;
PROCEDURE ERREUR(X) ; ... ;
PROCEDURE IDENTIFICATEUR ;
DEBUT COMMENTAIRE CETTE PROCEDURE RANGE DANS M LES 6
PREMIERS CARACTERES ET IGNORE LES SUIVANTS JUSQU AU
PROCHAIN DELIMITEUR ;
M:=CAR ;
POUR I:=2 PAS 1 JUSQUA 6 FAIRE
DEBUT
CARSUIVANT ;
SI ALPHANUMERIQUE ALORS M:=M+6+CAR
SINON ALLERA SORTIE ;
FIN ;
POUR I:=1 TANTQUE ALPHANUMERIQUE FAIRE
CARSUIVANT ;
SORTIE ;
FIN ;
PROCEDURE CONSULTER(T, N) ;
ENTIER N ; ENTIER TABLEAU T ;
DEBUT
POUR I:=1 PAS 1 JUSQUA N FAIRE
RECHERCHE: SI M=T[I] ALORS ALLERA RESULTAT ;
MISE EN TABLE: N:=N+1 ; T[N]:=M ; I:=N ;
RESULTAT: NUMERO:=I ;
FIN ;
PROCEDURE RANGER(TYPE, ARGUMENT) ;
VALEUR TYPE, ARGUMENT ; ENTIER TYPE, ARGUMENT ;
DEBUT COMMENTAIRE CETTE PROCEDURE FORME UNE UNITE SYNTAXIQUE
ET LA RANGE DANS LE TABLEAU SU ;
N:=N+1 ; SU[N]:=TYPE+ARGUMENT
FIN ;
PROCEDURE TABLE SB ;
DEBUT COMMENTAIRE CETTE PROCEDURE RECHERCHE UNE CORRESPONDANCE
DANS LA TABLE DES SYMBOLES DE BASE ENTRE APOSTROPHES ;

```

```

    POUR I:=1 PAS 1 JUSQUA 37 FAIRE
        SI M=TABLE SYMBOLE DE BASE[I] ALORS ALLERA R ;
    ERREUR(1) ;
R: NUMERO:=1 ;
    FIN ;
AIGUILLAGE EDITER:=NB,NB,NB,NB,NB,NB,NB,NB,NB,NB,
    ID,ID,ID,ID,ID,ID,ID,ID,ID,ID,
    ID,ID,ID,ID,ID,ID,ID,ID,ID,ID,
    ID,ID,ID,ID,ID,ID,
    APOSTROPHE,
    SD,SD,SD,SD,SD,SD,
    DEUX POINTS,ASTERISQUE,POINT,PAR DROITE,ESPACE ;

.....

EDITION: ; COMMENTAIRE LE PROGRAMME EDITEUR PROPREMENT DIT COMMENCE
    ICI ;
    I:=N:=NBENTIER:=NBIDEN:=0 ;
E1: CARSUIVANT ;
E2: ALLERA EDITER[CAR] ; ALLERA ILLEGAL ;

NB: M:=CAR ;
    POUR I:=2 PAS 1 JUSQUA 10 FAIRE
        DEBUT
        CARSUIVANT ;
        SI NUMERIQUE ALORS M:=M*10+CAR
        SINON ALLERA SORTIE ;
        FIN ;
    POUR I:=1 TANTQUE NUMERIQUE FAIRE
        DEBUT
        DEBORDEMENT: ERREUR(2) ; CARSUIVANT ;
        FIN ;
    SORTIE: CONSULTER(TABLE NOMBRE,NBENTIER) ;
    RANGER(ENTIER,NUMERO) ;
    ALLERA E2 ;

ID: IDENTIFICATEUR ;
    CONSULTER(TABLE IDEN,NBIDEN) ;
    RANGER(IDENTIFIER,NUMERO) ;
    ALLERA E2 ;

APOSTROPHE:
    CARSUIVANT ; IDENTIFICATEUR ;
    TABLE SB ;
    SI CAR ≠ VAPOSTROPHE ALORS ERREUR(4) ; CARSUIVANT ;
    SI NUMERO=NUMERO DE COMMENTAIRE ALORS
        ALLERA COMMENTAIRE ;
    RANGER(SYMBOLE DE BASE,TB CODE[NUMERO]) ;
    SI NUMERO=NUMERO DE FIN ALORS
        ALLERA COMMENTAIRE APRES FIN ;
    ALLERA E2 ;

```

SD: DELIMITEUR SIMPLE: ; COMMENTAIRE C EST LE CAS DES DELIMITEURS

+ - / = (, ;
 RANGER(SYMBOLE DE BASE, CODE[CAR-37]) ;
 ALLERA E1 ;

PAR DROITE:

CARSUIVANT ;
 SI CAR=VPOINT ALORS
 DEBUT
 RANGER(SYMBOLE DE BASE, CODE CROCHET DROIT) ;
 ALLERA E1
 FIN ;

SI ALPHABETIQUE ALORS
 COMMENTAIRE DE PARAMETRES:
 DEBUT
 POUR I:=0 TANTQUE ALPHABETIQUE FAIRE CARSUIVANT ;
 SI CAR=V2POINTS ALORS

DEBUT
 CARSUIVANT ;
 SI CAR=VPARGAUCHE ALORS
 DEBUT
 RANGER(SYMBOLE DE BASE, CODE VIRGULE) ;
 ALLERA E1
 FIN
 FIN ;
 ERREUR(3)
 FIN

SINON RANGER(SYMBOLE DE BASE, CODE PAR DROITE) ;
 ALLERA E2 ;

COMMENTAIRE APRES FIN:...

L3: CARSUIVANT ;

COMMENTAIRE:

SI CAR=V2POINTS ALORS
 DEBUT
 CARSUIVANT ;
 SI CAR=V2POINTS ALORS ALLERA L2 ;
 FIN ;
 ALLERA L3 ;

ILLEGAL: ECRIRE(" ILLEGAL ") ;

ESPACE: ALLERA E1 ;

DEUX POINTS:

CARSUIVANT ;
 SI CAR=V2POINTS ALORS

L2: DEBUT
 RANGER(SYMBOLE DE BASE, CODE POINT VIRG) ;
 ALLERA E1
 FIN

SINON SI CAR=VEGAL ALORS
 DEBUT
 RANGER(SYMBOLE DE BASE, CODE 2 PTS EGAL) ;

```
ALLERA E1  
FIN  
SINON  
DEBUT  
RANGER(SYMBOLE DE BASE, CODE 2 POINTS) ;  
ALLERA E2  
FIN ;
```

ASTERISQUE:....;

POINT:....;

FIN:

FIN DU PROGRAMME EDITEUR

Analyse syntaxique.

L'analyse syntaxique est réalisée selon deux approches bien différentes :

- au moyen d'algorithmes dont la structure reflète la syntaxe d'un langage particulier et applicables à ce langage seulement.

- au moyen d'algorithmes généraux permettant l'analyse syntaxique de toute une classe de langages dont la syntaxe peut être exprimée à l'aide d'un certain formalisme (généralement la forme normale de Backus).

2 - ALGORITHMES PARTICULIERS.

a) Méthode des états syntaxiques [36]

Cette méthode est une généralisation de la méthode de Bauer et Samelson [37] utilisant une matrice dite de transition. Nous allons décrire brièvement cette dernière méthode :

Le nombre de lignes de la matrice de transition est égal au nombre d'éléments d'entrée et le nombre de colonnes au nombre d'éléments pouvant apparaître au sommet de la pile d'opérateurs et délimiteurs.

Etant donné un élément d'entrée et un élément sommet de la pile, la matrice de transition indique soit :

- une erreur dans le cas d'une paire d'éléments erronés
- la liste de sous-programmes à exécuter pour chaque paire valide

Pour une instruction d'affectation très simple (sans fonctions, variables indicées et conditions) la matrice de transition a la forme suivante :

Sommet de pile Elément d'entrée	┌	(:=	+ -	× /	↑
Identificateur	k:=k+1 ; P2[k]:=S[i]; i:=i+1					
(Erreur	j:=j+1 ;	P1[j]:=S[i];	i:=i+1		
)	Erreur	j:=j-1 ; i:=i+1	Erreur	P2[k-1]:=OP(P1[j],P2[k-1],P2[k]) k:=k-1 ; j:=j-1		
:=	j:=j+1 ; P1[j]:=S[i]; i:=i+1	Erreur	j:=j+1; P1[j]:=S[i]; i:=i+1	Erreur		
┌	FIN	Erreur	AFF(P2[k-1],P2[k]); P2[k-1]:=P2[k]; k:=k-1 ; j:=j-1	P2[k-1]:=OP(P1[j],P2[k-1],P2[k])		
+ -	Erreur	j:=j+1 ; P1[j]:=S[i];	i:=i+1	k:=k-1 ;		
× /				j:=j-1		
↑						

La chaîne d'éléments d'entrée est désignée par le tableau unidimensionnel S, la pile opérateur par P1 et la pile opérande par P2.

Le principe de l'utilisation de cette matrice est le suivant :

Le contenu de la pile opérateurs définit un état qui ne dépend que de l'élément situé au sommet de la pile ; le prochain élément de la chaîne d'entrée et l'état déterminent la sortie des instructions du programme objet et l'état suivant.

Dans la généralisation, Grau considère le compilateur comme une structure de sous-programmes avec deux sous-programmes de base associés aux entités syntaxiques : "expression" et "instruction". Ces sous-programmes sont récursifs étant donné que les définitions syntaxiques correspondantes sont elles-mêmes récursives. Chacun de ces deux sous-programmes est constitué par un ensemble de sous-programmes mutuellement récursifs : par exemple le sous-programme "expression" peut appeler le sous-programme "terme" qui peut lui-même appeler le sous-programme "expression" à un niveau différent.

La pile opérateurs (ou pile de contrôle) peut aussi être considérée comme la pile des adresses de retour pour tous les sous-programmes du compilateur.

Nous donnons ci-après l'algorithme sous forme d'un programme algol avec les remarques additionnelles suivantes :

La matrice de transition a 32 colonnes (états) et 22 lignes (éléments d'entrée).

Un élément d'entrée est défini comme

```
<élément d'entrée> ::= début | fin | si | alors | sinon | ( | [ | ] |
    allera | pour | faire | pas | jusqua | tant que | ; | , | := |
    <opérateur> | <déclarateur> | <spécificateur> | <identificateur> |
    <nombre> | <valeur logique>
```

Les états sont définis à partir des règles syntaxiques d'algol.

Une méthode de détermination des états syntaxiques est décrit dans la thèse de G. Werner [38]

32 états

}

22 éléments d'entrée

	e ₀	e ₁	e ₂	e ₃		e ₃₁
<u>début</u>	E1	E1	E2	ER		
<u>pour</u>	E1	E1	E7			
<u>allera</u>						
<u>si</u>						
identif						

Dans le programme Algol qui suit, on utilise la matrice de transition au moyen d'une suite d'aiguillages.

SIGMA désigne la pile d'états et GAMMA la chaîne d'éléments d'entrée.


```

DEBUT COMMENTAIRE METHODE DES ETATS SYNTAXIQUES ;
  ENTIER S,G,ST,BEGIN,S0,S1,S2,BA1,BA2,P,0,I1,I2,I3,I4,EE0,EE1,EE2,
  EE3,GG,L1,L2,L3,CG,F0,F1,F2,F3,F4,F5,CE1,CE2,CE3,C1,C2,C3 ;
  ENTIER OMEGA,DECLPART ;
  ENTIER TABLEAU SIGMA[1:50],GAMMA[1:3000] ;
  AIGUILLAGE TABLE:=A1[ST],A2[ST],.....,A22[ST] ;
  AIGUILLAGE A1:=E1,E1,E2,ER,ER,ER,ER,ER,ER,ER,ER,ER,ER,ER,ER,ER,ER,ER,ER,ER,ER,ER,ER,E3,E4,E5,E6,E5,ER,ER,ER,ER,ER,ER,ER ;
  AIGUILLAGE A2:=E1,E1,E7..... ;

```

```

.....
AIGUILLAGE A22:=... ;
PROCEDURE ENT(X) ; ENTIER X ; DEBUT S:=S+1 ;
SIGMA[S]:=X FIN ;
PROCEDURE CH(X) ; ENTIER X ; SIGMA[S]:=X ;
PROCEDURE EXIT ; S:=S-1 ;
PROCEDURE REP ; DEBUT S:=S-1 ; G:=G-1 FIN ;
PROCEDURE PP ; G:=G-1 ;
PROCEDURE GT ; ALLERA NEXT ;
PROCEDURE EOB ; DEBUT S:=S-1 ; SI S=0 ALORS ALLERA END ;
  FIN ;

```

```

PROCEDURE OPR ; DEBUT OMEGA:=SIGMA[S] ; S:=S-1 ;
  SI PREC(SIGMA[S]) > PREC(GAMMA[G]) ALORS REP ;
  S:=S+1 ; ENT(GAMMA[G]) ; ENT(EE2) ; ENT(0)
  FIN ;

```

```

ENTIER PROCEDURE PREC(X) ;
  DEBUT COMMENTAIRE LA VALEUR DE CETTE PROCEDURE EST LA
  PRIORITE DE X ;

```

```

  FIN ;
ENTIER PROCEDURE TYPE(X) ;
  DEBUT COMMENTAIRE LA VALEUR DE CETTE PROCEDURE EST LE
  TYPE DE L UNITE SYNTAXIQUE X ;
  FIN ;

```

```

S0:=1 ; S1:=2,..... ; C3:=32 ;
S:=1 ;
G:=1 ;
SI GAMMA[G] # BEGIN ALORS ALLERA ER ;
SIGMA[S]:=S0 ;
NEXT: G:=G+1 ;
SV:=TYPE(GAMMA[G]) ;
COMMENTAIRE LA VALEUR PRISE PAR SV EST COMPRISE ENTRE
1 ET 22 SELON LE TYPE ;
ST:=SIGMA[S] ;
ALLERA TABLE[SV] ;

```

```

ER: ERROR() ;
E0: CH(S1) ; PP ; GT ;
E1: ENT(S2) ; PP ; GT ;
E2: ENT(S0) ; GT ;
E3: E4: E5: E6: GT ;
E7: ENT(F0) ; GT ;

```

```

.....
E37: DECL: G:=G+1 ;

```

SI GAMMA[G]=DECLPART ALORS ALLERA DECL ; GT ;

E61: OPR ; GT ;
END : FIN

Cet algorithme peut être mis en oeuvre de la façon suivante :

Dans la mémoire du calculateur, il y a une table divisée en sections dont chacune correspond à un élément d'entrée.

Lorsqu'on lit un nouvel élément d'entrée, l'adresse de la section correspondante est déterminée. Dans chaque section on trouve :

- l'ensemble des états qui peuvent intervenir dans cette section
- la liste des sous-programmes élémentaires utilisée dans cette section.
(par sous-programmes élémentaires on entend ceux qui doivent être exécutés lorsqu'un couple d'éléments (entrée, état) est valide).

A chaque sous-programme élémentaire sont associées deux mémoires :

- la première définit la liste des états pour lesquels ce sous-programme doit être exécuté.
- la seconde contient l'adresse d'appel du sous-programme.

Lorsqu'une section et un état sont donnés, on vérifie d'abord si cet état est possible. Si oui on détermine les sous-programmes élémentaires qui doivent être exécutés correspondant à l'état et à l'élément d'entrée, sinon il y a erreur.

La détermination des sous-programmes à exécuter est réalisée au moyen d'une intersection logique de l'état donné avec chacune des mémoires de la section correspondante contenant la liste des états pour lesquels les sous-programmes élémentaires doivent être exécutés.

Méthode des fonctions syntaxiques.

A toute définition syntaxique, on peut associer une fonction syntaxique. Appliquée à une suite de symboles sa valeur est égale à vrai si la suite de symboles représente un élément syntaxique de la catégorie définie, et égale à faux dans les autres cas.

La suite de symboles est balayée à l'aide d'un pointeur qui repère toujours le prochain élément à analyser.

Exemples de fonctions syntaxiques.

Définition d'un programme Algol :

<programme> ::= <bloc> | <instruction composée>

La fonction syntaxique associée est définie de la façon suivante :

Booléen procédure PROGRAMME ;

si INSTRUCTION COMPOSEE alors PROGRAMME := vrai
sinon si BLOC alors PROGRAMME := vrai
sinon PROGRAMME := faux

Définition d'un identificateur Algol :

<identificateur> ::= <lettre> | <identificateur><lettre> | <identificateur><chiffre>

Dans la fonction syntaxique correspondante on utilise un tableau RANGER POINTEUR [N] pour ranger la valeur du pointeur à l'entrée de la procédure et la restaurer à la sortie dans le cas d'une détection d'erreur.

Booléen procédure IDENTIFICATEUR ;

début

N := N+1 ;

RANGER POINTEUR [N] := POINTEUR ;

si LETTRE alors allera FAUX ;

BOUCLE : si LETTRE alors allera BOUCLE ;

si CHIFFRE alors allera BOUCLE ;

VRAI : IDENTIFICATEUR := vrai ; allera FIN ;

FAUX : IDENTIFICATEUR := faux ;

POINTEUR := RANGER POINTEUR [N] ;

FIN : N := N-1

fin IDENTIFICATEUR ;

Définition des expressions préfixées :

<expression> ::= <variable> | <op un><expression> |
 <op bin><expression><expression>

Booléen procédure EXPR ;

début

N := N+1 ;

RANGER POINTEUR [N] := POINTEUR

si VARIABLE alors allera VRAI ;

si OPUN alors début

si EXPR alors allera VRAI sinon allera FAUX

fin ;

si OPBIN alors début

si EXPR alors début

si EXPR alors allera VRAI sinon allera FAUX

fin

fin ;

FAUX : EXPR := faux

POINTEUR := RANGER POINTEUR [N] ;

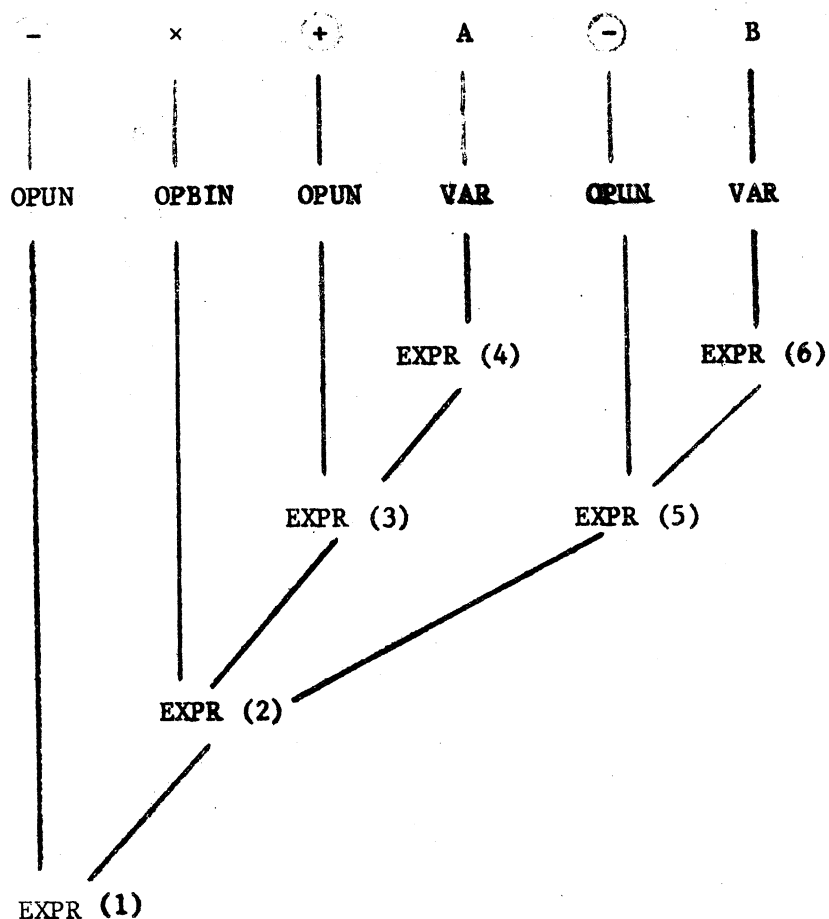
allera FIN ;

VRAI : EXPR := vrai ;

FIN : N := N-1

fin EXPR ;

Exemple :



Cet arbre syntaxique fait apparaître explicitement l'ordre des appels de la procédure récursive EXPR.

Vérificateurs syntaxiques.

Un vérificateur syntaxique est un algorithme permettant de tester la validité syntaxique d'un programme. A la différence d'un analyseur syntaxique qui fournit un arbre syntaxique, le vérificateur syntaxique ne renseigne que sur la validité syntaxique. Nous avons donné au paragraphe précédent des exemples de vérificateurs syntaxiques sous la forme de procédures booléennes (fonctions syntaxiques). Généralement, il est plus simple d'utiliser des procédures ordinaires (procédures syntaxiques).

La procédure Algol suivante décrit un algorithme pour la vérification des programmes écrits en Algol.

Cet algorithme est donné sous la forme d'une procédure PROGRAMME dont l'entrée est un programme ALGOL à analyser formé d'une suite d'unités syntaxiques. Toutes les fois que cette procédure a besoin d'une unité syntaxique elle utilise la procédure LIRE US (dont le corps est en code machine) qui lit l'unité syntaxique suivante et range le descripteur associé dans TEMP. La procédure PROGRAMME utilise d'autres procédures et en particulier les procédures ETIQUETTE, DECLARATION et INSTRUCTION. Lorsque le programme analysé est correct, la sortie de la procédure s'effectue normalement ; les sorties ERRE R 9, ERREUR 10 correspondent à une détection d'erreurs au niveau du premier début et du dernier fin du programme analysé.

La structure de cette procédure PROGRAMME reflète la syntaxe des programmes ALGOL 60, définie par les règles suivantes :

```

<PROGRAMME> ::= <BLOC> | <INSTRUCTION COMPOSEE>
<BLOC> ::= <BLOC NON ETIQUETE> <ETIQUETTE> : <BLOC>
<INSTRUCTION COMPOSEE> ::= <INSTRUCTION COMPOSEE NON ETIQUETEE> |
    <ETIQUETTE> : <INSTRUCTION COMPOSEE>
<BLOC NON ETIQUETE> ::= <TETE DE BLOC> ; <FIN D'INSTRUCTION COMPOSEE>
<INSTRUCTION COMPOSEE
NON ETIQUETEE> ::= début <FIN D'INSTRUCTION COMPOSEE>
<TETE DE BLOC> ::= début <DECLARATION> | <TETE DE BLOC> <DECLARATION>
<FIN D'INSTRUCTION COMPOSEE> ::= <INSTRUCTION> fin | <INSTRUCTION> ; fin
    D'INSTRUCTION COMPOSEE
    
```

qui peuvent se résumer schématiquement comme suit :

Le bloc est défini par :

ETIQUETTE : ... début DECLARATION ; ... DECLARATION ; INSTRUCTION ; ... INSTRUCTION fin

L'INSTRUCTION COMPOSEE est définie par :

ETIQUETTE : ... début INSTRUCTION ; ... INSTRUCTION fin

Un programme commence donc par une liste d'étiquettes (éventuellement vide), suivie du symbole début, suivie d'une liste de déclarations (éventuellement vide : cas de l'instruction composé), suivie d'une liste d'instructions et se termine par fin. Ces remarques permettent de transformer les règles de génération en règles de reconnaissance.

La procédure INSTRUCTION est une procédure récursive. Cette propriété n'est que la transposition de la récursivité de la définition syntaxique correspondante : dans l'analyse syntaxique d'une instruction on peut en effet démarrer l'analyse syntaxique d'une nouvelle instruction avant que la précédente soit terminée.

La structure de cette procédure reflète la syntaxe des instructions définie par les règles suivantes :

```
<INSTRUCTION> ::= <INSTRUCTION INCONDITIONNELLE> |
                  <INSTRUCTION CONDITIONNELLE> |
                  <INSTRUCTION POUR> |
```

Comme précédemment, on transforme ces règles de génération en règles de reconnaissance au moyen des remarques suivantes :

Une instruction pour commence obligatoirement par le symbole pour,
 une " conditionnelle " " " si,
 les instructions inconditionnelles commencent par un symbole différent de pour et si.

La procédure INSTRUCTION INCONDITIONNELLE est aussi une procédure récursive mais indirectement par l'intermédiaire de la procédure INSTRUCTION qui est utilisée dans sa définition.

PROCEDURE PROGRAMME ;
COMMENTAIRE FONCTIONS SYNTAXIQUES ;
ENTIER DEBUT, FIN, POINT VIRGULE, ETI, TEMP ;

BOOLEEN PROCEDURE DECLARATEUR(X) ;
DEBUT COMMENTAIRE CETTE PROCEDURE NE PREND LA VALEUR VRAI
 QUE SI LE DESCRIPTEUR X CORRESPOND A UN DECLARATEUR ;
FIN ;

PROCEDURE LIRE SU ;
DEBUT COMMENTAIRE CETTE PROCEDURE LIT L UNITE SYNTAXIQUE
 SUIVANTE ET RANGE SON DESCRIPTEUR DANS TEMP. DEBUT,
POINT VIRGULE REPRESENTENT LA VALEUR DES DESCRIPTEURS
 ASSOCIES AUX UNITES DE MEME NOM.
 TOUTES LES PROCEDURES UTILISEES APPELLENT LIRESU
 ET APRES L EXECUTION DE L UNE D ELLES, LE DESCRIPTEUR
 DE L UNITE SYNTAXIQUE SUIVANTE EST DANS TEMP ;
FIN ;

PROCEDURE DECLARATION ;
DEBUT COMMENTAIRE CETTE PROCEDURE EFFECTUE L ANALYSE
 SYNTAXIQUE D UNE DECLARATION ;
FIN ;

PROCEDURE ETIQUETTE ;
DEBUT COMMENTAIRE CETTE PROCEDURE EFFECTUE L ANALYSE
 SYNTAXIQUE D UNE SERIE D ETIQUETTES (PEUT ETRE VIDE) ;
FIN ;

PROCEDURE INSTRUCTION ;
DEBUT COMMENTAIRE CETTE PROCEDURE EFFECTUE L ANALYSE SYNTAXIQUE
 D UNE INSTRUCTION ;

PROCEDURE EXPR(TYPE) ;
DEBUT COMMENTAIRE CETTE PROCEDURE ANALYSE UNE EXPRESSION
 DONT LE TYPE (ARITH, BOOLEEN, DESIGN) EST AFFECTE A TYPE ;
FIN ;

ENTIER PROCEDURE TYP(X) ;
DEBUT COMMENTAIRE CETTE PROCEDURE DONNE LE TYPE DE X
 (ARITH, BOOLEEN,) ;
FIN ;

PROCEDURE LISTE DE POUR ;
DEBUT COMMENTAIRE ANALYSE D UNE LISTE DE POUR ;
FIN ;

PROCEDURE LISTE DE PARAMETRES ;
DEBUT ... FIN ;

PROCEDURE LISTE D INDICES ;
DEBUT ... FIN ;

COMMENTAIRE EN SE TERMINANT LES PROCEDURES ANALYSANT DES LISTES
 POINTENT LE CROCHET DE FIN DE LISTE ;

BOOLEEN PROCEDURE IDENTIFICATEUR ;...
ENTIER SI, ALORS, SINON, POUR, PAS, JUSQUA, TANTQUE, FAIRE,
PARGAUCHE, PARDROITE, CRODROIT, CROGAUCHE, T, TYPE ;

PROCEDURE INSTRUCTION INCONDITIONNELLE ;
BOOLEEN PROCEDURE DELIMITEUR D INSTRUCTION ;...
ENTIER ALLERA, DEUX POINTS EGAL ;

DEBUT

SI TEMP=DEBUT ALORS
 BLOC OU INSTRUCTION COMPOSEE:

DEBUT

LIRESU ;

BLOC: SI DECLARATEUR(TEMP) ALORS

DEBUT

DECLARATION ;

LIRESU ; ALLERA BLOC

FIN ;

FIN D INS COMPOSEE:

INSTRUCTION ;

SI TEMP=POINT VIRGULE ALORS

DEBUT

LIRESU ; ALLERA FIN D INS COMPOSEE

FIN ;

SI TEMP=FIN ALORS ALLERA SORTIE1 ;

ERREUR(1)

FIN BLOC OU INSTRUCTION COMPOSEE ;

SI TEMP=ALLERA ALORS

INSTRUCTION ALLERA:

DEBUT

LIRESU ;

EXPR(TYPE) ;

SI TYPE # DESIGN ALORS ERREUR(2) ;

ALLERA SORTIE

FIN ;

SI DELIMITEUR D INSTRUCTION(TEMP) ALORS

INSTRUCTION VIDE: ALLERA SORTIE ;

INSTRUCTIONS PROCEDURE ET D AFFECTATION:

SI IDENTIFICATEUR(TEMP) ALORS

DEBUT

T:=TYP(TEMP) ; LIRESU ;

SI DELIMITEUR D INSTRUCTION(TEMP) ALORS

PROCEDURE SANS PARAMETRE:

ALLERA SORTIE ;

```

SI TEMP=PARGAUCHE ALORS
PROCEDURE AVEC PARAMETRES:
  DEBUT
  LISTE DE PARAMETRES ;
  ALLERA SORTIE1
  FIN ;
AFFECTATION: ;
COMMENTAIRE ON RESTREINT UNE INSTRUCTION D AFFECTATION
A NE CONTENIR QU UNE PARTIE GAUCHE ;
SI TEMP:=DEUX POINTS EGAL ALORS
DEBUT
LIRESU ;
EXPR(TYPE) ;
SI T ≠ TYPE ALORS ERREUR(3) ;
ALLERA SORTIE
FIN AFFECTATION VARIABLE SIMPLE OU IDENTIFICATEUR
DE FONCTION ;
SI TEMP=PARGAUCHE ALORS
DEBUT
LISTE D INDICES ;
LIRESU ;
SI TEMP ≠ DEUX POINTS EGAL ALORS ERREUR(4) ;
EXPR(TYPE) ;
SI T ≠ TYPE ALORS ERREUR(5) ;
ALLERA SORTIE
FIN ; FIN ;
ERREUR(6) ;
SORTIE1: LIRESU ;
SORTIE:
FIN INSTRUCTION INCONDITIONNELLE ;

```

```

DEBUT INSTRUCTION:
ETIQUETTE ;
SI TEMP=POUR ALORS
INSTRUCTION POUR:
  DEBUT
  LIRESU ;
  LISTE DE POUR ;
  SI TEMP ≠ FAIRE ALORS ERREUR(7) ;
  LIRESU ;
  INSTRUCTION ;
  ALLERA FIN
  FIN ;
SI TEMP=SI ALORS
INSTRUCTION CONDITIONNELLE:
  DEBUT
  LIRESU ;
  EXPR(TYPE) ;
  SI TYPE ≠ BOOLEEN OU TEMP ≠ ALORS ALORS
  ERREUR(8) ;
  LIRESU ;
  ETIQUETTE ;

```

```
SI TEMP=POUR ALORS ALLERA INSTRUCTION POUR ;
INSTRUCTION INCONDITIONNELLE ;
SI TEMP=SINON ALORS
  DEBUT
  LIRESU ;
  INSTRUCTION ;
  FIN
  ALLERA FIN
FIN ;
INSINCONDITIONNELLE:
  INSTRUCTION INCONDITIONNELLE ;
FIN:
FIN INSTRUCTION ;

PROCEDURE ERREUR (X) ; ... ;
DEBUT ANALYSE:
  LIRESU ;
  ETIQUETTE ;
  SI TEMP ≠ DEBUT ALORS ERREUR(9) ;
  LIRESU ;
  TETE: SI DECLARATEUR(TEMP) ALORS
    DEBUT
    DECLARATION ;
    LIRESU ; ALLERA TETE
    FIN ;
  FIN D INS COMPOSEE:
    INSTRUCTION ;
    SI TEMP=POINT VIRGULE ALORS
      DEBUT
      LIRESU ;
      ALLERA FIN D INS COMPOSEE
      FIN
    SI TEMP
      FIN ALORS ERREUR(10) ;
FIN PROGRAMME ;
```

Cet algorithme met en évidence une propriété caractéristique de la syntaxe d'ALGOL : les définitions récursives. Dans l'analyse syntaxique d'ALGOL on trouve en définitive 3 procédures de base : celles relatives à l'analyse des instructions, des déclarations et des expressions. Ces procédures ont la particularité d'être mutuellement récursives : par exemple la procédure instruction utilise les procédures déclaration et expression, la procédure déclaration utilise les procédures instruction et expression (en effet une déclaration de procédure contient des instructions) ; il y a une exception pour la procédure expression qui n'utilise ni la procédure déclaration, ni la procédure instruction (cette exception tient au maintien d'une différence qui n'est qu'artificielle entre expression, instruction et déclaration).

3 - ALGORITHMES GENERAUX.

Ces algorithmes d'analyse syntaxique utilisent deux sortes de données :

- les règles syntaxiques du langage source
- le programme source à analyser

et fournissent comme résultat

- l'arbre de décomposition syntaxique du programme analysé.

Analyse unique et analyse multiple.

Si pour un certain mot du programme source, il existe plus d'une décomposition syntaxique, le langage est dit ambigu.

Dans ce cas l'algorithme d'analyse peut fournir

- soit une seule décomposition, cette décomposition étant choisie de préférence aux autres relativement à l'ordre des règles syntaxiques on parle alors d'analyse unique.

- soit toutes les décompositions possibles, on parle alors d'analyse multiple.

Un exemple bien connu tiré d'ALGOL 60 est celui relatif aux expressions booléennes de la forme :

si A alors B sinon C < D

avec les deux décompositions possibles :

(si A alors B sinon C) < D

si A alors B sinon (C < D)

La syntaxe a été modifiée dans le rapport révisé pour éliminer cette ambiguïté.

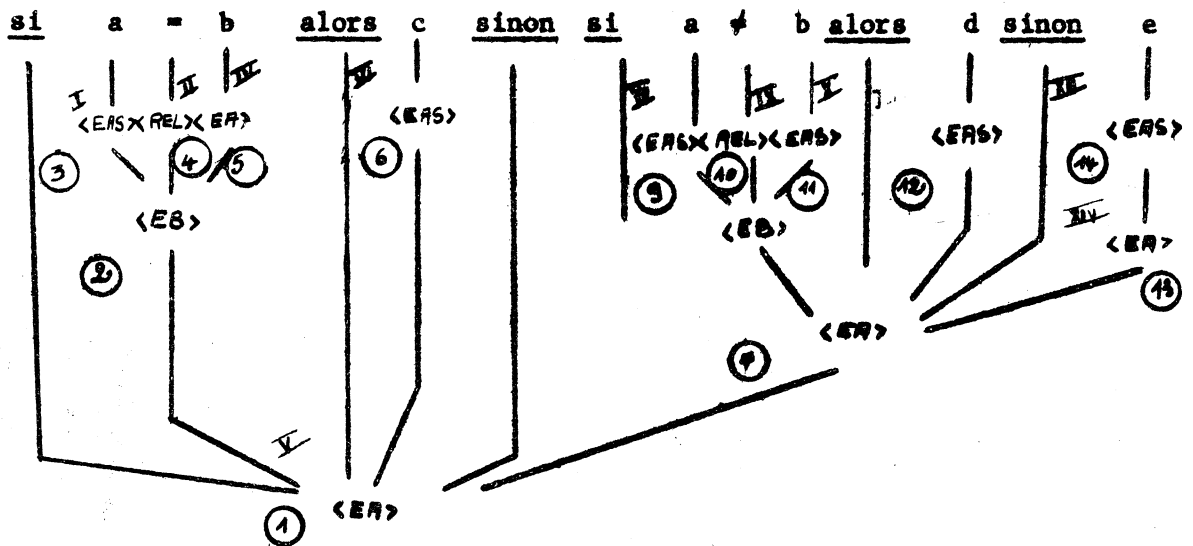
Les algorithmes d'analyse utilisés pour les langages de programmation font rarement de l'analyse multiple, par contre des algorithmes d'analyse multiple semblent couramment utilisés dans la traduction des langages naturels.

Analyse ascendante et descendante.

Considérons les règles suivantes correspondant à une définition simplifiée des expressions arithmétiques (les variables métalinguistiques ont une signification évidente).

1. $\langle EA \rangle ::= \langle EAS \rangle \mid \underline{\text{si}} \langle EB \rangle \underline{\text{alors}} \langle EAS \rangle \underline{\text{sinon}} \langle EA \rangle$
2. $\langle EAS \rangle ::= a \mid b \mid c \mid d \mid e$
3. $\langle REL \rangle ::= \neq \mid =$
4. $\langle EB \rangle ::= \langle EAS \rangle \langle REL \rangle \langle EAS \rangle$

Soit l'expression arithmétique :



et son arbre de décomposition syntaxique.

L'algorithme d'analyse doit trouver une décomposition qui ayant <EA> comme base construit l'arbre syntaxique correspondant à l'expression donnée.

Il se fixe donc <EA> comme but final à atteindre, mais ce but ne peut pas être atteint directement, l'algorithme va donc ranger le but final au sommet d'une pile et se définir une suite de buts intermédiaires correspondant aux différents noeuds de l'arbre syntaxique, qui doivent être atteints pour que le but final soit lui-même atteint. Les buts intermédiaires sont définis à partir des règles syntaxiques fournies à l'algorithme.

L'ordre dans lequel ces buts intermédiaires sont définis caractérise le mode ascendant ou descendant des algorithmes d'analyse.

Analyse descendante

On réécrit les définitions de la façon suivante :

```

<EA> → <EAS>
      ↘ si <EB> alors <EAS> sinon <EA>
<EAS> → a
      ↘ b
      ↘ c
      ↘ d
      ↘ e
<REL> → ≠
      ↘ =
<EB> → <EAS><REL><EAS>

```

Ces définitions sont rangées sous forme d'une table en liste, chaque élément correspondant à un noeud donnant l'adresse de l'élément précédent dans une alternative et l'adresse de l'alternative suivante.

Par exemple la première est rangée de la manière suivante :

J PREDECESSEUR [J] ARBRE [J] ALTERNANT [J]

adresse de l'élément	adresse élément précédent	élément	adresse de l'alternative suiv.
1	**	<EA>	*
2	1	<EAS>	4
3	*	*	*
4	1	<u>si</u>	*
5	4	<EB>	*
6	5	<u>alors</u>	*
7	6	<EAS>	*
8	7	<u>sinon</u>	*
9	8	<EA>	*
10	*	*	*
11	**	<EAS>	*
12	11	a	14
13	*	*	*
14			

Dans le tableau ARBRE, le symbole '*' indique la fin d'une règle et dans le tableau PREDECESSEUR, le symbole '**' indique que l'élément correspondant du tableau ARBRE est une tête de règle.

L'algorithme utilise deux piles : une pour les buts essayés BUT [K] et une pour les buts effectivement atteints TROUVE [L].

Au départ, les deux piles sont vides, on place dans BUT le but <EA> et on examine dans la définition syntaxique de <EA> la première alternative : <EAS> qui est empilée sur <EA>. On examine alors la première définition de <EAS>, comme il s'agit d'un symbole terminal (a) on le compare au premier symbole, de la chaîne à analyser qui est si ; comme il n'y a pas coïncidence, on passe à la seconde alternative de <EAS> et ainsi de suite. Finalement ayant examiné toutes les alternatives de <EAS> sans trouver de coïncidence, on en conclut que le but <EAS> ne peut être atteint, on l'efface alors de la pile BUT et on va examiner la deuxième alternative de <EA> qui est : si <EB> alors <EAS> sinon EA.

Le premier élément de cette définition étant un symbole terminal on examine la coïncidence avec l'élément courant de la chaîne analysée ; il y a coïncidence, on peut donc essayer cette règle, l'élément suivant est <EB> qui devient un but intermédiaire à atteindre, on va alors examiner la définition de <EB> et ainsi de suite

Remarque importante :

Cette description simplifiée escamote un certain nombre de difficultés lors de l'utilisation effective de l'algorithme, en particulier le recul dans le cas d'un but intermédiaire non atteint. La distinction entre élément terminal et élément non terminal dans la lecture des règles syntaxiques peut se réaliser au moyen d'une table supplémentaire donnant pour chaque élément sa classe (terminal ou non terminal) et si c'est le premier symbole d'une définition le début de sa définition dans la table des définitions syntaxiques. Pour le cas ci-dessus on aurait pour les deux premières définitions :

	TERMINAL	RACINE
Elément	Elément terminal	Début de la définition
<EA>	<u>faux</u>	1
<EAS>	<u>faux</u>	i
<EB>	<u>faux</u>	j
<u>si</u>	<u>vrai</u>	-
<u>alors</u>	<u>vrai</u>	-
<u>sinon</u>	<u>vrai</u>	-
etc		

i et j indiquent l'endroit où commencent dans la table les définitions de <EAS> et <EB>.

DEBUT COMMENTAIRE CE PROGRAMME REALISE L'ANALYSE DESCENDANTE
D UNE CHAINE DE SYMBOLES, LES PILES S1 ET S2 SONT APPELES BUT
ET TROUVE DANS CE PROGRAMME ;

ENTIER I,J,K,L,AXIOME,ETOILE,DEUX ETOILES ;
ENTIER TABLEAU CHAINE[:],ARBRE,PRECEDANT,ALTERNANT[:],
RACINE[:],BUT,TROUVE[:];
BOOLEEN TABLEAU TERMINAL[:];

PROCEDURE EMPILERBUT(T1,T2,T3) ; ENTIER T1,T2,T3 ;
DEBUT BUT[K]:=T1 ; BUT[K+1]:=T2 ; BUT[K+2]:=T3 ;
K:=K+3

FIN ;

PROCEDURE EMPILERTROUVE(T1,T2,T3,T4,T5) ;
ENTIER T1,T2,T3,T4,T5 ;
DEBUT TROUVE[L]:=T1 ; TROUVE[L+1]:=T2 ;
TROUVE[L+2]:=T3 ; TROUVE[L+3]:=T4 ;
TROUVE[L+4]:=T5 ; L:=L+5
FIN DE L EMPILAGE DES PARAMETRES ;

PROCEDURE TROUVALTERNANT(N) ; ENTIER N ;
DEBUT SI ALTERNANT[N] ≠ ETOILE ALORS
DEBUT J:=-ALTERNANT[N]-1 ;
ALLERA VERIFINDEREGL
FIN ;
J:=-PRECEDANT[J] ; I:=I-1
FIN ;

AXIOME:=ARBRE[1] ;
I:=0 ; J:=K:=L:=1 ;
EMPILERBUT(AXIOME,I+1,J) ;

VERIFSITERM:

SI TERMINAL[ARBRE[J+1]] ALORS
DEBUT SI CHAINE[I+1]=ARBRE[J+1] ALORS
DEBUT
I:=I+1 ; J:=J+1 ;
VERIFINDEREGL:
SI ARBRE[J+1]=ETOILE ALORS
DEBUT
EMPILERTROUVE(BUT[K-3],BUT[K-2],BUT[K-1],I,J) ;
SI BUT[K-3]=AXIOME ALORS ALLERA FIN ;
J:=BUT[K-1] ; K:=K-3 ;
ALLERA VERIFINDEREGL
FIN ;
ALLERA VERIFSITERM
FIN ;
J:=J+1 ; ALLERA RECULERI
FIN ;

NON TERMINAL:
EMPILERBUT(ARBRE[J+1],I+1,J+1) ;

J:=RACINE[ARBRE[J+1]] ;
ALLERA VERIFSITERM ;

RECULER:

SI NON TERMINAL[ARBRE[J]] ALORS
DEBUT SI PRECEDANT[J]=DEUX ETOILES ALORS

DEBUT I:=-BUT[K-2]-1 ;
J:=-BUT[K-1] ; K:=-K-3 ;
SI K=1 ALORS ALLERA ERREUR ;
TROUVALTERNANT(J)

FIN

SINON

DEBUT

EMPILERBUT(TROUVE[L-5],TROUVE[L-4],TROUVE[L-3]) ;
I:=-TROUVE[L-2] ; J:=-TROUVE[L-1]+1 ;
L:=L-5

FIN ;

ALLERA RECULER

FIN ;

RECULER1:

TROUVALTERNANT(J) ; ALLERA RECULER

FIN:

IN

Inconvénient de l'analyse descendante.

L'analyse descendante a un inconvénient notable : elle ne fonctionne plus pour des définitions syntaxiques récursives à gauche, c'est-à-dire de la forme suivante :

$$\langle A \rangle ::= \langle A \rangle b$$

en effet dans ce cas on empilera indéfiniment le but intermédiaire $\langle A \rangle$ sur la pile P1 sans jamais l'atteindre.

On peut remarquer d'une part que les définitions récursives d'ALGOL 60 sont presque toutes récursives à gauche et d'autre part qu'il est toujours possible de transformer une définition récursive à gauche en une autre récursive à droite.

Par exemple on peut remplacer la règle

$$\langle A \rangle ::= a \mid \langle A \rangle b$$

par les règles suivantes

$$\langle A \rangle ::= a \langle B \rangle$$

$$\langle B \rangle ::= b \mid b \langle B \rangle$$

Il est parfois possible d'éviter de telles transformations en changeant l'ordre des alternatives

Analyse ascendante.

L'analyse ascendante peut être décrite succinctement comme suit :

Partant du premier symbole à analyser on va **essayer** de former la plus grande construction syntaxique qui contient ce symbole comme premier élément et continuant ainsi de proche en proche jusqu'à ce qu'on trouve finalement :

$\langle \text{programme} \rangle$

Considérons les définitions suivantes :

$\langle P \rangle ::= \langle B \rangle$
 $\langle B \rangle ::= e : \langle B \rangle \mid \langle N \rangle \mid i$
 $\langle N \rangle ::= \langle T \rangle ; \langle F \rangle$
 $\langle T \rangle ::= \underline{\text{début}} \ d \mid \langle T \rangle ; d$
 $\langle F \rangle ::= \langle B \rangle ; \langle F \rangle \mid \langle B \rangle \underline{\text{fin}}$

La signification des métavariabes est :

$\langle P \rangle$ programme, $\langle B \rangle$ bloc, $\langle N \rangle$ bloc non étiqueté, $\langle T \rangle$ tête de bloc
 $\langle F \rangle$ fin d'instruction composée

et celle des terminaux désignés par des lettres minuscules

e étiquette d déclaration i instruction.

Les règles syntaxiques sont d'abord écrites sous la forme suivante, dite forme de reconnaissance.

1. $\langle B \rangle \rightarrow \langle P \rangle$
2. $e : \langle B \rangle \rightarrow \langle B \rangle$
3. $\langle N \rangle \rightarrow \langle B \rangle$
4. $\langle T \rangle ; d \rightarrow \langle T \rangle$
 $\quad \quad \quad \searrow$
 $\quad \quad \quad \langle F \rangle \rightarrow \langle N \rangle$
5. $\underline{\text{debut}} \ d \rightarrow \langle T \rangle$
6. $\langle B \rangle ; \langle F \rangle \rightarrow \langle F \rangle$
 $\quad \quad \quad \searrow$
 $\quad \quad \quad \underline{\text{fin}} \rightarrow \langle F \rangle$
7. $i \rightarrow \langle B \rangle$

où l'on a inversé les parties gauche et droite et où l'on a regroupé les définitions commençant par des symboles identiques.

Ces définitions sont alors rangées de la même manière que dans la méthode précédente ce qui donne ici :

Adresse de l'élément	Adresse élément précédent	élément	adresse de l'alternant suivant.
1	**		*
2	1	<P>	*
3	*	*	*
4	**	e	*
5	4	:	*
6	5		*
7	6		*
8	*	*	*
9	**	<N>	*
10	9		*
11	*	*	*
12	**	<T>	*
13	12	;	17
14	13	d	*
15	14	<T>	*
16	*	*	*
17	13	<P>	*
18	17	<N>	*
19	*	*	*
etc			

de la même façon que pour l'algorithme précédent la distinction entre éléments terminaux et non terminaux est faite au moyen d'une table qui précise également pour un élément où commence la règle correspondante dans la table des définitions.

Elément	Terminal	Début de la définition
<P>	<u>faux</u>	-
	<u>vrai</u>	1
e	<u>vrai</u>	4
:	<u>vrai</u>	-
<N>	<u>faux</u>	9
etc.		

Sélectivité.

Il y a une différence importante entre les deux méthodes. La méthode d'analyse ascendante que nous allons décrire est sélective. Cette sélectivité est réalisée par l'utilisation d'une table spéciale dite matrice des successeurs et précisant pour chaque symbole, terminal ou non terminal, si l'application d'une règle commençant par ce symbole est susceptible de conduire au but (final ou intermédiaire) considéré.

Pour l'exemple précédent, cette matrice a la valeur suivante :

	P	B	N	T	F
P					
B	①				①
N	1	①			1
T	1	1	①	①	1
F					
e	1	①			1
:					
;					
d					
i	1	①			1
<u>debut</u>	1	1	1	①	1
<u>fin</u>					

Cette matrice peut être construite à partir des définitions syntaxiques (par exemple pour le symbole e : la règle 2 indique que e conduit à , or conduit à <P> par la règle 1, donc e conduit aussi à <P> ; la règle 6 indique que conduit à <F> or e conduit à (règle 2) donc e conduit aussi à <F> , etc).

Considérons alors l'exemple :

e : debut d ; d ; debut d ; i fin fin

On commence par empiler dans BUT le but final <P> , puis on examine le premier symbole e et après avoir vérifié qu'il peut conduire à <P> d'après la matrice successeur, on cherche à reconnaître une construction syntaxique commençant par e (c'est-à-dire en 4 dans la table des définitions).

La définition indique que e doit être suivi du symbole :
ce qui est effectivement le cas, puis du symbole non terminal qui devient le premier but intermédiaire à atteindre il est donc empilé sur BUT à la suite de <P>.

On cherche alors si avec les symboles suivants (debut, d ; ...) on peut former un ; la matrice successeur indique que le premier symbole debut peut effectivement conduire à , on va appliquer alors la règle 5 qui indique que debut suivi de d conduit à <T> (ce qui est le cas ici).

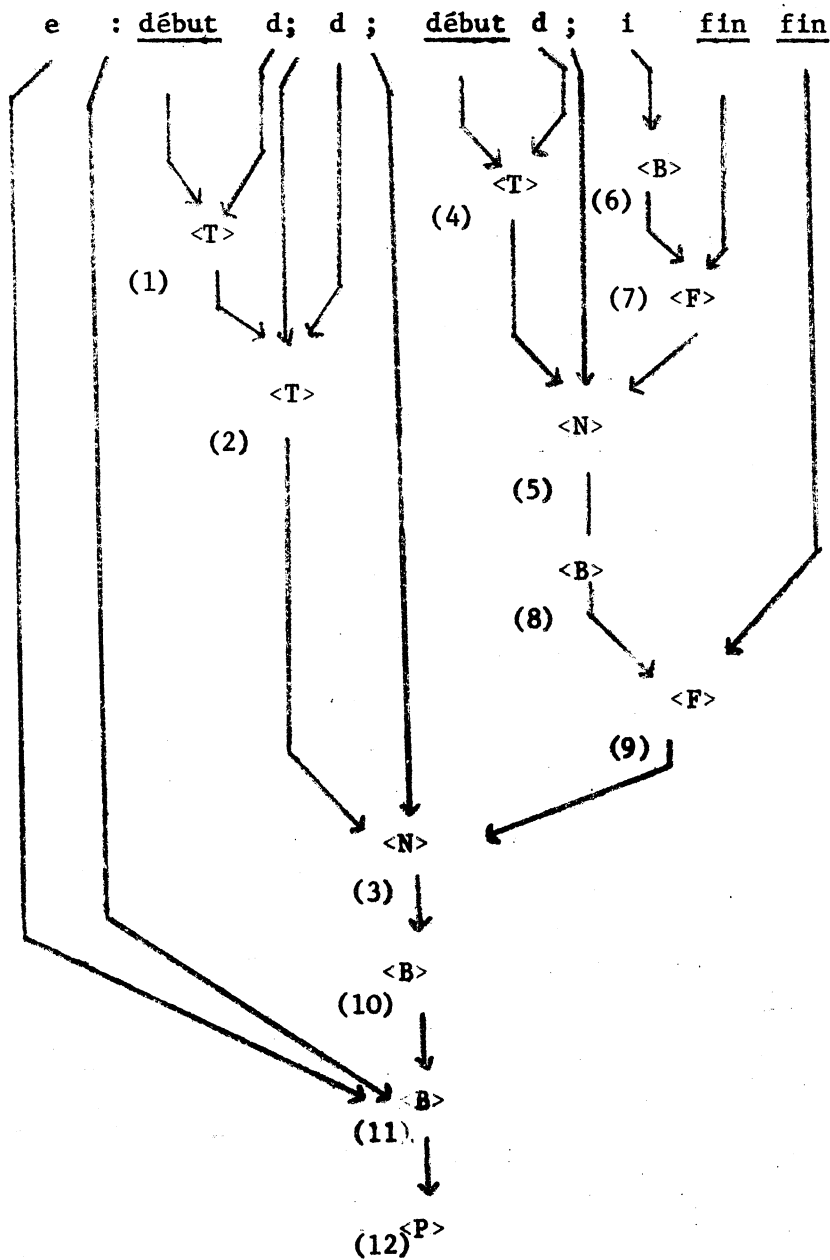
Or notre but est toujours d'atteindre un donc avant d'empiler le but trouvé <T> sur TROUVE, on vérifie d'après la matrice successeur que <T> peut conduire effectivement à .

Puis on cherche l'application d'une règle commençant par <T> et qui éventuellement conduira à . L'application de la règle 4 et la comparaison vérifiée avec les deux éléments suivants ; et d nous conduit de nouveau à <T>.

La caractéristique essentielle des méthodes ascendantes étant de toujours considérer la plus grande chaîne de symboles qui peut représenter une méta-variable donnée, on va donc essayer de continuer à chercher un <T> qui engloberait le précédent (la matrice successeur indique effectivement que <T> peut conduire à <T>. Une nouvelle application de la règle 5 incorpore au nouveau <T> les symboles ; et d qui suivent.

En ce point de l'analyse, nous avons donc deux buts trouvés en TROUVE (<T> et <T>) et deux buts intermédiaires en BUT (<P> et).

A la fin de l'analyse, la pile BUT sera vide et la pile TROUVE contiendra la suite des buts trouvés comme dans la méthode descendante décrivant la structure syntaxique selon le schéma suivant :



DEBUT COMMENTAIRE CE PROGRAMME REALISE L'ANALYSE ASCENDANTE D'UNE
 CHAINE DE SYMBOLES ;
ENTIER I, J, K, L, AXIOME, MARQUE, ETOILE, DEUX ETOILES ;
ENTIER TABLEAU CHAINE[:], ARBRE, PRECEDANT, ALTERNANT[:],
 RACINE[:], BUT, TROUVE[:] ;
BOOLEEN TABLEAU SUCESSEUR[:], :], TERMINAL[:] ;

PROCEDURE EMPILERTROUVE(T1, T2, T3, T4, T5) ;
ENTIER T1, T2, T3, T4, T5 ;..... ;

PROCEDURE EMPILERBUT(T1, T2, T3) ; ENTIER T1, T2, T3 ;

COMMENTAIRE CES DEUX PROCEDURES EMPIENT LES PARAMETRES ;

INITIALISATION:

K:=L:=I:=J:=1 ;
 EMPILERBUT(AXIOME, I, MARQUE) ;
SI NON SUCESSEUR[CHAINE[I], AXIOME] ALORS ALLERA
 ERREUR ;

ENTREE DE L ARBRE:

J:=RACINE[CHAINE[I]] ;

VERIFSITERM:

J:=J+1 ; I:=I+1 ;
SI TERMINAL[ARBRE[J]] ALORS
ALLERA SI CHAINE[I]-ARBRE[J] ALORS VERIFSITERM
SINON RECULER1 ;

VERIFINDEREGLA:

SI ARBRE[J+1]-ETOILE ALORS
DEBUT

I:=I-1 ;

SI ARBRE[J]-BUT[K-3] ALORS

DEBUT

EMPILERTROUVE(BUT[K-3], BUT[K-2], BUT[K-1], I, J) ;

SI BUT[K-3]-AXIOME ALORS ALLERA FIN ;

SI SUCESSEUR[BUT[K-3], BUT[K-3]] ALORS

DEBUT

J:=RACINE[ARBRE[K-3]] ;

ALLERA VERIFSITERM

FIN ;

ENLEVERBUT:

J:=BUT[K-1] ; K:=K-3 ;

ALLERA VERIFSITERM

FIN ;

SI NON SUCESSEUR[ARBRE[J], BUT[K-3]] ALORS

ALLERA RECULER2 ;

EMPILERTROUVE(ARBRE[J], BUT[K-2], 0, I, J) ;

J:=RACINE[ARBRE[J]] ;

ALLERA VERIFSITERM

FIN ;

SI NON SUCCESSEUR[CHAINE[I],ARBRE[J]] ALORS
 ALLERA RECULER1 ;
 EMPILERBUT(ARBRE[J],I,J) ;
 ALLERA ENTREE DE L ARBRE ;

RECULER1:

I:=I-1 ;

RECULER2:

SI ALTERNANT[J] ≠ ETOILE ALORS

DEBUT

J:=-ALTERNANT[J]-1 ;

ALLERA VERIFSITERM

FIN ;

J:=-PRECEDANT[J] ;

SI NON TERMINAL[ARBRE[J]]

ET PRECEDANT[J] ≠ DEUX ETOILES ALORS

DEBUT

EMPILERBUT(TROUVE[L-5],TROUVE[L-4],TROUVE[L-3]) ;

ENLEVERTROUVE:

J:=-TROUVE[L-1] ;

L:=-L-5 ;

ALLERA RECULER2

FIN ;

SI NON TERMINAL[ARBRE[J]] ET PRECEDANT[J]=DEUX ETOILES
 ALORS

ALLERA SI BUT[K-3]=TROUVE[L-5] ALORS ENLEVERBUT
 SINON ENLEVER TROUVE ;

SI PRECEDANT[J] ≠ DEUX ETOILES ALORS

ALLERA RECULER1 ;

J:=-BUT[K-1] ;

K:=-K-3 ;

ALLERA SI J-MARQUE ALORS ERREUR SINON RECULER1 ;

FIN:

FIN

C H A P I T R E IV

EVALUATION SEMANTIQUEI - DEFINITION ET EVALUATION DES PROGRAMMES OBJET.1. Définition des programmes objet.

Les programmes source peuvent être traduits directement en programmes en langage machine mais cette approche présente plusieurs inconvénients :

- le compilateur doit générer des ensembles d'instructions complexes même en utilisant les possibilités d'un assembleur.
- le compilateur réalise simultanément l'évaluation syntaxique et sémantique et le mélange inextricable des deux aspects rend les mises au point et les modifications difficiles.
- le compilateur dépend essentiellement de la machine utilisée
- certaines caractéristiques dynamiques des langages ne peuvent pas être utilisées si les programmes source sont traduits complètement en langage machine.

L'utilisation d'un langage intermédiaire s'impose pour **trois raisons** principales :

a) le langage intermédiaire est interprété en ce sens que chaque élément du langage intermédiaire est un appel de sous-programme dont l'exécution correspond à la sémantique de cet élément. Par exemple, la notation post-fixée correspond à un langage intermédiaire bien adaptée à l'évaluation récursive des expressions.

b) le langage intermédiaire étant différent du langage machine, la compilation n'est plus liée à une machine déterminée mais seulement à un langage intermédiaire appropriée au langage source utilisé.

Les éléments d'un langage objet intermédiaire peuvent être considérés comme des macroinstructions. Sur une machine disposant d'un macroassembleur on peut ainsi traduire en grande partie un programme intermédiaire en un programme machine afin de diminuer le temps d'interprétation pendant l'exécution. L'optimisation est aussi plus facilement réalisée au niveau des macroinstructions.

c) la technique interprétative est la seule qui permette la mise en oeuvre des caractéristiques dynamiques d'un langage source.

- récursivité
- tableaux à bornes variables
- vérification des expressions d'indices
- correspondance paramètres effectifs - paramètres formels.

Dijkstra [29] a montré que l'exécution interprétative utilisant une seule pile de travail représente la solution adéquate de ces problèmes.

La pile de travail est utilisée :

- pour l'allocation dynamique des mémoires : à l'entrée d'un bloc on réserve la zone de mémoire nécessaire pour les quantités locales de ce bloc, à la sortie cette zone est libérée.

- pour l'évaluation des expressions : allocation dynamique des registres contenant les quantités intermédiaires.

De cette façon, la capacité de mémoire nécessaire se trouve optimisée automatiquement. Lorsqu'un bloc est appelé récursivement, une nouvelle zone de mémoire est allouée à chaque appel et les zones de mémoire attachées à des niveaux de bloc qui sont dynamiquement différents sont toujours distinctes.

La définition du langage intermédiaire représente le travail le plus important dans la mise en oeuvre sur machine d'un nouveau langage source et nous allons maintenant décrire les principaux aspects de ce problème.

1.1 Expressions. Affectations. Langages intermédiaires issus de la notation postfixée.

La notation postfixée est la forme la plus simple de langage intermédiaire. Elle a l'inconvénient d'être incomplète dans sa forme habituelle. Par exemple dans la transformation de l'instruction

$R := (A + B \times 7) / D$ sous la forme suivante :
 $R A B 7 \times + D / :=$

n'apparaît pas la distinction entre :

- adresses et valeurs (R est une adresse, A, B, 7 et D sont des valeurs).

- entre valeurs désignées et vraies valeurs (A, B et D sont des valeurs désignées, 7 est une vraie valeur)

- le type des diverses quantités manipulées (entier, réel, etc...)

Diverses formes de langages intermédiaires.

L'expression précédente peut être traduite dans la forme suivante qui traduit l'évaluation récursive de l'expression.

Programme P1.

ADR	R	Empiler l'adresse de R
VAL	A	Empiler la valeur de A
VAL	B	Empiler la valeur de C
CST	7	Empiler la vraie valeur 7
×		Exécuter une multiplication
+		Exécuter une addition
VAL	D	Empiler la valeur de D
/		Exécuter une division
:=		Exécuter une affectation

Les ordres d'opérations sont sans adresses, les opérandes et le résultat étant implicitement définis par l'indice courant de la pile.

Les programmes P2 et P3 ci-après correspondent respectivement à des opérations à une adresse explicite et des opérations à deux adresses, l'astérisque désignant l'opérande sommet de pile. Dans le programme P4, on n'a pas utilisé d'astérisques mais l'ordre des opérandes doit être défini pour les opérations non commutatives dans le code opération.

Programme P2.

```

VAL  A
VAL  B
×    7
+
/    D
:=   R

```

Programme P3.

```

×  B, 7
+  A, *
/  *, D
:= R, *

```

Programme P4.

```

×  B, 7
+  A
/  d  D
:=  g  R

```

Ces programmes sont assez semblables. Dans la suite nous n'utiliserons que les types 1 et 3. On peut remarquer que l'effet de bord n'est pas clairement défini par les types 3 et 4.

Les opérateurs logiques et les opérateurs de relation se comportent de façon semblable. La compilation des expressions conditionnelles nécessite deux ordres de rupture de séquence :

- saut inconditionnel UJP
- saut si sommet de pile est faux JPF

Ces deux ordres comportent une adresse désignant le point du programme ou s'effectue inconditionnellement ou conditionnellement le transfert de contrôle.

Affectation.

L'opérateur d'affectation se comporte comme un opérateur arithmétique et est traité de façon analogue. La variable correspondant à la partie gauche est traduite par un ordre de mise en pile de l'adresse : ADR.

Lorsqu'une opération d'affectation est exécutée, elle trouve ses deux opérands au sommet de la pile : adresse de la variable partie gauche et valeur de l'expression partie droite. Après exécution, les deux opérands sont effacés de la pile.

Pour les instructions d'affectation multiple, on utilise un opérateur spécial $:=_m$; dans ce cas après exécution de l'affectation, le deuxième opérande (valeur de l'expression partie droite) est transféré au sommet de la pile après l'effacement des deux opérandes initiaux.

Exemple :

L'instruction d'affectation multiple

A := B := si C > D alors E sinon F ;

sera transformée en :

```

ADR    A
ADR    B
VAL    C
VAL    D
>
SCF    L1
VAL    E
SIC    L2
L1     VAL    F
L2     :=m
      :=

```

Profil qualitatif.

On désigne par accumulateur une position de la pile pendant l'évaluation d'une expression ou l'exécution d'une instruction d'affectation. Un accumulateur contient une adresse ou une valeur et un profil qualitatif (ensemble d'indications, empilées en même temps qu'une adresse ou une valeur). Le profil qualitatif donne des renseignements sur les quantités définies dans l'accumulateur associé : adresse ou valeur, type (reel, entier, booleen).

Pendant l'exécution, il joue plusieurs rôles : vérification des types associés aux expressions, transferts de type arithmétique (entier \leftrightarrow réel), détermination de la portée d'une variable indiquée, correspondance entre paramètres effectifs et paramètres formels.

Variables indicées.

Une variable spéciale τ est associée avec chaque tableau T. Pendant l'exécution, la valeur de τ fournit l'adresse du tableau et la fonction topographique permettant de déterminer l'adresse d'un élément.

La notation postfixée associée aux variables indicées découle de ces remarques :

Par exemple :

$T1 [T2 [A+B]] := T3 [C, D]$ est transformée en :

ADR	$\tau 1$	
ADR	$\tau 2$	
VAL	A	
VAL	B	
+		
INDA		indexer sur la valeur
INDV		indexer sur l'adresse
ADR	$\tau 3$	
VAL	C	
VAL	D	
INDV		
:=		

qui est la transposition de

$\tau 1 \tau 2 A B + + + \tau 3 C D, + :=$

Le symbole \dagger représente l'opérateur d'indexage.

De la même façon que les variables sont traduites par des ordres ADR ou VAL lorsqu'elles sont utilisées dans une partie gauche d'instruction d'affectation ou dans une expression, les opérateurs \dagger sont traduits par INDA ou INDV, INDA prenant l'adresse et INDV la valeur de la variable indiquée.

Il n'est pas nécessaire de donner la portée de INDA et INDV étant donné qu'elle peut être trouvée au moment de l'exécution en descendant dans la pile jusqu'au profil qualitatif associé à la variable τ correspondante.

1.2 Adressage dynamique.

A tout bloc d'un programme Algol, on peut associer un numéro de bloc lexicographique défini de la façon suivante ;

- le programme a un numéro de bloc égal à 0
- le numéro de bloc augmente de 1 à chaque entrée de bloc et diminue de 1 à chaque sortie.

Une déclaration de procédure est considérée comme un bloc et traitée de la même façon.

Une variable est représentée dans le programme objet par une adresse lexicographique n, p .

n est le numéro de bloc dans lequel la variable est déclarée.

p est l'adresse relative affectée à cette variable dans la zone locale de ce bloc.

Les adresses relatives sont affectées dans l'ordre lexicographique des déclarations. Le compilateur peut aussi évaluer le nombre de mémoires nécessaires pour les variables simples dont les déclarations n'apparaissent plus dans le programme objet, (l'affectation des tableaux est faite dynamiquement pendant l'exécution).

Dans les programmes objet la structure de blocs est traduite par les 2 instructions : entrée bloc EBL et fin de bloc FBL. EBL a un opérande qui définit le volume de mémoire à affecter. A l'exécution, cet ordre EBL réalise la réservation de mémoire dans la pile après avoir empilé les données de liaison. Les données de liaison sont des variables locales implicites pour chaque niveau de bloc. Le compilateur tient compte de ces variables implicites lorsqu'il affecte les adresses relatives.

Les données de liaison permettent le chaînage des activations de bloc et contiennent les éléments nécessaires à l'utilisation des adresses dynamiques. Pendant l'exécution, toute référence à une variable produit une adresse machine à partir d'une adresse relative, il est donc nécessaire de connaître l'adresse origine de la zone affectée au bloc de numéro n. Cette adresse origine est définie par la valeur du pointeur IV du bloc courant ; les données de liaison d'un bloc de numéro n contiennent la valeur du pointeur IV du bloc de numéro n-1. Cet ensemble de pointeurs constitue la chaîne lexicographique et peut être dupliquée dans un vecteur index afin de rendre plus aisée la détermination de l'adresse machine (adresse dynamique) à partir de l'adresse relative par indexage.

FBL qui délimite la fin d'un bloc dans le programme objet libère la zone de mémoire attachée à ce bloc.

La déclaration de procédure se comporte de la même façon. Il y a un ordre d'entrée procédure EPR et un ordre de fin procédure FPR - : ces ordres jouent le même rôle que EBL et FBL mais dans ce cas l'enchaînement d'activation des blocs est différent de l'enchaînement lexicographique.

Pour cette raison, EPR doit comporter un opérande dont la valeur définit le numéro de bloc de la procédure.

Paramètres.

Les paramètres formels se comportent de façon analogue aux quantités déclarées mais ils sont toujours représentés par une adresse dynamique et il y a des ordres spéciaux pour leur interprétation.

Les adresses dynamiques des paramètres formels sont déterminées par le compilateur à partir de la liste de paramètres formels. Une telle adresse se réfère à un accumulateur formel que l'on peut interpréter comme une variable locale de la procédure.

Pendant l'exécution, lorsqu'une procédure est activée, l'information donnant accès au paramètre effectif est placée dans l'accumulateur formel associé.

Un ordre formel exécuté à l'intérieur du corps de procédure se réfère à un accumulateur formel et peut donc attendre le paramètre effectif correspondant.

Un paramètre appelé par valeur est généralement rangé dans l'accumulateur formel associé (à l'exclusion du cas où le paramètre appelé par valeur est un tableau).

Tête de procédure.

Entre l'ordre EPR et le programme objet correspondant au corps de procédure, il y a traduction des parties valeur et spécification.

Dans l'interpréteur décrit ci-après, on suppose que tous les paramètres sont spécifiés.

Un élément de programme objet est associé à chaque paramètre formel et indique sa spécification et son mode d'appel (appel par valeur ou par nom). Ces éléments se trouvent dans le même ordre que la liste de paramètres formels associée. Pendant l'exécution, ils permettent de vérifier la correspondance paramètre effectif-paramètre formel et d'appeler par valeur si nécessaire.

Ils peuvent être considérés comme des ordres exécutés après l'ordre EPR ou être considérés comme des opérandes de EPR.

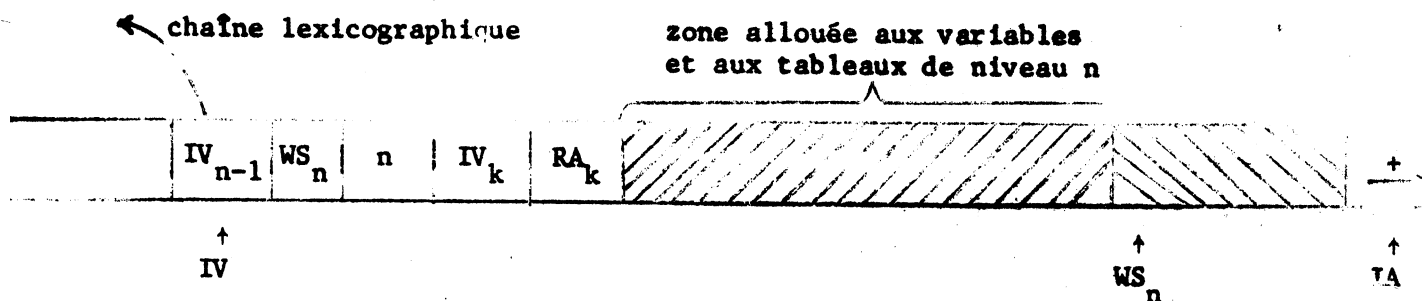
2. Evaluation des programmes objet. Interpréteur Algol.

L'interpréteur Algol décrit ci-après exécute un programme objet qui est directement dérivé de la notation postfixée.

On suppose que la vérification du type des expressions et des instructions est réalisée par le compilateur. Dans le cas contraire, l'interpréteur doit prévoir les vérifications sur les profils qualitatifs. Le langage objet est constitué d'ordres de longueur variable allant de 1 à 5 syllabes. La première syllabe définit le code opération OP. Le programme de contrôle de l'interpréteur détermine la longueur de l'ordre correspondant à chaque code opération, les autres syllabes étant désignées par PAR1, PAR2, PAR3, PAR4.

2.1 Pile d'évaluation.

Données de liaison :



Le mécanisme d'interprétation utilise les données suivantes :

IC : compteur d'instructions du langage objet

IV : pointeur indiquant l'adresse origine de la zone locale du bloc courant

IA : pointeur indiquant la première position libre au sommet de la pile

le vecteur index qui définit la chaîne lexicographique des blocs activés.

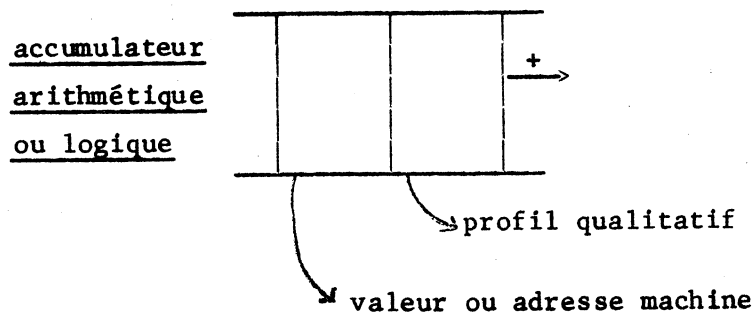
Pour le niveau n, les données de liaison sont :

- 1 - IV_{n-1} valeur du pointeur IV pour le niveau n-1 qui englobe lexicographiquement le niveau n
- 2 - WS_n zone de travail du niveau courant n : adresse du premier accumulateur libre à la suite de la zone locale
- 3 - n numéro de bloc du niveau courant
- 4 - IV_k valeur de IV pour le niveau K qui a activé le niveau courant
- 5 - RA_k adresse de retour au programme objet au niveau k

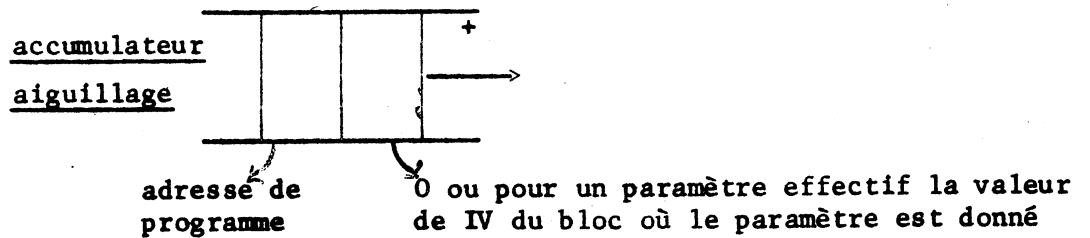
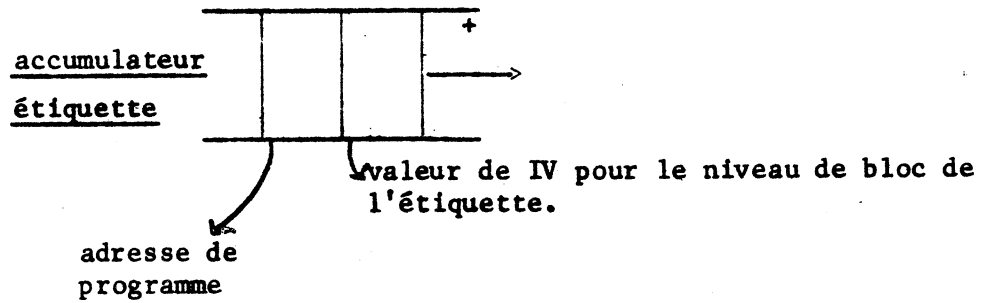
Les deux dernières données de liaison ne sont nécessaires que pour les procédures.

2.2 Accumulateurs.

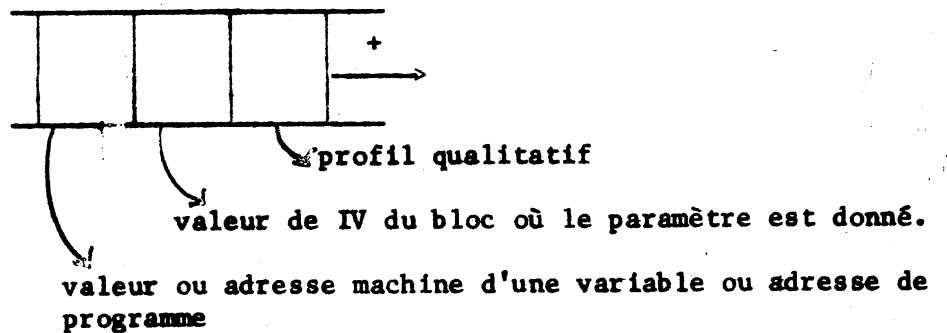
Un accumulateur est constitué par deux mots machine : un pour la valeur (ou l'adresse), l'autre pour le profil qualitatif.



Dans le cas d'un accumulateur-étiquette, il n'y a pas de profil qualitatif : le premier mot contient l'adresse du programme qui représente la valeur de l'étiquette et le deuxième la valeur de IV pour le niveau de bloc de l'étiquette.

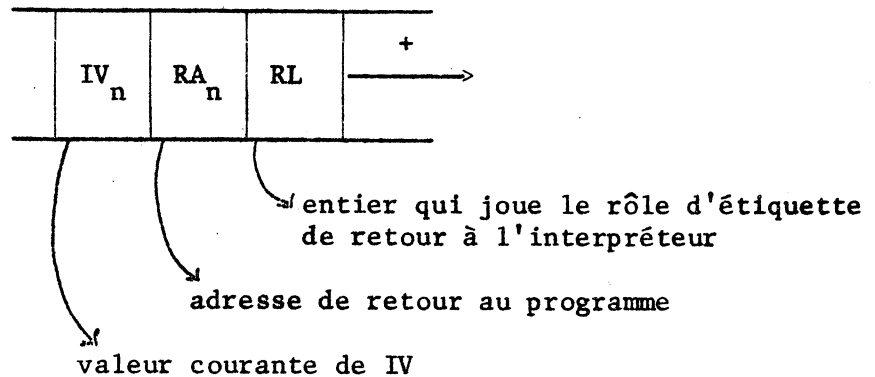


Dans le cas d'un accumulateur forme , il y a 3 mots machine :



Données de liaison empilées à l'activation d'un sous-programme paramètre.

3 données de liaison sont empilées à l'activation d'un sous-programme.



2.3 Langage objet.

Le compilateur attribue les adresses relatives à partir de l'adresse relative 3 pour les blocs et 5 pour les procédures pour tenir compte des données de liaison. Les adresses relatives sont attribuées d'après l'ordre des déclarations ou des paramètres formels selon le schéma suivant :

- 3 mots par paramètre formel.
- 1 mot par variable simple.
- 2 mots par tableau + pour chaque segment de tableau
la longueur de la fonction de rangement.
- 1 mot pour chaque instruction pour.

Lorsqu'un corps de procédure est un bloc non étiqueté, il y a théoriquement 2 niveaux de blocs, celui de la procédure avec les paramètres et celui du bloc correspondant au corps : dans ce cas particulier les variables locales du bloc non étiqueté sont mises à la suite des paramètres et le bloc associé au corps disparaît en tant que niveau.

Les parties adresse du langage objet sont désignées par PAR1, ---, PAR4 suivant le nombre de syllabes.

Une adresse relative représentée par n, p ou une adresse programme représentée par A, a nécessite deux syllabes.

On suppose que chaque syllabe à 8 bits (une adresse de programme est donc équivalente à $256 \times A + a$)

CODE	PAR1	PAR2		
ADD				opérations arithmétiques, logiques et de relation
...				
VVE	A	a	Vraie Valeur Entière	
VVR	A	a	Vraie Valeur Réelle	
VVB	A	a	Vraie Valeur Booléenne	
ADE	n	p	Adresse d'Entier	} n, p adresse relative d'une variable locale
ADR	n	p	Adresse de Réel	
ADB	n	p	Adresse de Booléen	
VAE	n	p	Valeur d'entier	
VAR	n	p	Valeur de Réel	
VAB	n	p	Valeur de Booléen	
AEF	n	p	Adresse d'Entier Formel	} n, p adresse relative d'un accumulateur formel
ARF	n	p	Adresse de Réel Formel	
ABF	n	p	Adresse de Booléen Formel	
VEF	n	p	Valeur d'Entier Formel	
VER	n	p	Valeur de Réel Formel	
VEF	n	p	Valeur de Booléen Formel	
INDV			Valeur d'une variable indiquée	
INDA			Adresse d'une variable indiquée	
3IC	A	a	Saut Inconditionnel	
3CF	A	a	Saut Conditionnel si Faux	

CODE	PAR1	PAR2	PAR3		
EBL	p	r		Entrée Bloc	n : numéro de bloc
TAB	p	nt		Création de Tableau	
FBL				Fin de bloc	
EPR	n	np	r	Entrée procédure	r : nombre mémoires à récursser
EPR				Fin procédure	
SFO	A	a	np	Saut fonction	nt : nombre de tableaux à créer
SPR	A	a	np	Saut Procédure	
SFFO	n	p	np	Saut Formel Fonction	np : nombre de paramètre A, a : adresse de progra
SFFO	n	p	np	Saut Formel Procédure	

Ordres sur les paramètres.

PVE	n	p		Paramètre Variable Entière
PCE	A	a	(1)	Paramètre Constante Entière
PTE	n	p		Paramètre Constante Entière
PVSE	A	a	(3)	Paramètre Variable SP Entière
PFE	A	a	(2)	Paramètre Fonction Entière
PEA	A	a	(3)	Paramètre Expression Arithmétique
PVR	n	p		} Ordres similaires pour le type réel
PCR	A	a	(1)	
PTR	n	p		
PVSR	A	a	(3)	
PFR	A	a	(2)	
PVB	n	p		} Ordres similaires pour le type booléen
PCB	A	a	(1)	
PTB	n	p		
PFB	A	a	(2)	
PEB	A	a	(3)	

PPRO	A	a	(2)
PAIG	A	a	(2)
PLAB	A	a	(4)
PEXD	A	a	(3)
PFOL	n	p	

- (1) A, a : est la valeur de la constante
- (2) A, a : est une adresse programme
- (3) A, a : est une adresse programme de sous-programme.
- (4) adresse programme où l'étiquette est définie comme n, A, a

Ordres de test et d'appel par valeur des paramètres.

TA	Test arithmétique
TB	Test Booléen
TTE	Test Tableau Entier
TFE	Test Fonction Entière
TTR	} Ordres similaires pour tableaux et fonctions de type Réel et Booléen
TFR	
TTB	
TFB	
TFB	
TPRO	Test procédure
TLAB	Test label
TAIG	Test aiguillage
TAVE	Test et Appel Valeur Entière
TAVR	Test et Appel Valeur Réelle
TAVB	Test et Appel Valeur Booléenne.

Ordres particuliers.

RET				Retour d'un sous-programme paramètre
LAB	A	a	n	Label n : numéro de bloc ; A, a : adresse programme
AIG	A	a		Aiguillage
LFAI	n	p		Label Formel ou aiguillage
VIA				Valeur d'un Indicateur d'aiguillage.
ALL				Aller a

Ordres relatifs à l'instruction faire.

FAIRE	A	a	B	b	{ Faire - Active le SP instruction qui suit <u>Faire</u> ; B, b est l'adresse de retour. Retour du SP Instruction qui suit <u>faire</u> (p est l'adresse relative de la variable cale qui donne l'adresse de retour. Activation de SP } pour une variable con- Retour de SP } lée indiquée Valeur de la variable contrôlée sans dés- pilage de l'adresse. Saut si terminé à l'adresse A, a
RSPF	P				
ASP	A	a			
RSP					
VAL					
SST	A	a			

2.4 Exemples de séquences de programmes objets.

1) Expressions et instructions d'affectation.

Considérons l'instruction d'affectation suivante dans un corps de procédure fonction de type entier d'identificateur F :

$F := A + (\text{si } B > C \text{ alors } D-E \text{ sinon } G)$

En supposant que les identificateurs A, B, C sont déclarés au niveau et F au niveau 4, cette instruction est traduite par la séquence suivante :

ADE	4,0
VAR	3,9
VAR	3,10
VAR	3,11
SUP	>
SCF	A,a
RVA	3,12
RVA	3,13
SOU	-
SIC	B,b

A	a	RVA	3,14	
B	b	ADD		+
		ASF		:=F

Remarque 1 :

L'adresse de l'accumulateur réservé dans la pile lors de l'activation de la fonction est indiquée avec une adresse relative de 0. L'adresse relative n, 0 correspond en fait à l'adresse n, -2 pour l'interpréteur. L'affectation à un identificateur de fonction est semblable à l'affectation de variable. Toutefois, lors de l'affectation d'une valeur à un identificateur de fonction, cette valeur n'est pas rangée dans la zone de travail locale mais dans un accumulateur résultat : l'ordre d'affectation est donc différent et construit également au profil qualitatif dans l'accumulateur.

Remarque 2 :

Lorsqu'un ordre formel provoque l'activation d'une fonction sans paramètres jouant le rôle de paramètre effectif pour un paramètre formel avec une spécification de type, l'activation commence par créer le profil qualitatif dans l'accumulateur résultat. Ceci permet de traiter les transferts de type arithmétiques

2) Tableaux.

La variable locale τ , associée à chaque tableau et qui représente le tableau correspondant dans le programme objet, est représentée par deux mémoires qui contiennent pendant l'exécution :

- l'adresse de la fonction topographique
- l'adresse du tableau.

On suppose que le compilateur a évalué l'encombrement de la fonction topographique et a réservé la zone nécessaire pour les variables τ associées aux tableaux d'un même segment.

3) Activation des blocs et des procédures.

L'activation d'un bloc ne soulève pas de problèmes particuliers. Nous insisterons donc surtout sur l'activation des procédures.

Pour une procédure fonction, l'appel de la fonction est suivi d'un ordre REJ qui supprimera l'accumulateur résultat inutile dans ce cas.

Une procédure est activée en deux étapes :

a) l'ordre SPR (Saut Procédure) empile les données de liaison relatives au niveau courant contenant l'appel de la procédure. Ensuite, l'ordre SPR provoque l'exécution des ordres de transfert des paramètres qui créent les accumulateurs formels avec les caractérisations dynamiques des paramètres effectifs appelés par nom. L'exécution de ces ordres de transfert est contrôlée par le nombre de paramètres figurant dans l'appel. Lorsque les transferts sont terminés, il y a un saut vers le premier ordre du corps de procédure et l'exécution continue normalement.

b) l'ordre EPR (Entrée Procédure) premier ordre de la procédure, vérifie le nombre de paramètres effectifs et a le même effet que l'ordre EBL (Entrée Bloc) : empilage des données de liaison, etc... Toutefois dans le cas d'une activation d'une procédure paramètre, l'ordre EPR effectue d'abord la remise à jour du vecteur index.

4) Correspondance paramètres effectifs, paramètres formels. Appel par nom.

L'ordre EPR est suivi par des ordres qui correspondent à la traduction des parties valeur et spécification de la tête de procédure. Ces ordres vérifient la correspondance entre les paramètres effectifs et formels, dans le cas d'un paramètre appelé par valeur ils rangent la valeur dans l'accumulateur formel associé

Les transferts de type ne sont autorisés que dans le cas de paramètres spécifiés de type réel ou entier : la valeur est alors rangée avec le type associé. Dans tous les autres cas, les spécifications doivent concorder exactement.

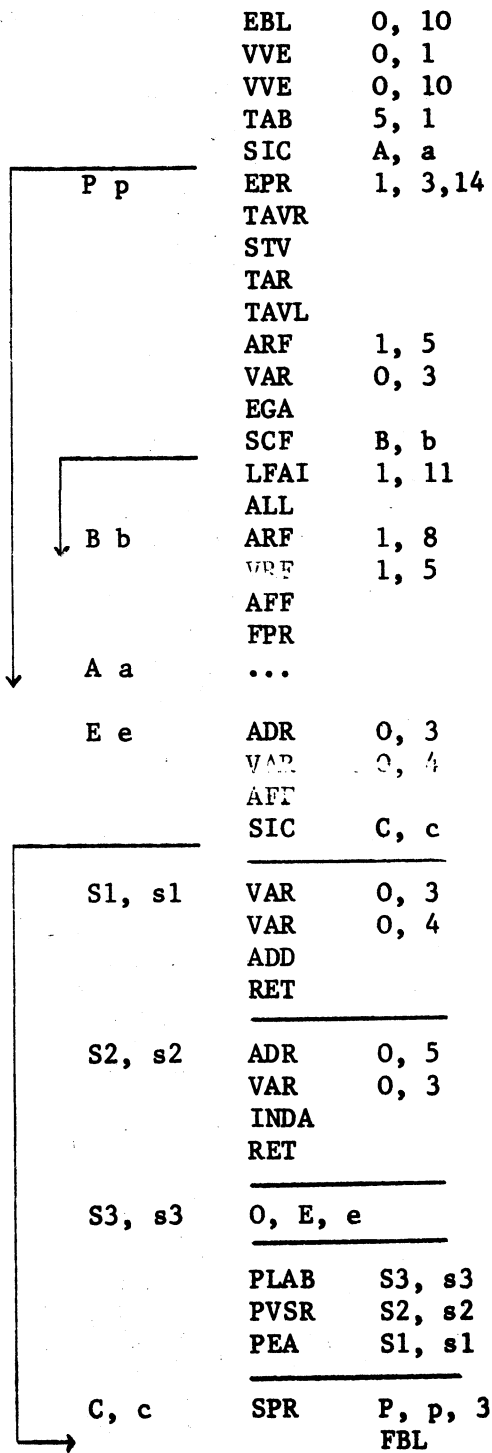
Il n'y a donc qu'un seul ordre pour les spécifications de type entier et réel : l'ordre TA. Toutefois, cet ordre ne réalise pas un test complet : si le paramètre apparaît dans la partie gauche d'une instruction d'affectation, l'ordre formel correspondant doit vérifier que le paramètre effectif est une variable avec le type spécifié.

Les ordres de test des paramètres effectifs, de même que les ordres de transfert des paramètres ne comportent pas d'adresse d'accumulateur formel : ils se suivent dans le même ordre que les paramètres dans la liste. L'adresse de l'accumulateur formel pour cette catégorie d'ordres est indiquée par la valeur du pointeur IAF qui progresse après l'exécution d'un de ces ordres. L'utilisation de ce pointeur soulève une difficulté dans le cas d'une procédure récursive avec un paramètre appelé par valeur : si un sous-programme paramètre est appelé on doit activer une procédure et dans ce cas il faut empiler la valeur du pointeur IAF avant l'appel du sous-programme paramètre.

Dans l'interpréteur décrit ci-après, la partie de l'interpréteur qui détermine la valeur d'un paramètre avec une spécification de type n'est pas écrite sous forme de sous-programme. Les ordres de test de l'appel par valeur exécutent l'ordre formel sur le type de la valeur qui laissera la valeur au sommet de la pile. Chaque ordre de test et d'appel par valeur est suivi par un ordre de stockage de la valeur (STV) qui stocke la valeur dans l'accumulateur formel dont l'adresse est sur la pile. Cette façon de faire est inefficace, spécialement dans le cas où le paramètre effectif est une variable simple : il suffirait de répéter ce qui est fait dans l'ordre formel et d'éviter la nécessité de l'ordre STV.

Dans l'exemple suivant, on trouve un ordre de test et d'appel par valeur d'une étiquette (TAVL) qui exécute ce travail pour les paramètres spécifiés étiquette et appelés par valeur.

```
debut reel A, B ; tableau T[1 : 10] ;  
  procedure P(X, Y, E) ; reel X, Y , étiquette E ; valeur X, E ;  
    debut  
      si X=A alors allera E ;  
      Y := X  
    fin ;  
  .....  
  L : A := B ;  
  P(A+B, T[A], L)  
fin
```



} X
Y
E

Test ou test et appel par valeur
des paramètres effectifs

=

:=

A := B

SP : A+B

SP : T[A]

(1)

Ordre de transferts
des paramètres

Saut procédure

(1) Lorsqu'un paramètre effectif est une étiquette, il ne peut pas être mis d l'ordre de transfert du paramètre effectif : en effet tous ces ordres ont une longueur fixe de 3 syllabes pour des raisons de commodité. Il en résulte que l'adresse programme et le numéro de bloc d'un paramètre effectif étiquette ap raissent au milieu des sous-programmes paramètres.

Si le paramètre est un nom, il apparaît directement dans l'ordre de transfert

Par exemple :

P(A, B, L) est traduit :

	SIC	C, c	
S1 s1	O, E, e		L
	PLAB, S1, s1		B
	PVR O, 4		A
	PVR O, 3		
C c	SPR	P, p, 3	

5) Etiquette. Expression de désignation. Instruction aller a.

L'ordre LAB utilisé pour une étiquette, l'ordre LFAI pour un paramé étiquette, l'évaluation d'une expression de désignation placent au sommet de pile l'adresse programme correspondant à cette étiquette et la valeur IV_N de pour le niveau de bloc de cette étiquette.

Un ordre ALL prend la valeur de l'étiquette au sommet de la pile. : l'étiquette n'est pas locale au bloc courant, l'ordre ALL doit remettre à jo les pointeurs IV et IA ainsi que le vecteur index (l'étiquette pouvant être paramètre).

6) Aiguillage.

Déclaration d'aiguillage.

La déclaration d'aiguillage est traitée de la même façon que la déclaration de procédure. Il y a un sous-programme pour chaque élément de la liste. Un élément est compilé comme un sous-programme paramètre effectif et se termine par un ordre de retour RET.

Ces sous-programmes sont suivis par la liste de leur adresse programme dans l'ordre inverse de la liste d'aiguillages et par le nombre d'éléments. L'adresse programme de la position contenant le nombre d'éléments remplace l'identificateur d'aiguillage dans le programme objet.

Indicateur d'aiguillage.

Les deux ordres LAIG et LFAI empilent l'adresse d'un aiguillage (adresse programme de la position contenant le nombre d'éléments).

L'ordre LFAI est utilisé pour un aiguillage paramètre et empile en plus de l'adresse de l'aiguillage, la valeur du pointeur IV correspondant au niveau pour lequel l'aiguillage est paramètre effectif.

L'ordre IAIG est utilisé pour un aiguillage dans la portée de sa déclaration et empile la valeur 0 au lieu de la valeur IV.

L'ordre VIA (Valeur d'un Indicateur d'aiguillage) trouve la valeur de l'expression en indice et l'adresse de l'aiguillage dans les deux accumulateurs sommets de pile et peut ainsi déterminer l'adresse programme du sous-programme associé à l'élément désigné. Le sous-programme est activé comme un paramètre effectif sous-programme, toutefois la remise à jour du vecteur index n'est nécessaire que dans le cas d'un aiguillage paramètre.

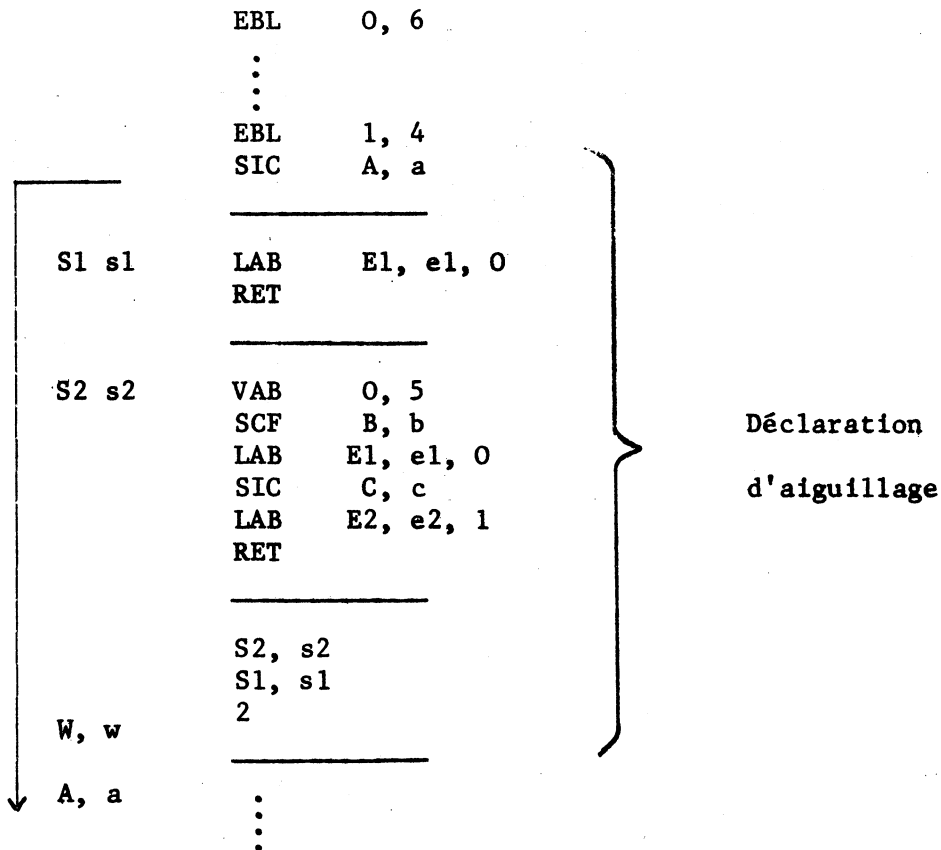
Exemple :

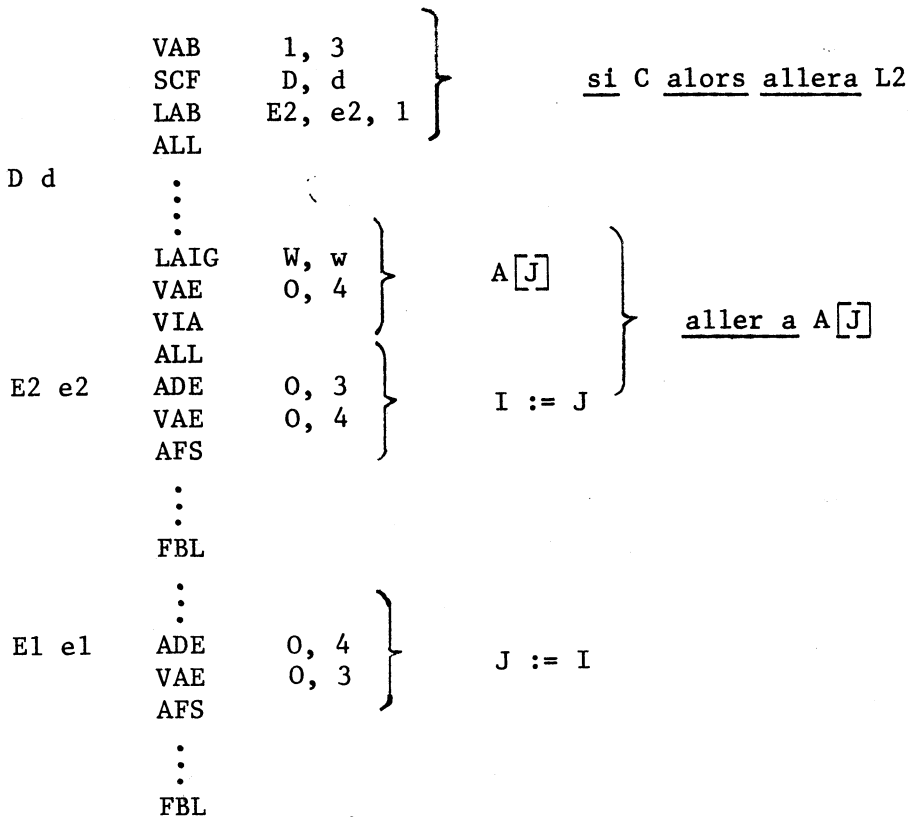
```

debut entier I, J ; Booleen B ;
    ....
    début Booleen C ;
    aiguillage A := L1, si B alors L1 sinon L2 ;
    si C alors aller a L2 ;
    ....
    aller a A [J] ;
    ....
    L2 : I := J
    fin ;
L1 : J := I
fin

```

est traduit par la séquence suivante :





7) Instruction pour.

L'instruction contrôlée par la proposition pour est compilée comme un sous-programme. Les éléments de la liste de pour activent le sous-programme par l'intermédiaire de l'ordre FAIRE. Cet ordre a 2 opérandes : l'adresse programme du sous-programme et l'adresse de retour. Le traitement de l'adresse de retour doit permettre la récursivité : pour cela le compilateur associé à chaque instruction pour une variable locale de même niveau que l'instruction pour. Pendant l'exécution du sous-programme, l'adresse de retour est rangée dans cette mémoire. En effet si on se contentait d'empiler l'adresse de retour, les instructions aller a conduisant à l'extérieur de l'instruction contrôlée créeraient des difficultés.

La variable associée à une instruction pour est désignée par son adresse relative car on sait qu'elle est locale au bloc courant ; cette adresse relative est un opérande de l'ordre RSPF qui termine le sous-programme.

Lorsque la variable contrôlée est une variable indicée, l'évaluation son adresse est compilée comme un sous-programme. Il est alors suffisant d'emporter l'adresse de retour pour permettre la récursivité. Ce sous-programme est activé par l'ordre ASP et se termine par l'ordre de retour RSP.

Éléments de la liste de pour.

La traduction des éléments de la liste de pour est similaire à celle décrite dans le rapport révisé d'Algol 60. L'adresse de la variable contrôlée évaluée pour chaque affectation.

Un ordre ASP est utilisé dans le cas d'une variable contrôlée indicée.

Éléments avec pas et jusqu'à.

pour V := A pas B jusqu'à C faire

L'adresse de la variable contrôlée peut être trouvée dans la pile pour un sous-programme : un ordre spécial, VAL, est utilisé pour exécuter V := V+B. L'ordre VAL empile la valeur d'une variable dont l'adresse est empilée sans détruire cette adresse. De plus, on effectue une affectation multiple qui laisse la valeur dans la pile afin de pouvoir évaluer V-C.

L'ordre SST, qui précède l'ordre FAIRE, trouve les valeurs de V-C et dans la pile et teste les conditions de fin d'itération. Si B est une expression il est traduit par un sous-programme activé par ASP et qui se termine par l'ordre RSP.

Le pas est évalué deux fois comme il est indiqué dans le rapport Algol 60 mais on suppose que l'évaluation de C ne provoque pas d'effet de bord sur V dans le cas où V est une variable indicée.

Exemple :

pour T[I] := EA1, EA2 tant que EB, EA3 pas EA4 jusqua EA5 faire INST

est traduit par la séquence suivante où l'on a utilisé les noms symboliques des variables au lieu des adresses relatives

V v	SIC A, a ADE T VAE I INDA RSP	}	adresse de T[I]
-----	---	---	-----------------

A a ASP V, v

Evaluation de
l'expression EA1

E1 e1 AFS
 FAIRE S, s, E1, e1
 ASP V, v

Evaluation de
l'expression EA2

AFS
 Evaluation de
l'expression EB

E2 e2 SCF E2, e2
 FAIRE S, s, E1, e1
 ASP V, v

Evaluation de
l'expression EA3

B b SIC L, 1
 Evaluation de
l'expression EA4

```

DEBUT COMMENTAIRE INTERPRETEUR ALGOL ;
ENTIER IC, IA, IV, OP, PAR1, PAR2, PAR3, PAR4, IV1, IAF, J, K,
NPA, NP, VALP, ADRESSE, RL, TYPE, ADRESSE ETIQUETTE, A,
SWITCH, ADRESSE RETOUR, VAL REEL, VAL ENTIER, VAL BOOL,
AD ENTIER, AD REEL, AD BOOL, SSP AD ENTIER, SSP AD REEL,
SSP AD BOOL, VAL CONST ENTIER, VAL CONST REEL, VAL CONST BOOL,
TAB ENTIER, TAB REEL, TAB BOOL, FONCT ENTIER, FONCT REEL,
FONCT BOOL, PROCEDURE, SSP EXP ARITH, SSP EXP BOOL,
SSP EXP DESIGN, AIGU, ETIQ, INDEFINI, NINDICES,
DERNIERE AD, AD MAP FONCT, NBRE TAB, NBRE DELEMENTS ;
BOOLEEN CLE, CLE FONCT ;
PROCEDURE ERREUR(N) ; ..... ;
REEL PROCEDURE ADD FLOT(X) ; ..... ;
ENTIER PROCEDURE ADD ENT(X) ; ..... ;
REEL PROCEDURE NEG FLOT(X) ; ..... ;
REEL PROCEDURE FLOT(X) ; ..... ;
ENTIER PROCEDURE NEG ENT(X) ; ..... ;

```

```

PROCEDURE ACTIVER(ER) ; ENTIER ER ;
  DEBUT COMMENTAIRE PREPARATION DE L ACTIVATION D UN SOUS
  PROGRAMME DE PARAMETRE. IAF POINTE L'ACCUMULATEUR FORMEL
  CORRESPONDANT. ON EMPILE ER QUI TRANSMETTRA UNE ETIQUETTE
  INTERNE A L INTERPRETEUR AU DERNIER ORDRE DU SOUS PROGRAMME
  (RET) ;
  PILE[IA]:=IV ;
  PILE[IA+1]:=IC ;
  PILE[IA+2]:=ER ; IA:=IA+3 ;
  IV:=PILE[IAF+1] ; IC:=PILE[IAF] ;
  MISE A JOUR DISPLAY
  FIN ACTIVER ;

```

```

PROCEDURE MISE A JOUR DISPLAY ;
  DEBUT ENTIER NB, IVN ; IVN:=IV ;
  POUR NB:=PILE[IV+2] PAS -1 JUSQU A 1 FAIRE
  DEBUT DISPLAY[NB]:=IVN ; IVN:=PILE[IVN] FIN ;
  FIN MISE A JOUR DISPLAY ;

```

```

ENTIER PROCEDURE ADRESSE ABSOLUE ;
  ADRESSE ABSOLUE:=DISPLAY[PAR1]+( SI PAR2=0 ALORS -2
  SINON PAR2) ;

```

```

BOOLEEN PROCEDURE EFFECTIF(MOD) ; ENTIER MOD ;
  EFFECTIF:=PILE[IAF+2]=MOD ;
  COMMENTAIRE TEST D UN ACCU FORMEL ;

```

```

PROCEDURE CHANGER TYPE(TYPE, INDEX) ; ENTIER TYPE, INDEX ;
  PILE[INDEX]:= SI TYPE=VAL REEL ALORS
  FLOT(PILE[INDEX]) SINON
  ENTIER(PILE[INDEX]+0.5) ;

```

```

ENTIER TABLEAU P[1:1000], PILE[0:500], DISPLAY[0:16],

```

NADR[0:] ;
AIGUILLAGE RETOUR:=RETOUR1,RETOUR2,RETOUR3,RETOUR4,
 RETOURS,RETOUR6,RETOUR7,RETOUR8 ;
AIGUILLAGE NB ADRESSE:=ADR0,ADR1,ADR2,ADR3,ADR4 ;
AIGUILLAGE OPERATION:=ADD,SOU,MUL,.....

.....;
COMMENTAIRE CHACUNE DES ETIQUETTES DE LA LISTE D AIGUILLAGE EST
 LE NOM D UN ORDRE ET ETIQUETTE LE SOUS PROGRAMME
 CORRESPONDANT DE L INTERPRETEUR ;

.....;
 IC:=IV:=IA:=0 ;

NOYAU DE CONTROLE:
 E: OP:=P[IC] ; ALLERA NB ADRESSE[NADR[OP]+1] ;
 ADR4: PAR4:=P[IC+4] ;
 ADR3: PAR3:=P[IC+3] ;
 ADR2: PAR2:=P[IC+2] ;
 ADR1: PAR1:=P[IC+1] ;
 ADR0: IC:=IC+NADR[OP]+1 ;
ALLERA OPERATION[OP] ;
 E1: IA:=IA+1 ; ALLERA E ;
 E2: IA:=IA+2 ; ALLERA E ;
 EM1: IA:=IA-1 ; ALLERA E ;
 EM2: IA:=IA-2 ; ALLERA E ;
 FIN DU NOYAU DE CONTROLE: ;

ORDRES ARITHMETIQUES ET LOGIQUES:
 ADD: SI PILE[IA-1]=PILE[IA-3] ALORS
ALLERA SI PILE[IA-1]=VAL ENTIER ALORS ADDI
SINON ADDR ;
SI PILE[IA-1]=VAL ENTIER ALORS
 PILE[IA-2]:=FLOT(PILE[IA-2])
SINON DEBUT PILE[IA-4]:=FLOT(PILE[IA-4]) ;
 PILE[IA-3]:=VAL REEL
FIN ;
 ADDR: PILE[IA-4]:=ADD FLOT(PILE[IA-4],PILE[IA-2]) ;
ALLERA EM2 ;
 ADDI: PILE[IA-4]:=ADD ENT(PILE[IA-4],PILE[IA-2]) ;
ALLERA EM2 ;
 NEG: PILE[IA-2]:= SI PILE[IA-1]=VAL ENTIER ALORS
 NEG ENT(PILE[IA-2]) SINON
 NEG FLOT(PILE[IA-2]) ;
ALLERA E ;
COMMENTAIRE L EXECUTION DES AUTRES ORDRES ARITHMETIQUES
 OU LOGIQUES OU RELATIONS EST ANALOGUE ;

ORDRES SUR ADRESSE:
 ADE: ADRESSE D ENTIER: PILE[IA+1]:=AD ENTIER ;
 L: PILE[IA]:=ADRESSE ABSOLUE ; ALLERA E2 ;
 ADR: ADRESSE DE REEL: PILE[IA+1]:=AD REEL ; ALLERA L ;
 ADB: ADRESSE DE BOOLEEN: PILE[IA+1]:=AD BOOL ; ALLERA L ;

ORDRES SUR VALEUR:

VAE: PILE[IA+1]:=VAL ENTIER ;
 LO: PILE[IA]:=PILE[ADRESSE ABSOLUE] ; ALLERA E2 ;
 VAR: PILE[IA+1]:=VAL REEL ; ALLERA LO ;
 VAB: PILE[IA+1]:=VAL BOOL ; ALLERA LO ;

SAUTS:

SIC: SAUT INCONDITIONNEL: IC:=256+PAR1+PAR2 ;
ALLERA E ;
 SCF: SAUT SI FAUX: IA:=IA-2 ;
ALLERA SI PILE[IA]=FAUX ALORS SIC SINON E ;

AFFECTATIONS:

AS: CLE:=FAUX ; ALLERA AS1 ;
 ASM: AFFECTATION MULTIPLE: CLE:=VRAI ;
 AS1: CLE FONCT:=FAUX ;
 AS2: ADRESSE:=PILE[IA-4] ;
 TYPE:=SI PILE[IA-3]=AD ENTIER ALORS
 VAL ENTIER SINON VAL REEL ;
 SI PILE[IA-1] ≠ TYPE ALORS
 CHANGER TYPE(TYPE, IA-2) ;
 PILE[ADRESSE]:=PILE[IA-2] ;
 SI CLE ALORS DEBUT PILE[IA-4]:=PILE[IA-2] ;
 PILE[IA-3]:=TYPE
 FIN RECOPIE DE L ACCUMULATEUR VALEUR ;
 SI CLE FONCT ALORS
 DEBUT SI PILE[ADRESSE+1]=TYPE ALORS
ALLERA AS3 ;
 SI PILE[ADRESSE+1]=0 ALORS
 PILE[ADRESSE+1]:=TYPE SINON
 CHANGER TYPE(PILE[ADRESSE+1],ADRESSE) ;
 AS3: FIN ;
 IA:=IA-(SI CLE ALORS 2 SINON 4) ;
ALLERA E ;
 ASF: AFFECTATION FONCTION: CLE:=FAUX ; ALLERA AS4 ;
 ASFM: AFFECTATION FONCTION MULTIPLE: CLE:=VRAI ;
 AS4: CLE FONCT:=VRAI ; ALLERA AS2 ;
 REJ: EFFACEMENT DE L ACCUMULATEUR SOMMET: ALLERA EM2 ;

TABLEAUX ET VARIABLES INDICEES:

TAB: CONSTRUCTION DE TABLEAU:

DERNIERE AD:=IV+PAR1 ;
 AD MAP FONCT:=DERNIERE AD+2 ;
 NBRE TAB:=PAR2 ;
 K:=IA ; IA:=PILE[IV+1] ;

CONSTRUCTION DE LA MAP FONCT:

POUR K:=K PAS -2 JUSQUA IA FAIRE

DEBUT COMMENTAIRE LA FONCTION DE MAPPING EST CONSTRuite
 A PARTIR DES VALEURS EMPILEES DES BORNES ET EST PLACEE A
 PARTIR DE L ADRESSE: AD MAP FONCT. ON FAIT DES VERIFICATIONS
 ON CALCULE LE NOMBRE D ELEMENTS DU TABLEAU ;

```

POUR K = 1 PAS 2 JUSQUA 2*NBRE TAB FAIRE
  DEBUT
  PILE[DERNIERE AD-K+1]:=AD MAP FONCT ;
  PILE[DERNIERE AD-K+2]:=IA ;
  IA:=IA+NBRE D ELEMENTS ;
  FIN ;
PILE[IV+1]:=IA ;
COMMENTAIRE MISE A JOUR DE LA SECONDE DONNEE DE LIAISON
POUR INDiquer LA PREMIERE MEMOIRE LIBRE APRES LES RESERVATIONS
POUR TABLEAU ;
INDV: VALEUR DE VARIABLE INDICEE:
  CLE:= VRAI ; ALLERA INDI ;
INDA: ADRESSE DE VARIABLE INDICEE:
  CLE:= FAUX ;
INDI: NINDICES:=0 ;
  POUR K:=IA-2 TANTQUE PILE[K+1] ≠ AD... FAIRE
  DEBUT SI PILE[K+1]=VAL REEL ALORS
    PILE[K]:=ENTIER(PILE[K]+0.5) ;
    NINDICES:=NINDICES+1 ; IA:=IA-2
  FIN ;
OBTENTION DE L'ADRESSE ;
COMMENTAIRE ON OBTIENT L'ADRESSE A PARTIR DES VALEURS
EMPILEES DES INDICEEES GRACE AUX INFORMATIONS CONTENUES
DANS LA FONCTION DE MAPPING. DIVERSES VERIFICATIONS SONT FAITES
SI CLE ALORS PILE[IA-2]:=ADRESSE SINON
  DEBUT
  PILE[IA-2]:=PILE[ADRESSE] ;
  PILE[IA-1]:=VAL... ;
  FIN ;
  ALLERA E ;

BLOCS ET PROCEDURES:
EBL: ENTREE BLOC:
  PILE[IA]=IV ; DISPLAY[PAR1]:=IV:=IA ;
  PILE[IV+1]:=IA:=IV+PAR2 ;
  PILE[IV+2]:=PAR1 ;
  ALLERA E ;
FBL: FIN DE BLOC:
  IA:=IV ; IV:=PILE[IV] ;
  ALLERA E ;
SFO: SAUT A FONCTION:
  IA:=IA+2 ; PILE[IA-1]:=0 ;
SPR: SAUT A PROCEDURE:
  COMMENTAIRE DEUX DES DONNEES DE LIAISON SONT EMPILEES
  ICI(ADRESSE DE RETOUR AU PROGRAMME OBJET ET VALEUR DE IV
  POUR LE BLOC APPELANT LA PROCEDURE) LES AUTRES DONNEES DE
  LIAISON SONT EMPILEES PAR EPR ;
  PILE[IA+3]:=IV ; PILE[IA+4]:=IC ;

```



```

IVI:=0 ; NPA:=NP:=PAR3 ;
ADRESSE:=256*PAR1+PAR2 ;
IC: IC-1 ;
TRANSFERT DES PARAMETRES:
IC:= SI NP=0 ALORS ADRESSE SINON IC-6 ;
NP:=NP-1 ;
ALLERA E ;
EPR: ENTREE PROCEDURE:
SI NPA ≠ PAR2 ALORS ERREUR(1) ;
SI IV1 ≠ 0 ALORS
    DEBUT IV:=IV1 ; MISE A JOUR DISPLAY FIN ;
PILE[IA]:=DISPLAY[PAR1-1] ;
DISPLAY[PAR1]:=IV:=IA ;
PILE[IV+1]:=IA:=IV+PAR3 ;
PILE[IV+2]:=PAR1 ;
IAF:=IV+5 ;
ALLERA E ;
FPR: FIN DE PROCEDURE:
IC:=PILE[IV+4] ; IA:=IV ; IV:=PILE[IA+3] ;
MISE A JOUR DISPLAY ;
ALLERA E ;

```

CONSTRUCTION PAR LES ORDRES DE TRANSFERT DE PARAMETRE DES ACCUMULATEU
FORMELS CORRESPONDANT AUX PARAMETRES EFFECTIFS:

```

PVE: PARAMETRE EFFECTIF ENTIER:
    PILE[IAF+2]:=AD ENTIER ;
    L1: PILE[IAF]:=ADRESSE ABSOLUE ;
    L2: IAF:=IAF+3 ;
    ALLERA TRANSFERT DES PARAMETRES ;
PCE: PARAMETRE EFFECTIF CONSTANTE ENTIERE:
    PILE[IAF+2]:=VAL CONST ENTIER ;
    L4: PILE[IAF]:=P[256 * PAR 1 + PAR 2] ;
    ALLERA L2 ;
PTE: PARAMETRE EFFECTIF TABLEAU ENTIER:
    PILE[IAF+2]:=TAB ENTIER ;
    ALLERA L1 ;
PVSE: PARAMETRE EFFECTIF VARIABLE INDICEE ENTIER:
    PILE[IAF+2]:=SSP AD ENTIER ;
    L3: PILE[IAF+1]:=IV ;
    PILE[IAF]:=256*PAR1+PAR2 ;
    ALLERA L2 ;
PFE: PARAMETRE EFFECTIF FONCTION ENTIERE:
    PILE[IAF+2]:=FONCT ENTIER ;
    ALLERA L3 ;
PEA: PARAMETRE EFFECTIF EXPRESSION ARITHMETIQUE:
    PILE[IAF+2]:=SSP EXP ARITH ;
    ALLERA L3 ;
PVR: PILE[IAF+2]:=AD REEL ; ALLERA L1 ;
PCR: PILE[IAF+2]:=VAL CONST REEL ; ALLERA L4 ;
PTR: PILE[IAF+2]:=TAB REEL ; ALLERA L1 ;
PVSR: PILE[IAF+2]:=SSP AD REEL ; ALLERA L3 ;
PFR: PILE[IAF+2]:=FONCT REEL ; ALLERA L3 ;

```

PVB: PILE[IAF+2]:=AD BOOL ; ALLERA L1 ;
 PCB: PILE[IAF+2]:=VAL CONST BOOL ; ALLERA L4 ;
 PTB: PILE[IAF+2]:=TAB BOOL ; ALLERA L1 ;
 PVS: PILE[IAF+2]:=SSP AD BOOL ; ALLERA L3 ;
 PFB: PILE[IAF+2]:=FONCT BOOL ; ALLERA L3 ;
 PEB: PILE[IAF+2]:=SSP EXP BOOL ; ALLERA L3 ;
 PPRO: PARAMETRE EFFECTIF PROCEDURE:
 PILE[IAF+2]:=PROCEDURE ;
 ALLERA L3 ;
 PAIG: PARAMETRE EFFECTIF AIGUILLAGE:
 PILE[IAF+2]:=AIGU ;
 ALLERA L3 ;
 PETI: PARAMETRE EFFECTIF ETIQUETTE:
 PILE[IAF+2]:=ETIQ ;
 ADRESSE ETIQUETTE:=256*PARI+PAR2 ;
 PILE[IAF+1]:=DISPLAY[P[ADRESSE ETIQUETTE]] ;
 PILE[IAF]:=256*P[ADRESSE ETIQUETTE+1]+P[ADRESSE ETIQUETTE
 +2] ;
 ALLERA L2 ;
 PEXD: PARAMETRE EXPRESSION DE DESIGNATION:
 PILE[IAF+2]:=SSP EXP DESIGN ;
 ALLERA L3 ;
 PFOL: PARAMETRE EFFECTIF EST UN FORMEL:
 A:=PILE[ADRESSE ABSOLUE+2] ;
 SI A=VAL ENTIER ALORS ALLERA PVE ;
 SI A=VAL REEL ALORS ALLERA PVR ;
 SI A=VAL BOOL ALORS ALLERA PVB ;
 POUR J:=0 PAS 1 JUSQUA 2 FAIRE
 PILE[IAF+J]:=PILE[ADRESSE ABSOLUE+J] ;
 ALLERA L2 ;
 ORDRES DE VERIFICATION DE LA CORRESPONDANCE PARAMETRE EFFECTIF FORMEL
 ET D APPEL PAR VALEUR UTILISES APRES L ENTREE DE PROCEDURE:
 TA: TEST ARITHMETIQUE ;
 SI EFFECTIF(AD ENTIER) OU EFFECTIF(AD REEL) OU
 EFFECTIF(VAL CONST ENTIER) OU EFFECTIF(VAL CONST REEL) OU
 EFFECTIF(VAL ENTIER) OU EFFECTIF(VAL REEL) OU
 EFFECTIF(SSP AD ENTIER) OU EFFECTIF(SSP AD REEL) OU
 EFFECTIF(SSP EXPR ARITH) OU
 EFFECTIF(FONCT REEL) OU EFFECTIF(FONCT ENTIER)
 ALORS ALLERA L5 SINON ERREUR(2) ;
 L5: IAF:=IAF+3 ;
 ALLERA E ;
 TB: TEST BOOLEEN: ;
 COMMENTAIRE CET ORDRE EST ANALOGUE A TA AVEC LE TYPE BOOLEEN ;
 TTE: TEST TABLEAU ENTIER:
 SI EFFECTIF(TAB ENTIER) ALORS
 ALLERA L5 SINON ERREUR(4) ;
 TFE: TEST FONCTION ENTIERE:
 SI EFFECTIF(FONCT ENTIER) ALORS ALLERA L5
 SINON ERREUR(5) ;
 TTR: TEST TABLEAU REEL: ;

TTB: TEST TABLEAU BOOLEEN: ;
 TFR: TEST FONCTION REEL: ;
 TFB: TEST FONCTION BOOLEENNE: ;
 TPRO: TEST PROCEDURE:
 SI EFFECTIF(PROCEDURE) OU EFFECTIF(FONCT REEL)
 OU EFFECTIF(FONCT ENTIER) OU EFFECTIF(FONCT BOOL) ALORS
 ALLERA L5 SINON ERREUR(6) ;
 TETI: TEST ETIQUETTE:
 SI EFFECTIF(ETIQ) OU EFFECTIF(SSP EXP DESIGN)
 ALORS ALLERA L5 SINON ERREUR(7) ;
 TAIG: TEST AIGUILLAGE:
 SI EFFECTIF(AIGU) ALORS ALLERA L5 SINON ERREUR(8) ;

 APPELS PAR VALEUR:
 TAVE: TEST ET APPEL VALEUR ENTIER:
 PILE[IA]:=-IAF ; IA:=-IA+1 ;
 K:=-2 ; ALLERA VALEUR PARAMETRE ARITHMETIQUE ;
 COMMENTAIRE L ADRESSE DE L ACCUMULATEUR FORMEL EST
 EMPILEE POUR UTILISATION PAR RVAL. PUIS IL Y A SAUT A VEF:
 CET ORDRE LAISSERA UNE VALEUR ENTIERE AU SOMMET DE LA PILE ;
 TAVR: TEST ET APPEL VALEUR REEL:
 PILE[IA]:=-IAF ; IA:=-IA+1 ; K:=5 ;
 ALLERA VALEUR PARAMETRE ARITHMETIQUE ;
 TAVB: TEST ET APPEL VALEUR BOOLEEN:
 PILE[IA]:=-IAF ; IA:=-IA+1 ;
 ALLERA VBF1 ;
 RVAL: RANGEMENT VALEUR :
 IAF:=PILE[IA-3] ;
 PILE[IAF]:=PILE[IA-2] ;
 PILE[IAF+2]:=PILE[IA-1] ; IA:=-IA-3 ;
 ALLERA L5 ;
 COMMENTAIRE CET ORDRE SUIT DANS LE PROGRAMME OBJET LES
 ORDRES TAVE, TAVB, TAVR ;
 TARETI: TEST APPEL ET RANGEMENT VALEUR ETIQUETTE:
 SI EFFECTIF(ETIQ) ALORS ALLERA L5 ;
 SI EFFECTIF(SSP EXP DESIGN) ALORS
 DEBUT
 STACK[IA]:=-IAF ; IA:=-IA+1 ;
 ACTIVER(8) ;
 ALLERA E
 FIN APRES EXECUTION DU SOUS PROGRAMME DE PARAMETRE
 EFFECTIF, L ORDRE RET PROVOQUE UN SAUT A RETOUR8 ;
 ERREUR(9) ;
 RETOUR8: RANGER ETIQUETTE:
 IAF:=PILE[IA-3] ;
 PILE[IAF]:=PILE[IA-2] ;
 PILE[IAF+1]:=PILE[IA-1] ;
 PILE[IAF+2]:=ETIQ ;
 ALLERA L5 ;

 ORDRE DE RETOUR TERMINANT UN SOUS PROGRAMME DE PARAMETRE EFFECTIF:
 RET: RL:=PILE[IA-3] ; IC:=PILE[IA-4] ; IV:=PILE[IA-5] ;

PILE[IA-5]:=PILE[IA-2] ; PILE[IA-4]:=PILE[IA-1] ;
COMMENTAIRE L EXECUTION DU SOUS PROGRAMME AVAIT ETE
 PREPAREE PAR ACTIVER. LE RESULTAT AU SOMMET DE LA PILE EST
 DEPLACE APRES PRELEVEMENT DES DONNEES DE LIAISON ;
 MISE A JOUR DISPLAY ; IA:=IA-3 ;
ALLERA RETOUR[RL] ;

ORDRES SUR FORMELS :

EMPILER ADRESSE :

AEF: ADRESSE ENTIER FORMEL :

IAF:=ADRESSE ABSOLUE ;
SI EFFECTIF(AD ENTIER) OU EFFECTIF(TAB ENTIER) ALORS
DEBUT PILE[IA+1]:=AD ENTIER ;
 LF1: PILE[IA]:=PILE[IAF] ; IA:=IA+2 ;
ALLERA E
 FIN ;
SI EFFECTIF(VAL ENTIER) ALORS
DEBUT PILE[IA+1]:=AD ENTIER ;
 LF2: PILE[IA]:=IAF ; IA:=IA+2 ;
ALLERA E
 FIN ;
SI EFFECTIF(SSP AD ENTIER) ALORS
DEBUT ACTIVER(1) ; ALLERA E FIN APRES
 EXECUTION DU SOUS PROGRAMME L ORDRE DE RETOUR
 PROVOQUE UN SAUT A RETOUR1 ;

ERREUR(10) ;

ARF: ADRESSE ENTIER FORMEL :

ABF: ADRESSE BOOLEEN FORMEL :

RETOUR1: ALLERA E ;

EMPILER VALEUR :

VEF: VALEUR ENTIER FORMEL ; K:=2 ; ALLERA LF3 ;

VRF: VALEUR REEL FORMEL ; K:=5 ;

LF3: IAF:=ADRESSE ABSOLUE ;

VALEUR PARAMETRE ARITHMETIQUE :

SI EFFECTIF(VAL CONST ENTIER) OU
EFFECTIF(VAL ENTIER) ALORS

DEBUT

VALP:= SI K=2 ALORS PILE[IAF]

SINON FLOT(PILE[IAF]) ;

LF31: PILE[IA+1]:= SI K=2 ALORS VAL ENTIER

SINON VAL REEL ;

PILE[IA]=VALP ; IA:=IA+2 ;

ALLERA E

FIN ;

SI EFFECTIF(VAL CONST REEL) OU
EFFECTIF(VAL REEL) ALORS

DEBUT

VALP:= SI K=2 ALORS ENTIER(PILE[IAF]+0.5)

SINON PILE[IAF] ;

```

    ALLERA LF31 ;
    FIN ;
    SI EFFECTIF(AD ENTIER) ALORS
    DEBUT
    VALP:= SI K=2 ALORS PILE[PILE[IAF]] SINON
    FLOT(PILE[PILE[IAF]]) ;
    ALLERA LF31
    FIN ;
    SI EFFECTIF(AD REEL) ALORS
    DEBUT
    VALP:= SI K=2 ALORS ENTIER(PILE[PILE[IAF]]+0.5)
    SINON PILE[PILE[IAF]] ;
    ALLERA LF31
    FIN ;
    SI EFFECTIF(SSP AD ENTIER) ALORS
    DEBUT ACTIVER(K) ; ALLERA E FIN ;
    SI EFFECTIF(SSP AD REEL) ALORS
    DEBUT ACTIVER(K+1) ; ALLERA E FIN ;
    SI EFFECTIF(SSP EXP ARITH) ALORS
    DEBUT ACTIVER(K+2) ; ALLERA E FIN ;
    SI EFFECTIF(FONCT ENTIER) ALORS
    DEBUT SI K=2 ALORS ALLERA LF4 ;
    PILE[IA+1]:=VAL REEL ; ALLERA LF5
    FIN ;
    SI EFFECTIF(FONCT REEL) ALORS
    DEBUT SI K ≠ 2 ALORS ALLERA LF4 ;
    PILE[IA+1]:=VAL ENTIER ; ALLERA LF5
    FIN ;
    ERREUR(11) ;
    LF4: PILE[IA+1]:=0 ;
    LF5: PAR3:=0 ; IA:=IA+2 ;
    ALLERA PROCEDURE FORMELLE ;
    COMMENTAIRE L ACCUMULATEUR DE RESULTAT EST RESERVE ET
    LE NOMBRE DE PARAMETRES(EFFECTIFS) EST FORCE A ZERO ;

```

RETOURS APRES EXECUTIONS DES SOUS PROGRAMMES DE PARAMETRES
EFFECTIFS:

```

    RETOUR2: PILE[IA-2]:=PILE[PILE[IA-2]] ;
    LF6: PILE[IA-1]:=VAL ENTIER ;
    ALLERA E ;
    RETOUR3: PILE[IA-2]:=ENTIER(PILE[PILE[IA-2]]+0.5) ;
    ALLERA LF6 ;
    RETOUR4: SI PILE[IA-1]=VAL ENTIER ALORS ALLERA E ;
    PILE[IA-2]:=ENTIER(PILE[IA-2]+0.5) ;
    ALLERA LF6 ;
    RETOUR5: PILE[IA-2]:=FLOT(PILE[PILE[IA-2]]) ;
    LF7: PILE[IA-1]:=VAL REEL ;
    ALLERA E ;
    RETOUR6: PILE[IA-2]:=PILE[PILE[IA-2]] ;
    ALLERA LF7 ;
    RETOUR7: SI PILE[IA-1]=VAL REEL ALORS ALLERA E ;
    PILE[IA-2]:=FLOT(PILE[IA-2]) ;

```

ALLERA LF7 ;
COMMENTAIRE FIN DES SOUS PROGRAMMES INTERPRETEURS LIES A VEF
 ET VRF ;
 VBF: VALEUR BOOLEEN FORMEL:
 IAF:=ADRESSE ABSOLUE ;
 VBF1:.....; COMMENTAIRE ORDRE ANALOGUE A VEF ET VRF ;

 ORDRES SUR PROCEDURE AIGUILLAGE ETIQUETTE FORMEL:
 SFOF: SAUT FONCTION FORMELLE:
 PILE[IA+1]:=0 ; IA:=IA+2 ;
 SPRF: SAUT PROCEDURE FORMELLE:
 IAF:=ADRESSE ABSOLUE ;
 PROCEDURE FORMELLE:
 PILE[IA+3]:=IV ; PILE[IA+4]:=IC ;
 NPA:=NP:=PAR3 ;
 IV1:=PILE[IAF+1] ; ADRESSE:=PILE[IAF] ;
 IAF:=IA+5 ;
 ALLERA TRANSFERT DES PARAMETRES ;
 COMMENTAIRE EPR COMMENCERA PAR RESTAURER IV ET LE DISPLAY
 A L ETAT QU ILS AVAIENT POUR LE BLOC OU A ETE APPELEE LA
 PROCEDURE AYANT POUR PARAMETRE EFFECTIF LA PROCEDURE
 CONSIDEREE. C EST LA SEULE DIFFERENCE ENTRE SPRF ET SPR ;
 EAIF: ETIQUETTE OU AIGUILLAGE FORMEL:
 IAF:=ADRESSE ABSOLUE ;
 SI EFFECTIF(ETIQ) OU EFFECTIF(AIGU) ALORS
 DEBUT PILE[IA]:=PILE[IAF] ;
 PILE[IA+1]:=PILE[IAF+1] ;
 ALLERA E2
 FIN ;
 EXP DE DESIGNATION: ACTIVER (1) ;
 ALLERA E ;

 ETIQUETTE ET AIGUILLAGE:
 ETI: ETIQUETTE:
 PILE[IA+1]:=DISPLAY[PAR3] ;
 L6: PILE[IA]:=256*PAR1+PAR2 ;
 ALLERA E2 ;
 AIG: AIGUILLAGE:
 PILE[IA+1]:=0 ; ALLERA L6 ;
 VIA: VALEUR INDICATEUR D AIGUILLAGE:
 J:= SI PILE[IA-1]=VAL ENTIER ALORS
 PILE[IA-2] SINON ENTIER(PILE[IA-2]+0.5) ;
 IV1:=PILE[IA-3] ; SWITCH:=PILE[IA-4] ;
 SI J > P[SWITCH] OU J < 0 ALORS
 DEBUT PILE[IA-4]:=INDEFINI ;
 ALLERA EM2
 FIN ;
 ACTIVATION COMME SOUS PROGRAMME DE PARAMETRE:
 PILE[IA-4]:=IV ; PILE[IA-3]:=IC ; PILE[IA-2]:=1 ;
 IA:=IA-1 ;
 SI IV1 ≠ 0 ALORS
 DEBUT IV:=IV1 ; MISE A JOUR DISPLAY FIN POUR

```

        UN AQUILLAGE FORMEL ;
        IC:=PISWITCH-2*J1+256+PISWITCH-2*J+11 ;
        ALLERA E ;
        COMMENTAIRE APRES EXECUTION DU SOUS PROGRAMME
        LE RETOUR A L'INTERPRETEUR EST RETOURI C.A.D.E ;
ALL: ALLERA ;
        SI PILE[IA-2]=INDEFINI ALORS ALLERA EM2 ;
        IC:=PILE[IA-2] ;
        SI IV=PILE[IA-1] ALORS ALLERA EM2 ;
        IV:=PILE[IA-1] ; IA:=PILE[IV+1] ;
        MISE A JOUR DISPLAY ;
        ALLERA E ;

INSTRUCTION POUR:
FAIRE: ACTIVATION DU SOUS PROGRAMME INSTRUCTION:
        ADRESSE:=256*PARI+PAR2 ;
        ADRESSE RETOUR:=256*PAR3+PAR4 ;
        IC:=ADRESSE+1 ; PILE[IV+P[ADRESSE]]:=ADRESSE RETOUR ;
        ALLERA E ;
RSPF: RETOUR SOUS PROGRAMME FAIRE:
        IC:=PILE[IV+PARI] ; ALLERA E ;
ASP: ACTIVER SOUS PROGRAMME:
        PILE[IA]:=IC ; IC:=256*PARI+PAR2 ;
        ALLERA EI ;
RSP: RETOUR SOUS PROGRAMME ACTIVE PAR ASP:
        IC:=PILE[IA-1] ; ALLERA EM1 ;
VAL: VALEUR D'UNE VARIABLE CONTROLEE DONT L'ADRESSE
        EST EMPILEE SANS EFFACER L'ADRESSE:
        PILE[IA]:=PILE[PILE[IA-2]] ;
        PILE[IA+1]:= SI PILE[IA-1]=AD ENTIER
        ALORS VAL ENTIER SINON VAL REEL ;
        ALLERA E2 ;
SST: SAUT SI TERMINE: ELEMENT EPUISE:
        SI PILE[IA-4]+SIGNE(PILE[IA-2]) > 0
        ALORS IC:=256*PARI+PAR2 ;
        ALLERA EM2 ;
FIN INTERPRETEUR

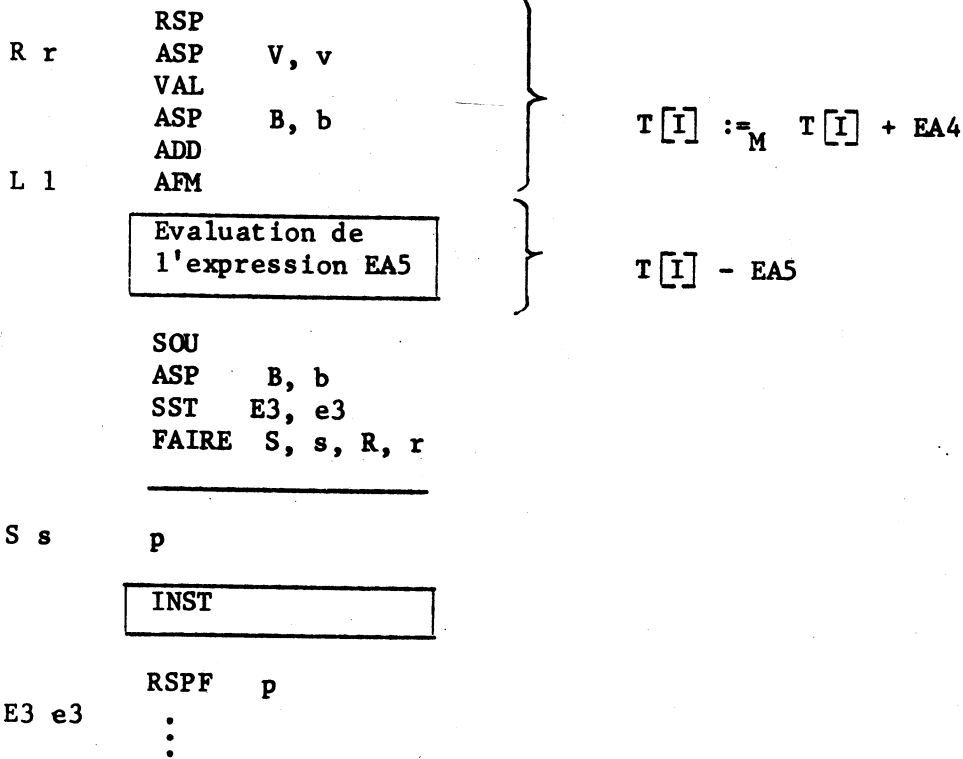
```

II - GENERATION DES PROGRAMMES OBJET.

1. Méthode de la double priorité.

L'algorithme de la double priorité des opérateurs et délimiteurs utilise une pile pour les rangements intermédiaires. Le désempilage d'un opérateur qui correspond à la génération de l'opération associée, est contrôlé par le résultat de la comparaison entre la priorité du délimiteur courant du programme source et la priorité du délimiteur au sommet de la pile.

On a vu précédemment que certains délimiteurs se comportent à la fois comme une parenthèse droite et une parenthèse gauche. La méthode de double priorité considère de tels délimiteurs comme deux délimiteurs distincts ; ils sont lus avec une priorité de parenthèse droite dans le programme source et ensuite empilés avec une priorité de parenthèse gauche.



$$T[I] :=_M T[I] + EA4$$

$$T[I] - EA5$$

L'algorithme décrit ci-après utilise la méthode de la double priorité pour compiler des expressions avec la table suivante.

<u>Délimiteur</u>	Priorité de pile	Priorité de programme source.
[(<u>si</u>	0	/
<u>alors</u>	0	1
<u>sinon</u>	1	2
]) , ω	/	1
=	3	3
⊃	4	4
∨	5	5
∧	6	6
┌	7	7
< < = ≠ > >	8	8
+ -	9	9
× / ÷	10	10
↑	11	11

ω joue le rôle de délimiteur d'expression.

Les délimiteurs d'expression conditionnelle :)] ω et alors doivent avoir une priorité de programme source inférieure ou égale à la priorité de pile de sinon afin de permettre la génération associée à ce dernier délimiteur. Mais le délimiteur sinon ne peut pas délimiter une expression conditionnelle dans le programme source et ne doit pas permettre l'empilage d'un autre sinon : la priorité de pile de sinon doit donc être inférieure à la priorité de programme source de sinon.

L'algorithme suivant compile les expressions selon cette méthode. Cependant, la compilation d'un indicateur de fonction réalise un traitement particulier des listes de paramètres effectifs : lorsqu'on doit compiler une expression, le délimiteur de paramètre effectif lu dans le programme source se comporte toujours comme une parenthèse droite pour générer les opérations qui sont empilées au-dessus du délimiteur "FONCTION".

Ce dernier délimiteur est une parenthèse gauche particulière (désignée par PAR GAUCHE FONCTION) qui caractérise le traitement des paramètres effectifs.

Cette méthode serait plus commode si l'on utilisait une deuxième pile on doit en effet déplacer le délimiteur FONCTION pour ranger au-dessous de ce délimiteur la caractérisation d'un paramètre effectif (ou l'ordre de transfert d'un paramètre effectif). Une autre méthode consisterait à utiliser une variable d'état, la caractérisation du paramètre effectif serait alors empilée de façon habituelle et générée par priorité.

De la même façon, le crochet spécial CRO GAUCHE VARIABLE est utilisé pour le traitement des paramètres effectifs commençant par une variable indiquée on ne peut pas savoir si l'on a une expression ou seulement une variable indiquée jusqu'à la lecture du crochet droit $\}$. Dans le premier cas, on doit prendre la valeur et dans le second l'adresse.

Exemple :

$$A + TA[B+C, D] \times F(E, TB[TC[G, H]], TD[I] + J) \omega$$

sera traduit par la séquence suivante :

	VAL	A	
	ADR	TA	
	VAL	B	
	VAL	C	
	+		
	VAL	D	
	INDVAL		} Ordre d'appel de la fonction
	VALED	F	
	Ad3		
Ad1	ADE	TB	
	ADR	TC	
	VAL	G	
	VAL	H	
	INDVAL		
	INDADR		
	RETOUR		
Ad2	ADR	TD	
	VAL	I	
	INDVAL		
	VAL	J	
	+		
	RETOUR		

	Ad2	Caractérisations des paramètres effectifs :
	(-Ad1)	nom (E) ou adresse programme du sous-programme, positif
	E	pour un sous-programme expression (Ad2), négative pour
Ad3	3	un sous-programme adresse (-Ad1)
	x	nombre de paramètres
	+	

```

DEBUT COMMENTAIRE GENERATION-EXPRESSIONS-DOUBLE PRIORITE ;
  ENTIER I,J,K,L,NP,UN,BIN,OMEGA,PARGAUCHE,PARGAUCHE FONCT,
  PARDROITE,CROGAUCHE,CROGAUCHE PE,CRODROIT,VIRGULE,
  SI,ALORS,ALORSO,SINON,SINONI ;
  ENTIER VAL,AD,INDVAL,INDAD,SAUT SI FAUX,SAUT INCOND,
  RETOUR,VAL INDICATEUR FONCT ;
  COMMENTAIRE LES ENTIERS PRECEDENTS DESIGNENT LES CODES
  DES ORDRES ;
  ENTIER TABLEAU E,S(1:1000),P(1:50) ;
  ENTIER PROCEDURE NOP(X) ; ..... ;
  COMMENTAIRE LA VALEUR DE CETTE PROCEDURE EST LE NOMBRE
  D OPERANDES ASSOCIES A L OPERATEUR X ;
  BOOLEEN PROCEDURE OPERANDE(X) ; ..... ;
  BOOLEEN PROCEDURE OPERATEUR(X) ; ..... ;
  BOOLEEN PROCEDURE IDEN FONCT(X) ; ..... ;
  BOOLEEN PROCEDURE IDEN TAB(X) ; ..... ;
  COMMENTAIRE LA VALEUR DES PROCEDURES PRECEDENTES EST VRAI
  LORSQUE X CORRESPOND AU NOM DE LA FONCTION ;

PROCEDURE EMPILER(X) ; ENTIER X ;
  DEBUT J:=J+1 ; P(J):=X FIN ;

PROCEDURE EMPILER CAR PE(X) ; ENTIER X ;
  DEBUT COMMENTAIRE CETTE PROCEDURE EMPILE UNE CARACTERISATION
  DE PARAMETRE EFFECTIF SOUS LES 2 MEMOIRES DU SOMMET QUI SONT
  RELEVES CE SONT LA PARENTHESE GAUCHE DE FONCTION ET LE
  COMPTEUR DE PARAMETRES QUI EST INCREMENTE ;
  J:=J+1 ; P(J):=P(J-1) ;
  P(J-1):=P(J-2)+1 ; P(J-2):=X
  FIN ;

PROCEDURE GENERER1(X) ; ENTIER X ;
  DEBUT K:=K+1 ; S(K):=X FIN ;
PROCEDURE GENERER(X) PARTIE ADRESSE: (Y) ; ENTIER X,Y ;
  GENERER1(X*4096+Y) ;

PROCEDURE COMPLETER GENERATION ;
  COMMENTAIRE MISE EN PLACE DE LA PARTIE ADRESSE D UN ORDRE OBJET
  DONT L ADRESSE EST EMPILEE ;
  S(P(J-1)):=S(P(J-1))+K+1 ;

ENTIER PROCEDURE PRIORITE PILE(X) ; ... ;

ENTIER PROCEDURE PRIORITE CHAINE(X) ; ..... ;

PROCEDURE ERREUR(N) ; ... ;

.....
I:=J:=K:=0 ;
A: PROGRESSER: I:=I+1 ;

```

EXPRESSION:

```

SI E[I]=OMEGA ALORS ALLERA FIN ;
SI OPERANDE(E[I]) ALORS
  DEBUT
    SI IDEN FONCT(E[I]) ALORS ALLERA FONCTION ;
    SI IDEN TAB(E[I]) ALORS
      DEBUT
        GENERER(AD,E[I]) ;
        SI E[I+1] ≠ CROGAUCHE ALORS ERREUR(1) ;
        EMPILER(CROGAUCHE) ; I:=I+1 ;
        ALLERA A
      FIN ;
    GENERER(VAL,E[I]) ;
  COMPARAISON:
    SI J < 0 ALORS ALLERA A ;
    SI PRIORITE PILE(P[J]) >
      PRIORITE CHAINE(E[I]+1) ALORS
        DEBUT
          SI NOP(P[J])=UN OU NOP(P[J])=BIN ALORS
            DEBUT
              GENERER1(P[J]) ; J:=J-1 ;
              ALLERA COMPARAISON
            FIN ;
          SI P[J] ≠ SINONI ALORS ERREUR(2) ;
          COMPLETER GENERATION ; J:=J-2 ;
          ALLERA COMPARAISON
        FIN ;
    ALLERA A
  FIN ;

```

```

SI OPERATEUR(E[I]) OU E[I]=PARGAUCHE OU E[I]=SI ALORS
  DEBUT EMPILER(E[I]) ; ALLERA A FIN ;

```

DELIMITEUR DE TYPE PARDROITE:

```

SI E[I]=PARDROITE ALORS
  DEBUT SI P[J]=PARGAUCHE FONCT ALORS ALLERA FINAPPEL ;
    SI P[J] ≠ PARGAUCHE ALORS ERREUR(3) ;
    J:=J-1 ; ALLERA COMPARAISON ;
  FIN ;
SI E[I]=CRODROIT ALORS
  DEBUT SI P[J]=CROGAUCHE PE ALORS ALLERA LIST1 ;
    LI: GENERER1(INDVAL) ; J:=J-1 ;
    ALLERA COMPARAISON
  FIN ;
SI E[I]=VIRGULE ALORS
  DEBUT SI P[J]=PARGAUCHE FONCT ALORS ALLERA LIST2 ;
    SI P[J] ≠ CROGAUCHE OU
      P[J] ≠ CROGAUCHE PE ALORS ERREUR(5) ;
    ALLERA A
  FIN ;
SI E[I]=ALORS ALORS
  DEBUT SI P[J] ≠ SI ALORS ERREUR(6) ;

```

```

GENERER(SAUT SI FAUX,0) ; J:=J-1 ;
EMPILER(K) ; EMPILER(ALORSO) ;
ALLERA A
FIN L ADRESSE DE L ORDRE INCOMPLET EST EMPILEE
SOUS LE DELIMITEUR ;
SI E[I]=SINON ALORS
DEBUT SI P[J] ≠ ALORSO ALORS ERREUR(7) ;
GENERER(SAUT INCOND,0) ;
COMPLETER GENERATION ;
P[J-1]:=K ; P[J]:=SINON1 ;
ALLERA A
FIN ;
ERREUR(8) ;

```

FONCTION:

```

GENERER(VAL INDICATEUR FONCT,E[I]) ;
GENERER1(0) ;
EMPILER(K) ; EMPILER(0) ; EMPILER(PARDROITE FONCT) ;
COMMENTAIRE L ADRESSE DE L ORDRE INCOMPLET, ET LE COMPTEUR
DE PARAMETRES SONT EMPILEES ;
SI E[I+1] ≠ PARGAUCHE ALORS
ALLERA FIN APPEL2 ;
PLIST: I:=I+1 ;
LIST3: I:=I+1 ;
SI OPERANDE(E[I]) ET (E[I+1]=VIRGULE OU E[I+1]=
PARDROITE)
ALORS PARAMETRE NOM:
DEBUT EMPILER CAR PE(E[I]) ;
L2: ALLERA SI E[I+1]=VIRGULE ALORS
PLIST SINON FIN APPEL1
FIN ;
SI IDEN TAB(E[I]) ALORS
VARIABLE INDICEE:
DEBUT GENERER(AD,E[I]) ;
SI E[I+1] ≠ CROGAUCHE ALORS ERREUR(9) ;
EMPILER CAR PE(-K) ;
EMPILER(CROGAUCHE PE) ;
I:=I+1 ; ALLERA A
FIN ;
EXPRESSION1: EMPILER CAR PE(K+1) ; ALLERA EXPRESSION ;
LIST1:
SI E[I+1]=VIRGULE OU E[I+1]=PARDROITE ALORS
DEBUT GENERER1(INDAD) ; J:=J-1 ;
GENERER1(RETOUR) ; ALLERA L2
FIN ;
P[J-3]:=-P[J-3] ; ALLERA L1 ;
COMMENTAIRE LA CARACTERISATION DE PARAMETRE
EFFECTIF VARIABLE INDICEE A ETE CHANGEES EN CARACTERISATION
D EXPRESSION AVANT D EFFECTUER LE SAUT A EXPRESSION ;
LIST2: GENERER1(RETOUR) ; ALLERA LIST3 ;
FIN APPEL1: I:=I+1 ; ALLERA FIN APPEL2 ;

```

```

FIN APPEL : GENERER1(RETOUR) ;
FIN APPEL2 :
    NP := P[J-1] ;
    POUR L := 1 PAS 1 JUSQUA NP FAIRE
        GENERER1(P[J-1-L]) ;
        COMMENTAIRE DESEMPILAGE DES CARACTERISATIONS ET DU
        NOMBRE DE PARAMETRES ;
    J := J-(NP+1) ;
    COMPLETER GENERATION ;
    GENERER1(NP) ;
    J := J-2 ; ALLERA COMPARAISON ;
FIN :
FIN

```

2. Méthode des procédures syntaxiques.

Le programme suivant décrit un ensemble de procédures syntaxiques pour la compilation des expressions et des instructions d'affectation Algol.

Pour l'écriture de ces procédures, il est nécessaire de transformer la forme de génération de la description syntaxique d'Algol en une forme de reconnaissance selon le principe suivant :

Une définition telle que :

<TERME ARITH> ::= <FACTEUR ARITH> | <TERME ARITH> <OPMUL> <FACTEUR ARITH>

doit être réécrite :

<TERME ARITH> ::= <FACTEUR ARITH> { <OPMUL> <FACTEUR ARITH> }

Les métasymboles "{" et "}" servent à indiquer que les éléments syntaxiques écrits entre accolades peuvent être répétés un nombre quelconque de fois compris 0. L'algorithme décrit utilise la forme équivalente suivante pour les expressions et instructions d'affectation.

<INST AFFEC> ::= <PARTIE GAUCHE> <FIN PARTIE GAUCHE> <EXPR ARITH>

<FIN PARTIE GAUCHE> ::= { <PARTIE GAUCHE> }

<PARTIE GAUCHE> ::= <IDENTIAFFEC> ::=

<IDENT AFFEC> ::= <VARIABLE> | <IDENTIFICATEUR DE PROCEDURE>

<EXPR ARITH> ::= si <EXPR BOOL> alors <EXPR ARITH SIMPLE> sinon <EXPR ARITH> | <EXPR ARITH SIMPLE>

```

<EXPR ARITH SIMPLE> ::= <PREMIER TERME><FIN TERME>
<PREMIER TERME> ::= <TERME> | <OPADD><TERME>
<FIN TERME> ::= { <OPADD><TERME> }
<TERME> ::= <FACTEUR><FIN FACTEUR>
<FIN FACTEUR> ::= { <OPMUL><FACTEUR> }
<FACTEUR> ::= <PRIMAIRE><FIN PRIMAIRE>
<FIN PRIMAIRE> ::= { + <PRIMAIRE> }
<PRIMAIRE> ::= <NOMBRE SANS SIGNE> | <VARIABLE SIMPLE> | <VARIABLE INDICEE> |
               <INDICATEUR DE FONCTION> | (<EXPR ARITH>)

```

On peut remarquer que l'on rencontre une nouvelle difficulté dans la reconnaissance d'un FIN PARTIE GAUCHE, étant donné que le délimiteur qui permet cette reconnaissance apparaît à la fin d'une chaîne dont la longueur est indéterminée dans le cas d'une variable indicée. Dans un but de simplicité, on a résolu le problème dans le cas présent par un balayage arrière bien que l'on puisse imaginer d'autres façons.

Le programme source est traduit en un programme objet constitué de macroinstructions. Les macroinstructions ont une ou deux adresses associées aux opérandes. Le résultat des opérations est toujours empilé. Une partie adresse égale à zéro signifie que l'opérande associé est un résultat intermédiaire. Les macroinstructions peuvent être repérées par des étiquettes numériques.

Exemple :

L'instruction d'affectation multiple :

$T[A] := B := T2[C, D] := E \times (\text{si } F \text{ alors } G+H \text{ sinon } I) + J$

sera traduite par la séquence :

	VAL	A
	INDADR	T, 1
	VAL	C
	VAL	D
	INDADR	T2, 2
	SCF	F, 1
	+	G, H
	SIC	2
1	NOP	
	VAL	I
2	NOP	
	×	E, 0
	+	O, J
	:= _M	O, 0
	:= _M	B, 0
	:=	O, 0

```

DEBUT COMMENTAIRE GENERATION-AFFECTATIONS-PROCEDURES SYNTAXIQUES ;
ENTIER I,K,R,SI,ALORS,SINON,PARGAUCHE,PARDROITE,DEUX PTS EGAL,
DEUX PTS EGALM,PLUS,MOINS,NEG,EXP,VAL,NOP,SCD,SIC,BLANC,ABSENTE,
MANQUANT,ETIQUETTE DISPONIBLE ;
ENTIER TABLEAU E[1:1000] ;
BOOLEEN PROCEDURE OPADD(X) ; ... ;
BOOLEEN PROCEDURE OPMUL(X) ; ... ;
BOOLEEN PROCEDURE OPEXP(X) ; ... ;
BOOLEEN PROCEDURE IDEN FONCT(X) ; ... ;
BOOLEEN PROCEDURE IDEN TAB(X) ; ... ;
BOOLEEN PROCEDURE VAR SIMPLE(X) ; ... ;
BOOLEEN PROCEDURE NBRE SS SIGNE(X) ; ... ;
PROCEDURE MACRO(OPERATEUR,OPERANDE1,OPERANDE2) ;
  DEBUT COMMENTAIRE GENERATION D UNE MACRO INSTRUCTION
  A 2 ADRESSES, UN RESULTAT INTERMEDIAIRE Y EST DENOTE PAR 0 ;
  ...
  FIN ;
PROCEDURE MACRO ETIQUETEE(ETIQUETTE,OPERATION,PARTIE ADRESSE) ;
  DEBUT ... FIN GENERATION D UNE MACRO INSTRUCTION ETIQUETEE
  ET NE COMPORTANT QU'UNE SEULE PARTIE ADRESSE ;

PROCEDURE PLACER EN TEMP(V) ; ENTIER V ;
  MACRO(VAL,V,BLANC) ;
  COMMENTAIRE GENERATION DE LA MACRO INSTRUCTION CORRESPONDANT
  A L'EMPILAGE DE LA VALEUR D'UNE VARIABLE SIMPLE EN MEMOIRE
  TEMPORAIRE ;

PROCEDURE EXPR BOOL(EB RESULT) ;
  DEBUT COMMENTAIRE CETTE PROCEDURE ANALYSE UNE EXPRESSION
  BOOLEENNE ET GENERE LE CODE CORRESPONDANT.
  L'ADRESSE DE LA VALEUR BOOLEENNE RESULTANTE EST
  PLACEE DANS EB RESULT.
  CETTE PROCEDURE EST ANALOGUE A EXPR ARITH ;
  ...
  FIN ;

PROCEDURE EXPR ARITH(EA RESULT) ; ENTIER EA RESULT ;
  DEBUT
  SI E[I]=SI ALORS
  EXPRESSION CONDITIONNELLE:
  DEBUT ENTIER EAX,EB,EAS,ETIQUETTE1,ETIQUETTE2 ;
  COMMENTAIRE LES ETIQUETTES SONT NUMERIQUES ;
  ETIQUETTE1:=ETIQUETTE DISPONIBLE ;
  ETIQUETTE2:=ETIQUETTE1+1 ;
  ETIQUETTE DISPONIBLE:=ETIQUETTE DISPONIBLE+2 ;
  I:=I+1 ;
  EXPR BOOL(EB) ;
  SI E[I] # ALORS OU EB=MANQUANT ALORS ERREUR(1) ;
  I:=I+1 ;
  MACRO(SCD,EB,ETIQUETTE1) ;
  EXPR ARITH SIMPLE(EAS) ;
  SI E[I] # SINON OU EAS=MANQUANT ALORS

```

```

ERREUR(2) ;
SI EAS # 0 ALORS PLACER EN TEMP(EAS) ;
I:=I+1 ;
MACRO(SIC, ETIQUETTE2, BLANC) ;
MACRO ETIQUETEE(ETIQUETTE1, NOP, BLANC) ;
EXPR ARITH(EAX) ;
SI EAX=MANQUANT ALORS ERREUR(3) ;
SI EAX # 0 ALORS PLACER EN TEMP(EAX) ;
MACRO ETIQUETEE(ETIQUETTE2, NOP, BLANC) ;
EA RESULT:=0 ;
ALLERA SORTIE
FIN LA VALEUR D UNE EXPRESSION CONDITIONNELLE EST TOUJOURS
EN MEMOIRE TEMPORAIRE ;
EXPR ARITH SIMPLE(EA RESULT) ;
SORTIE:
FIN DE EXPR ARITH ;

```

```

PROCEDURE EXPR ARITH SIMPLE(EAS RESULT) ;
  ENTIER EAS RESULT ;
  DEBUT ENTIER TERME1 ;
    SI E[I]=MOINS ALORS
      DEBUT I:=I+1 ;
      TERME ARITH(TERME1) ;
      MACRO(NEG, TERME1, BLANC) ;
      SI TERME1=MANQUANT ALORS ERREUR(4) SINON
        TERME1:=0 ;
      FIN
    SINON DEBUT
      SI E[I]=PLUS ALORS I:=I+1 ;
      TERME ARITH(TERME1) ;
      SI TERME1=MANQUANT ALORS ERREUR(5) ;
      FIN ;
  RESTE TERMES(TERME1, EAS RESULT)
  FIN EXPR ARITH SIMPLE ;

```

```

PROCEDURE RESTE TERMES(EAS1, EASX RESULT) ;
  VALEUR EAS1 ; ENTIER EAS1, EASX RESULT ;
  DEBUT EASX RESULT:=EAS1 ;
    SI OP ADD(E[I]) ALORS
      DEBUT ENTIER OPERATEUR, TERME2 ;
      OPERATEUR:=E[I] ; I:=I+1 ;
      TERME ARITH(TERME2) ;
      SI TERME2=MANQUANT ALORS ERREUR(6) ;
      MACRO(OPERATEUR, EAS1, TERME2) ;
      RESTE TERMES(0, EASX RESULT)
    FIN
  FIN RESTE TERMES ;

```

```

PROCEDURE TERME ARITH(TERM RESULT) ;
  ENTIER TERM RESULT ;
  DEBUT ENTIER FACTEUR1 ;
    FACTEUR ARITH(FACTEUR1) ;

```

```

    SI FACTEUR1=MANQUANT ALORS ERREUR(7) ;
    RESTE FACTEURS(FACTEUR1, TERM RESULT)
  FIN TERME ARITH ;

```

```

PROCEDURE RESTE FACTEURS(TERM1, TERMX RESULT) ;
  VALEUR TERM1 ; ENTIER TERM1, TERMX RESULT ;
  DEBUT TERMX RESULT:=TERM1 ;
  SI OPMUL(E[I]) ALORS
    DEBUT ENTIER OPERATEUR, FACTEUR2 ;
    OPERATEUR:=E[I] ; I:=I+1 ;
    FACTEUR ARITH(FACTEUR2) ;
    SI FACTEUR2=MANQUANT ALORS ERREUR(8) ;
    MACRO(OPERATEUR, TERM1, FACTEUR2) ;
    RESTE FACTEURS(0, TERMX RESULT)
  FIN
FIN RESTE FACTEURS ;

```

```

PROCEDURE FACTEUR ARITH(FACT RESULT) ;
  ENTIER FACT RESULT ;
  DEBUT ENTIER PRIMAIRE1 ;
  PRIMAIRE(PRIMAIRE1) ;
  SI PRIMAIRE1=MANQUANT ALORS ERREUR(9) ;
  RESTE PRIMAIRES(PRIMAIRE1, FACT RESULT)
  FIN FACTEUR ARITH ;

```

```

PROCEDURE RESTE PRIMAIRES(FACTEUR1, FACTX RESULT) ;
  VALEUR FACTEUR1 ; ENTIER FACTEUR1, FACTX RESULT ;
  DEBUT FACTX RESULT:=FACTEUR1 ;
  SI OPEXP(E[I]) ALORS
    DEBUT ENTIER PRIMAIRE2 ; I:=I+1 ;
    PRIMAIRE(PRIMAIRE2) ;
    SI PRIMAIRE2=MANQUANT ALORS ERREUR(10) ;
    MACRO(EXP, FACTEUR1, FACTEUR2) ;
    RESTE PRIMAIRES(0, FACTX RESULT)
  FIN
FIN RESTE PRIMAIRES ;

```

```

PROCEDURE PRIMAIRE(PRIM RESULT) ; ENTIER PRIM RESULT ;
  DEBUT
    SI NBRE SS SIGNE(E[I]) OU VAR SIMPLE(E[I]) ALORS
      DEBUT PRIM RESULT:=E[I] ; ALLERA EXIT FIN ;
    SI IDEN TAB(E[I]) ALORS
      DEBUT ... ; ALLERA EXIT FIN ;
    SI IDEN FONCT(E[I]) ALORS
      DEBUT ... ; ALLERA EXIT FIN ;
    SI E[I]=PARGAUCHE ALORS
      DEBUT I:=I+1 ;
      EXPR ARITH(PRIM RESULT) ;
      SI E[I] ≠ PARDROITE ALORS ERREUR(11)
      SINON
        ALLERA EXIT
      FIN ;
  FIN ;

```

```

    PRIM RESULT:=MANQUANT ; ALLERA EXIT2 ;
    EXIT: I:=I+1 ; EXIT2:
    FIN PRIMAIRE ;

```

```

PROCEDURE AFFECTATION ;
    DEBUT ADRESSE1,EA,II, KK ;

```

```

PROCEDURE PRENDRE ADRESSE(ADRESSE) ; ENTIER ADRESSE ;
    DEBUT
    SI VAR SIMPLE(E[I]) ALORS
        DEBUT ADRESSE:=E[I] ; ALLERA SORTIE FIN ;
    SI IDEN TAB(E[I]) ALORS
        DEBUT ... ; ALLERA SORTIE FIN COMPILATION DU CALCUL
        DE L ADRESSE DE LA VARIABLE INDICEE.
        ADRESSE PREND LA VALEUR 0 PUISQUE L ADRESSE DE LA VARIABLE
        SERA EMPILEE ;
    SI IDEN FONCT(E[I]) ALORS
        DEBUT ... ; ALLERA SORTIE FIN L ADRESSE A ETE OBTENUE
        COMME POUR UNE VARIABLE SIMPLE ;
    ADRESSE:=ABSENTE ;
    SORTIE:
    FIN PRENDRE ADRESSE ;

```

```

PROCEDURE SUITE AFFEC ET EXPR ;
    DEBUT ENTIER ADRESSE2 ;
    SE SOUVENIR: II:=I ; KK:=K ;
    PRENDRE ADRESSE(ADRESSE2) ;
    SI E[I+1]=DEUX PTS EGAL ET
        ADRESSE2 ≠ ABSENTE ALORS
        DEBUT I:=I+2 ;
        SUITE AFFEC ET EXPR ;
        MACRO(DEUX PTS EGALM,EA) ;
        ALLERA FIN
        FIN ;
    RESTAURER: I:=II ; K:=KK ;
    EXPR ARITH(EA) ;
    FIN:
    FIN SUITE AFFEC ET EXPR ;
    INSTRUCTION D AFFECTATION:
    PRENDRE ADRESSE(ADRESSE1) ;
    SI E[I+1] ≠ DEUX PTS EGAL OU ADRESSE1=ABSENTE
        ALORS ERREUR(0) ;
    I:=I+2 ;
    SUITE AFFEC ET EXPR ;
    MACRO(DEUX PTS EGAL,ADRESSE1,EA)
    FIN D AFFECTATION ;

```

```

    o o o o o o o o o o o o o o o o
    PROGRAMME:

```

```

    o o o o o

```

INSTRUCTION D AFFECTATION: AFFECTATION ;

o o o o

FIN

3. Méthode de précedence avec vérification syntaxique complète.

L'analyseur syntaxique que nous allons décrire est bien adapté à la macro-génération pour les raisons suivantes :

- il n'y a pas de balayage arrière.
- il saute certaines étapes intermédiaires inutiles pour la macro-génération.
- il détecte toutes les erreurs syntaxiques.

Cet algorithme est à la fois efficace et rapide mais ne peut être utilisé que pour les grammaires de précedence.

Pour toute grammaire où il n'existe pas de règles contenant deux symboles non terminaux adjacents, Floyd [10] a défini trois relations binaires entre symboles terminaux.

($\langle . , \hat{=} , . \rangle$) définies de la façon suivante :

- | | | |
|-----------------|---|---|
| $a \langle . b$ | } | <p>Il existe</p> <ul style="list-style-type: none"> - une règle telle que : $X \rightarrow \dots a U \dots$ - une dérivation telle que : $U \rightarrow x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_n$ <p>avec $n > 1$ et b symbole terminal le plus à gauche de x_n</p> |
| $a \hat{=} b$ | } | <p>Il existe :</p> <ul style="list-style-type: none"> - une règle telle que $x \rightarrow \dots ab \dots$
ou $x \rightarrow \dots a U b \dots$ avec U non terminal. |
| $a . \rangle b$ | } | <p>Il existe :</p> <ul style="list-style-type: none"> - une règle telle que $x \rightarrow \dots U b \dots$ - une dérivation telle que $U \rightarrow x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_n$ <p>avec $n > 1$ et a symbole terminal le plus à droite de x_n.</p> |

Une grammaire est de précedence si pour chaque couple ordonné (a, b) symboles terminaux une seule des trois relations précédentes est vérifiée.

Floyd [10] a décrit un algorithme très simple pour l'analyse des langages finis par des grammaires de précedence mais qui laisse passer un certain nombre d'erreurs. L'algorithme ci-après mis au point par A. Colmerauer, [39], permet la détection de toutes les erreurs grâce à l'utilisation combinée des relations de précedence et des règles de description syntaxique.

Le programme utilise les conventions suivantes :

- les symboles terminaux sont numérotés de 1 à TMAX et les symboles non terminaux de TMAX + 1 à SYMAX.
- on ajoute la règle $S' \rightarrow \# S \#$ (le programme à analyser doit donc être écrit entre ces deux symboles).
- STOP et AXIOME représentent les valeurs codées des symboles $\#$ et S.
- les règles syntaxiques sont rangées dans le tableau REGLE de la façon suivante :

Les règles : $S \rightarrow a$, $S \rightarrow a S$ seront représentées comme

S	1	a	S	2	a	S	0	...	
---	---	---	---	---	---	---	---	-----	--

↑
REGLE MAX

Les relations de précédence sont représentées dans les 2 tableaux booléens PE et PS :

si T_i et T_j sont des symboles terminaux

$PE[i, j]$ a la valeur vrai si $T_i \doteq T_j$.

$PS[i, j]$ a la valeur vrai si $T_i \dot{>} T_j$.

Les possibilités de dérivation sont représentées dans le tableau booléen REMPLACI

$REPLACE[i, j]$ a la valeur vrai si $N_i = N_j$

ou si $N_i \rightarrow N_{i_1} \rightarrow N_{i_2} \rightarrow \dots \rightarrow N_j$.

N_i et N_j étant des symboles non terminaux.

DEBUT COMMENTAIRE GENERATION=METHODO DE PRECEDENCE ;
ENTIER TMAX, SYMAX, FILEMAX, REGLEMAX, STOP, AXIOME ;

.....

DEBUT

ENTIER I, J, K, L, M, Q, LONGUEUR, NBSUJETS ;
ENTIER TABLEAU PILE[1:FILEMAX], REGLE[1:REGLEMAX] ;
BOOLEEN TABLEAU PE, PS[1:TMAX, 1:TMAX],
REPLACE[TMAX+1:SYMAX, TMAX+1:SYMAX] ;
BOOLEEN PROCEDURE TERMINAL(S) ; ENTIER S ;
TERMINAL:=S < TMAX ;
ENTIER PROCEDURE SYMBOLE SUIVANT ;
DEBUT FIN ;
PROCEDURE ERREUR ; ;
PROCEDURE MACRO GENERATION ; ;

.....
 INITIALISATION:

.....
 I:=J:=1 ;
 PILE[1]:=SYMBOLE SUIVANT ;
 PILE[FILEMAX]:=SYMBOLE SUIVANT ;

DELIMITATION A DROITE D UNE PHRASE PRIMAIRE:

SI NON PS[PILE[J], PILE[FILEMAX]] ALORS

DEBUT

SI PILE[FILEMAX]=STOP ALORS ALLERA FIN ;

J:=I:=I+1 ;

PILE[I]:=PILE[FILEMAX] ;

PILE[FILEMAX]:=SYMBOLE SUIVANT ;

ALLERA DELIMITATION A DROITE D UNE PHRASE PRIMAIRE ;

FIN ;

LONGUEUR:= SI I=J ALORS 0 SINON I ;

DELIMITATION A GAUCHE D UNE PHRASE PRIMAIRE:

Q:=PILE[J] ;

J:=J-1 ;

SI NON TERMINAL(PILE[J]) ALORS

DEBUT

LONGUEUR:=LONGUEUR+1 ;

L1: J:=J-1 ;

SI NON TERMINAL(PILE[J]) ALORS ALLERA L1

FIN ;

SI PE[PILE[J], Q] ALORS

ALLERA DELIMITATION A GAUCHE D UNE PHRASE PRIMAIRE ;

TROUVER LES SUJETS:

NBSUJETS:=0 ;

POUR K:=2, K+REGLE[K]+2 TANTQUE REGLE[K-1] ≠ 0 FAIRE

SI LONGUEUR=REGLE[K] ALORS

DEBUT

L:=1 ;

POUR M:=K+LONGUEUR PAS -1 JUSQUA K+1 FAIRE

DEBUT

```

SI TERMINAL[REGLE[M]] ALORS
  ALLERA SI REGLE[M]=PILE[L] ALORS
    L4 SINON REGLE SUIVANTE ;
  SI TERMINAL[PILE[L]] ALORS ALLERA
  REGLE SUIVANTE ;
L2: SI REPLACE[REGLE[M],PILE[L]] ALORS
  ALLERA L3 ;
  L:=L-1 ;
  ALLERA SI TERMINAL[PILE[L]] ALORS
  REGLE SUIVANTE SINON L2 ;
L3: L:=L-1 ;
  ALLERA SI TERMINAL[PILE[L]] ALORS
  L5 SINON L3 ;
L4: L:=L-1 ;
L5:
FIN ;
NBSUJETS:=NBSUJETS+1 ;
PILE[PILEMAX-NBSUJETS]:=REGLE[K-1] ;
REGLE SUIVANTE:
FIN ;

```

REDUCTION:

```

SI NBSUJETS=0 ALORS DEBUT ERREUR ; ALLERA L6 FIN ;
MACROGENERATION ;
POUR K:=1 PAS 1 JUSQUA NBSUJETS FAIRE
  PILE[J+K]:=PILE[PILEMAX-K] ;
  I:=J+NBSUJETS ;
ALLERA DELIMITATION A DROITE D UNE PHRASE PRIMAIRE ;
FIN:
  POUR K:=2 PAS 1 JUSQUA 1 FAIRE
    SI REPLACE[AXIOME,PILE[K]] ALORS ALLERA L6 ;
  ERREUR ;
L6:

```

FIN
FIN

4. Utilisation des macroinstructions dans la génération.

Pour montrer l'intérêt de l'utilisation des macroinstructions dans la génération des programmes, considérons le cas des expressions arithmétiques.

Le résultat de la compilation d'une expression peut être donné sous forme d'une liste de macroinstructions à deux paramètres.

Par exemple l'expression :

$$(A + B) / (E \times (C + D) - F)$$

sera traduite par la liste :

1	FADD	A,B
2	FADD	C,D
3	FMUL	E,O
4	FSOU	O,F
5	FDIV	O,O

(les mémoires intermédiaires utilisées pour l'évaluation sont optimisées par l'utilisation d'une pile, les adresses O indiquant que l'opérande est au sommet de cette pile de mémoires intermédiaires).

Cette liste de macros pourrait s'interpréter directement au moyen de sous-programmes correspondant aux diverses macro-opérations mais voulant tirer profit de la machine sur laquelle sera évaluée cette expression, nous définissons un ensemble de macros qui permettra de générer une séquence d'instructions optimales pour cette machine.

Nous pouvons décrire ces macros comme des procédures ALGOL de la façon suivante :

procedure FADD (A, B) ; chaîne A, B ;

début commentaire OP1 et OP2 sont des quantités booléennes définissant la nature des opérandes de l'opération FADD, EGAL (U, V) est une procédure booléenne prenant la valeur vrai si U et V sont du même type et la valeur faux dans le cas contraire.

Le type est une valeur booléenne égale à :

vrai si l'opérande est symbolique,

faux si l'opérande est une mémoire intermédiaire ;

booleen OP1, OP2 ;

OP1 := EGAL (A, vrai) ;

OP2 := EGAL (B, vrai) ;

si OP1 \wedge OP2 alors debut

FADD1 (A, B) ;

aller a F ;

fin ;

si OP1 alors debut ;

FADD 23 (A) ;

aller a F ;

fin ;

si OP2 alors debut ;

FADD 23 (B) ;

aller a F ;

fin ;

FADD 4 ;

F : RESAC

fin ;

procedure FADD1 (A,B) ; chaîne A, B ;

debut commentaire cette procédure génère le programme correspondant à l'addition de 2 opérandes symboliques, CLA et FAD sont des procédures en code ;

RANGER ;

CLA (A) ;

FAD (B)

fin ;

procedure FADD 23 (A) ; chaîne (A) ;

debut commentaire cas où l'un des deux opérandes est défini par une mémoire intermédiaire, MQ est une quantité booléenne indiquant si le registre MQ est utilisé ou non ;

si MQ alors MQAC ;

FAD (A)

fin ;

procedure FADD 4 ;

debut commentaire cas où les deux opérandes sont définis par des mémoires intermédiaires ;

FADD 23 ('PILE + INDEX -1') ;

SET (INDEX, INDEX -1)

fin ;

procédure RANGER

debut commentaire cette procédure permet le rangement du contenu des registres AC et MQ ;

si AC alors RANGAC ;

si MQ alors RANGMQ

fin ;

procedure MQAC ;

debut commentaire permutation des registres AC et MQ ;

STQ (PILE + INDEX) ;

CLA (PILE + INDEX)

fin ;

procédure RESAC ;

debut

AC := vrai ;

MQ := faux

fin ;

procedure RANGAC ;

debut

STO (TABLE + INDEX) ;

SET (INDEX, INDEX + 1)

fin ;

procedure RANGMQ ;

debut

STQ (TABLE + INDEX) ;

SET (INDEX, INDEX + 1)

fin ;

Si nous disposons d'un macro-assembleur capable de faire de l'assemblage conditionnel, il est facile de transposer ces procédures en langage de macro-assembleur.

On trouvera ci-après leur transposition en langage MAP 7044.

FADD	MACRO	A,B	
OP1	SET	0	
OP2	SET	0	
	IFT	A = 0	
OP1	SET	4	PREMIER OPERANDE EN PILE
	IFT	B = 0	
OP2	SET	8	SECOND OPERANDE EN PILE
	IFT	OP1 + OP2 = 0	
	FADD1	A,B	AUCUN OPERANDE EN PILE
	IFT	OP1 + OP2 = 4	
	FADD23	B	PREMIER OPERANDE EN PILE
	IFT	OP1 + OP2 = 8	
	FADD23	A	SECOND OPERANDE EN PILE
	IFT	OP1 + OP2 = 12	
	FADD4		LES DEUX OPERANDES EN PILE

	RESAC		
	ENDM	FADD	
FADD1	MACRO	A,B	
	RANGER		RANGE, S'IL Y A LIEU, LE RESULTAT PRECEDENT
	CLA	A	
	FAD	B	
	ENDM	FADD1	
FADD23	MACRO	A	
	IFT	MQ = 2	SI L'OPERANDE EST EN MQ, L'AMENER EN AC
	MQAC		
	FAD	A	
	ENDM	FADD23	
FADD4	MACRO		
FADD23	TABLE + INDEX - 1		
INDEX	SET	INDEX -1	REAJUSTER LE NIVEAU DE LA PILE
	ENDM	FADD4	
RANGER	MACRO		
	IFT	AC = 1	
	RANGAC		
	IFT	MQ = 2	
	RANGMQ		
	ENDM	RANGER	
MQAC	MACRO		
	STQ	TABLE + INDEX	
	CLA	TABLE + INDEX	
	ENDM	MQAC	

ACMQ	MACRO	
	STO	TABLE + INDEX
	LDQ	TABLE + INDEX
	ENDM	ACMQ
RESAC	MACRO	
AC	SET	1
MQ	SET	0
	ENDM	RESAC

On remarquera que pour le calculateur 7044 la séquence d'instructions machine produite est réellement optimisée du point de vue utilisation des mémoires de travail et en particulier les registres AC et MQ.

(Pour la permutation des registres AC et MQ il est plus rapide de transférer le registre AC dans une mémoire et de rappeler le contenu de cette mémoire dans le registre MQ que d'opérer par décalages).

L'expression donnée précédemment sous la forme d'une liste de 5 macros fournira le programme :

CLA	A
FAD	B
STO	TABLE + INDEX - 2
CLA	C
FAD	D
STO	TABLE + INDEX - 1
LDQ	TABLE + INDEX - 1
FMP	E
FSB	F
STO	TABLE + INDEX - 1
CLA	TABLE + INDEX - 2
FDP	TABLE + INDEX - 1

tandis que l'expression :

$$((A \times B + C) / D \times E - F) / G$$

donnerait :

LDQ	A
FMP	B
FAD	C
FDP	D
EMP	E
FSB	F
FDP	G

Cette utilisation des macros peut être étendue à tout le langage et permet ainsi de séparer nettement l'utilisation des particularités machine pour le programme objet de la compilation proprement dite.

Voici à titre d'exemple les trois macros qui permettent la génération des expressions conditionnelles.

SCD	MACRO	A
	TZE	A. 'A
AC	SET	O
	ENDM	SCD
SIC	MACRO	A,B
	TRA	A. 'B
A. 'A	EQU	*
AC	SET	O
	ENDM	SIC
FEC	MACRO	A
A. 'A	EQU	*
	ENDM	FEC

Etant donné l'expression conditionnelle :

$D \times E + (A + (\text{si } AA \vee BB \wedge CC \text{ alors } (\text{si } AA \wedge CC \text{ alors } E + F$
 $\text{sinon } G + H) \text{ sinon si } AA \wedge BB \text{ alors } I + J \text{ sinon } I - J)) \times B + C$

nous obtenons :

<u>Liste des macros</u>			<u>Programme MAP généré</u>	
1	FMUL	D,E	LDQ	D
2	ET	BB,CC	FMP	E
3	OU	AA,0	STO	TABLE + INDEX - 1
4	SCD	11	CAL	BB
5	ET	AA,CC	ANA	CC
6	SCD	8	ORA	AA
7	FADD	E,F	TZE	A.11
8	SIC	8,10	CAL	AA
9	FADD	G,H	ANA	CC
10	FEC	10	TZE	A.8
11	SIC	11,18	CLA	E
12	ET	AA,BB	FAD	F
13	SCD	15	TRA	A.10
14	FADD	I,J	A.8 EQU	*
15	SIC	15,17	CLA	G
16	FSOU	I,J	FAD	H
17	FEC	17	A.10 EQU	*
18	FEC	18	TRA	A.18
19	FADD	A,0	A.11 EQU	*
20	FMUL	0,B	CAL	AA
21	FADD	0,0	ANA	BB
22	FADD	0,C	TZE	A.15

	CLA	I
	FAD	J
	TRA	A.17
A.15	EQU	+
	CLA	I
	FSB	J
A.17	EQU	*
A.18	EQU	*
	FAD	A
	STO	TABLE + INDEX
	LDQ	TABLE + INDEX
	EMP	B
	FAD	TABLE + INDEX - 1
	FAD	C

III - COMPILATEURS MODULAIRES ET PARAMETRIQUES.

Un système complet pour la mise en oeuvre d'un langage présente plusieurs parties bien distinctes :

- macroévaluateur
- analyseur syntaxique
- générateur
- interpréteur

Chacune de ces parties peut elle-même se décomposer en plusieurs parties : par exemple, l'analyse syntaxique peut consister en une phase d'analyse lexicographique suivie d'une phase d'analyse syntaxique proprement dite au niveau des unités syntaxiques résultant de la phase précédente.

Cet aspect modulaire est très important pour l'écriture et la maintenance de tels systèmes étant donné la complexité et la diversité des transformations nécessaires pour passer de la lecture d'un texte source à l'évaluation du programme objet équivalent.

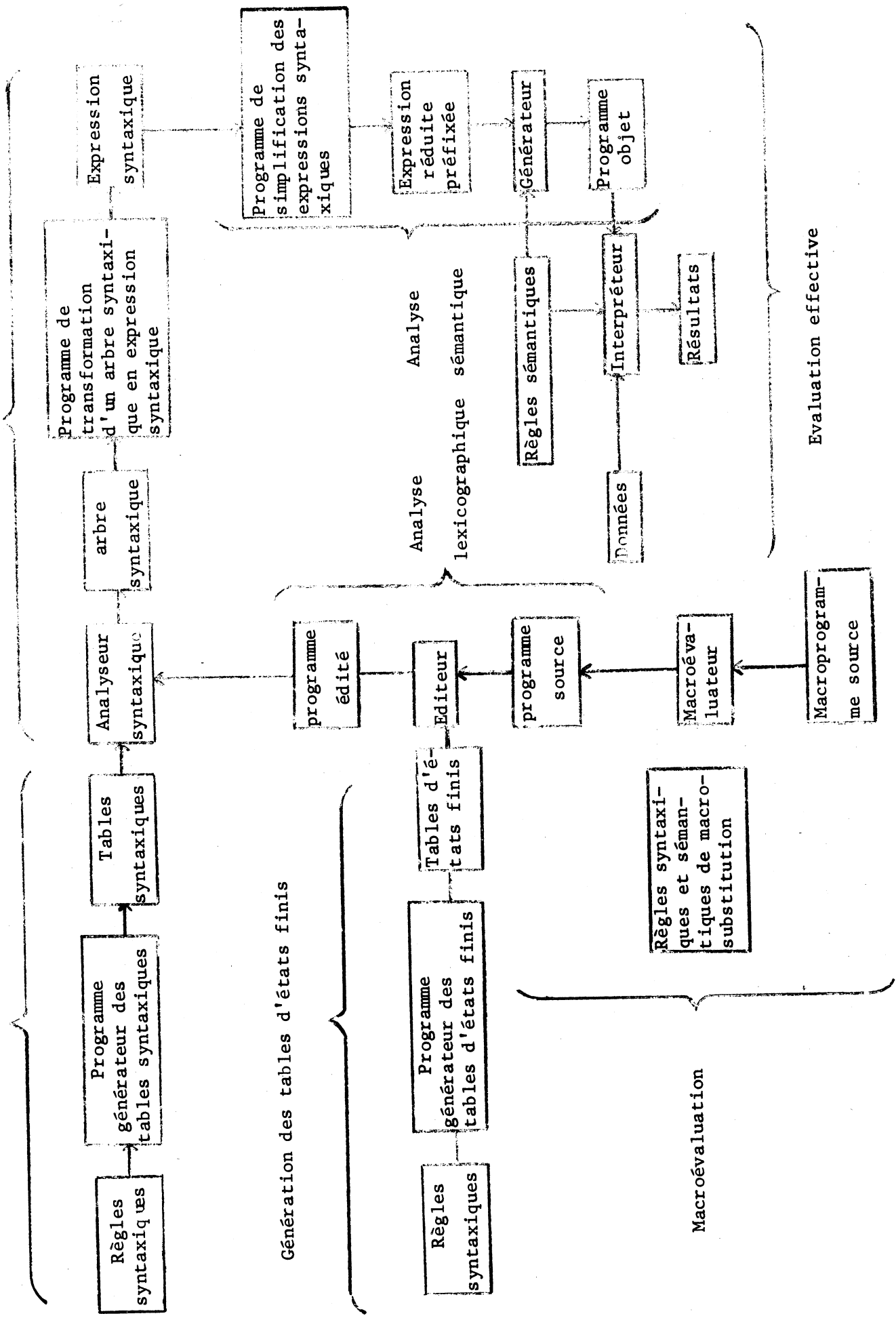
L'aspect paramétrique est de nature bien différente :
alors que l'aspect modulaire vise à rendre plus facile la mise en oeuvre d'un langage ou de sous-ensembles de ce langage sur une série de calculateurs, l'aspect paramétrique vise à rendre possible sur un même calculateur la mise en oeuvre d'une série de langages caractérisés par des similarités de description tant syntaxique que sémantique.

Le diagramme suivant décrit un système expérimental en cours de réalisation.

Nous donnons également quelques détails sur la production de la forme réduite préfixée et un mécanisme de génération des macros.

Analyse syntaxique

Génération des tables syntaxiques



Génération des tables d'états finis

Macroévaluation

Evaluation effective

Forme réduite préfixée.

Considérons les règles suivantes définissant une instruction d'affectation très simple :

$\langle A \rangle ::= v e \langle E \rangle$
 $\langle E \rangle ::= \langle T \rangle \mid \langle E \rangle a \langle T \rangle$
 $\langle T \rangle ::= \langle F \rangle \mid \langle T \rangle m \langle F \rangle$
 $\langle F \rangle ::= n | v | g \langle E \rangle d$

Les métavariabes ont la signification suivante :

$\langle A \rangle$ affectation, $\langle E \rangle$ expression, $\langle T \rangle$ terme, $\langle F \rangle$ facteur.

Les symboles terminaux :

v (variable), e (deux points égal), a (op. additif), m (op. multiplic.), n (nombre), g (parenthèse gauche), d (parenthèse droite).

L'arbre de décomposition généré par l'analyse syntaxique pour l'instruction d'affectation suivante est représenté ainsi :

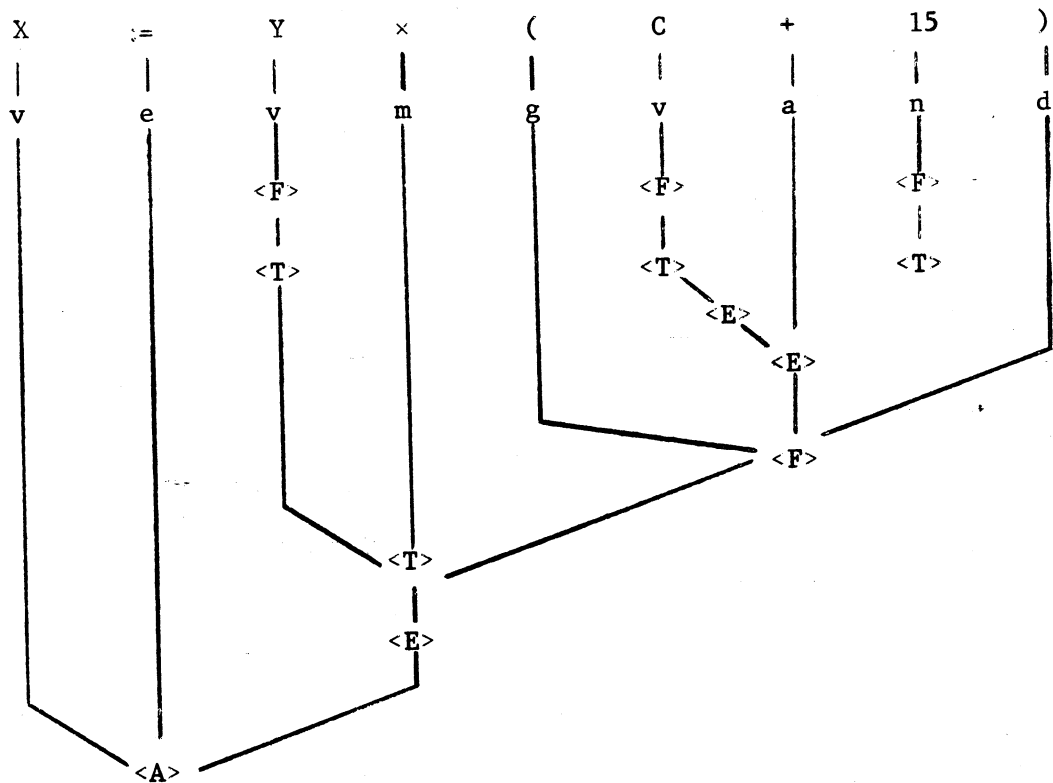
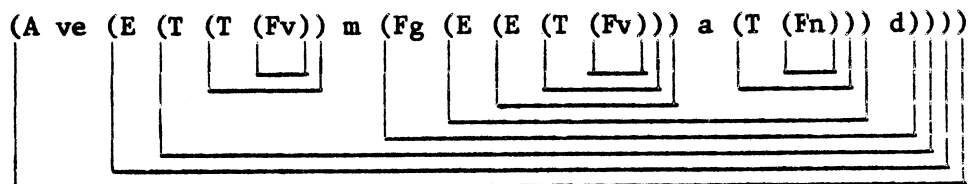


Figure 9

Il est facile de faire apparaître le résultat de l'analyse syntaxique sous la forme de liste suivante (structure de liste) :



Ayant le résultat sous cette forme, il faut maintenant éliminer des symboles non terminaux redondants pour la génération, à savoir tous les symboles non terminaux qui ne constituent pas un noeud de branchement dans l'arbre de décomposition et réécrire la liste de telle façon que les opérateurs préfixent les opérands correspondants.

Pour l'exemple ci-dessus on veut obtenir :

(e v (m v (a v n)))

qui représente l'expression initiale sous forme préfixée parenthésée.

Nous décrivons ci-après la procédure **FORME REDUITE PREFIXEE** qui réalise cette transformation.

Macro-génération.

A partir de cette forme on génère une liste de macro-instructions de forme :

```
ADD      C, 15
MUL      Y, 0
AFF      X, 0
```

représentant le programme objet correspondant qui peut être directement interprété ou envoyé vers un macro-assembleur pour produire du langage machine.

Nous décrivons également la procédure **MACRO-GENERATION** qui fournit cette liste de macros.

```

ENTIER PROCEDURE FORME PREFIXEE REDUITE(L) ; VALEUR L ;
  ENTIER L ;
DEBUT COMMENTAIRE CETTE PROCEDURE TRANSFORME UN ARBRE SYNTAXIQUE
  ECRIT SOUS FORME DE LISTE EN UNE EXPRESSION PREFIXEE D OU
  SONT ELIMINES TOUS LES SYMBOLES REDONDANTS ;
ENTIER PROCEDURE LISTE3(S1,S2,S3) ;
  VALEUR S1,S2,S3 ; ENTIER S1,S2,S3 ;
  LISTE3:=CONS(S1,CONS(S2,CONS(S3,1))) ;
AIGUILLAGE NON TERMINAL:=A,E,T,F ;
ALLERA NON TERMINAL[ETIQUETTE(CADAR(L))] ;

```

```

A:
  FORME PREFIXEE REDUITE:=
    LISTE3(CADDR(L),CADR(L),FORME PREFIXEE REDUITE(CADDR(L))) ;
  ALLERA FIN ;

```

```

E: T:
  FORME PREFIXEE REDUITE:=
    SI CDDR(L)=1 ALORS FORME PREFIXEE REDUITE(CADR(L))
    SINON LISTE3(CADDR(L),FORME PREFIXEE REDUITE(CADR(L)),
      FORME PREFIXEE REDUITE(CADDR(L))) ;
  ALLERA FIN ;

```

```

F:
  FORME PREFIXEE REDUITE:=
    SI CDDR(L)=1 ALORS CADR(L)
    SINON FORME PREFIXEE REDUITE(CADDR(L)) ;

```

```

FIN:
FIN FORME PREFIXEE REDUITE ;

```



```

ENTIER PROCEDURE MACRO GENERATION(L) ;
VALEUR L ; ENTIER L ;
DEBUT COMMENTAIRE GENERATION DES MACRO INSTRUCTIONS A
    PARTIR DE LA FORME PREFIXEE REDUITE ;
BOOLEEN B1,B2 ; ENTIER T0,T1,T2 ;
    ;
    PROCEDURE MACRO(U,V,W) ;
    VALEUR U,V,W ; ENTIER U,V,W ;
    DEBUT COMMENTAIRE EDITION DE MACRO ;
    SAUTLIGNE ;
    ECRIRE ATOME(AUX[EQT[TCAR[TCADR[U]]]]) ;
    ECRIRE CHAINE( " " ) ;
    SI V=0 ALORS ECRIRE CHAINE( " , " )
        SINON ECRIRE ATOME(V) ;
    ECRIRE CHAINE( " , " ) ;
    SI W=0 ALORS ECRIRE CHAINE( " 0 " )
        SINON ECRIRE ATOME(W) ;
    SAUTLIGNE
    FIN MACRO ;

T0:=CAR(L) ; T1:=CADR(L) ; T2:=CADDR(L) ;
B1:=ATOME(T1) ; B2:=ATOME(T2) ;
SI B1 ET B2 ALORS MACRO(T0,T1,T2)
    SINON
        SI B1 ALORS
            DEBUT MACRO GENERATION(T2) ;
            MACRO(T0,T1,0)
            FIN
        SINON
            SI B2 ALORS
                DEBUT MACRO GENERATION(T1) ;
                MACRO(T0,0,T2)
                FIN
            SINON DEBUT MACRO GENERATION(T1) ;
                MACRO GENERATION(T2) ;
                MACRO(T0,0,0)
                FIN
    FIN MACRO GENERATION ;

```

COMPILATEURS CONVERSATIONNELS ET INCREMENTIELS.

C H A P I T R E I

COMPILATEURS POUR SYSTEMES A ACCES MULTIPLE

L'utilisation des compilateurs conversationnels n'est pas exclusivement limitée aux systèmes à accès multiple. Sur de petits calculateurs de tels systèmes ont été mis au point et utilisés avec succès. Toutefois les langages utilisés étaient généralement très simples et la réalisation ne posait pas de problèmes vraiment très difficiles.

Le développement des systèmes à accès multiple nécessite la mise en oeuvre et l'utilisation de compilateurs dans un environnement très différent de celui des systèmes moniteurs classiques fonctionnant en mode séquentiel.

I - COMPILATEURS NON CONVERSATIONNELS.

Quelques systèmes à accès multiple n'utilisent pas de compilateurs conversationnels c'est-à-dire qu'il n'y a pas d'interaction directe entre le programmeur et le programme en cours de compilation.

Au niveau du programmeur, les seules différences avec le mode d'exploitation classique résident dans la disponibilité d'un terminal et éventuellement une certaine priorité par rapport aux travaux faisant partie d'un train moniteur.

Dans la première version du système DIAMAG [40], le programmeur demande l'exécution des programmes à partir d'un terminal en fournissant les mêmes informations que dans l'exploitation séquentielle normale (cartes de contrôle, programmes et données). Le mécanisme de priorité de ce système fonctionne de la façon suivante :

A la fin de chaque travail exécuté en mode moniteur normal, une interruption se produit pour tester si un travail issu d'un terminal est prêt à être exécuté. Dans l'affirmative ce travail a priorité sur le travail suivant du train moniteur et est exécuté immédiatement. Des priorités entre terminaux et entre directions de transmission sur un même terminal (dans le cas d'une transmission simple) peuvent être définies de façon analogue.

Dans les systèmes fonctionnant purement en mode séquentiel, les compilateurs restent du type usuel en ce sens que les compilateurs traitent les programmes source l'un après l'autre jusqu'à ce que chacun d'eux soit terminé ou qu'une erreur soit détectée et arrête la compilation.

Dans les systèmes à multiprogrammation, plusieurs compilations peuvent être réalisées simultanément (ou alternativement à tour de rôle pendant de courts intervalles de temps).

Deux solutions sont alors possibles :

a) il y a autant de copies du compilateur que de programmes en cours de compilation ; le fonctionnement du compilateur ne diffère pas alors de l'exploitation en mode séquentiel.

b) il y a une seule copie du compilateur utilisable pour tous les programmes en cours de compilation ; dans ce cas le compilateur doit être réentrant c'est-à-dire utilisable par plusieurs programmes source en même temps.

Un tel compilateur doit avoir les propriétés suivantes :

- la partie commune ne doit pas se modifier elle-même pendant l'exécution.

- la zone de travail et les parties variables du compilateur (tables, listes) doivent être considérées comme externes au compilateur et attachées en tant que données à chacun des programmes source en cours de compilation.

Chaque fois qu'une interruption se produit au cours d'une compilation, l'état du compilateur correspondant au programme source intéressé est rangé dans la zone de travail relative à ce programme source de manière à permettre la reprise de la compilation un peu plus tard dans des conditions normales.

II - COMPILATEURS CONVERSATIONNELS.

Il existe une grande variété de systèmes conversationnels allant des analyseurs syntaxiques conversationnels aux systèmes de compilation et d'interprétation incrémentielle.

2.1 Analyseurs syntaxiques.

Ces programmes qui réalisent la vérification syntaxique des programmes source sans génération de code objet ont été décrits précédemment. En programmation conversationnelle, ils doivent être réentrants et permettre de réaliser facilement les corrections.

Dans le cas de langages source à structure de blocs, lorsqu'une erreur de syntaxe a été détectée et corrigée, l'analyse syntaxique doit être reprise à partir du dernier bloc dont l'élément modifié est un constituant.

2.2 Interpréteurs.

Dans un système à accès multiple, il est utile d'avoir un mode d'utilisation en machine de bureau c'est-à-dire permettant l'évaluation des expressions, l'affectation et l'impression des résultats intermédiaires ou finaux.

Le langage utilisé dans ce mode est en fait un petit sous-ensemble du langage de base et peut être mis en oeuvre simplement par l'utilisation de techniques d'interprétation telles que celles décrites en 2e partie, chapitre I.

La plupart des langages évolués nécessitent également un mécanisme d'interprétation pendant l'exécution pour la mise en oeuvre des caractéristiques dynamiques. (Cf. 2e Partie. Chapitre III).

2.3 Compilateurs incrémentiels ou différentiels.

Dans un système à accès multiple, si la génération d'un programme objet est réalisée en même temps que l'analyse syntaxique alors chaque fois qu'une erreur est détectée ou qu'une modification est faite volontairement par le programmeur, une recompilation du programme source est nécessaire. Ceci est dû à ce qu'un programme objet est considéré comme un seul bloc d'instructions reliées implicitement les unes aux autres par l'ordre de succession et d'imbrication reflétée dans le programme objet.

La compilation incrémentielle (ou différentielle) des programmes c'est-à-dire la compilation indépendante des instructions par l'utilisation d'indicateurs explicites pour définir l'enchaînement des instructions rend possible des corrections ou des changements au prix d'une recompilation de la seule instruction intéressée.

Facile à mettre en oeuvre pour des langages dont la structure de programme est simple, la compilation incrémentielle est beaucoup plus difficile pour les langages évolués caractérisés par la définition récursive de la notion d'instruction.

2.4 Compilateurs symétriques.

L'utilisation d'un système à accès multiple est basée sur les possibilités de bibliothèque et de manipulation de fichiers notamment pour ranger et rechercher l'information à partir d'un terminal. Lors de la compilation des programmes, on doit conserver à la fois la forme source et la forme objet des programmes. Certains compilateurs dits symétriques conservent uniquement la forme objet : la forme source étant reconstruite, si cela est nécessaire, à partir de la forme objet.

2.5 Interpréteurs incrémentiels.

Dans un système à accès multiple, non seulement la compilation mais aussi l'exécution des programmes doivent être conçues de manière à permettre un véritable dialogue entre le programme et le système. Si les instructions d'un programme peuvent être compilées indépendamment, elles peuvent aussi être exécutées indépendamment pourvu que soient définies et accessibles les valeurs nécessaires à leur évaluation.

Nous allons décrire trois systèmes qui permettent d'illustrer les concepts précédents.

III - UN COMPILATEUR FORTRAN INCREMENTIEL ET SYMETRIQUE [41].

Le système comprend :

- un calculateur 7040, avec une unité de disques 1301, un tambour magnétique 7320 et une unité de contrôle de transmissions 7740 auxquels peuvent être connectés une série de terminaux du type 1050.

- un ensemble de programmes permettant la mise en oeuvre du système conversationnel.

3.1 Langage source.

Le langage source est le langage Fortran II auxquels ont été ajoutées quelques instructions spéciales (commandes) permettant la mise au point, l'exécution et la manipulation des programmes.

3.2 Langage objet.

Le langage objet est un langage intermédiaire directement interprété pendant l'exécution. Ce langage intermédiaire comporte des instructions à deux adresses et simule le fonctionnement d'une machine à pile. (cf. 2e Partie, chapitre III).

Par exemple, l'instruction d'affectation :

$$X = ((-B) + \text{RAC2} (B ** 2 - 4. * A + C)) / (2. * A) \omega$$

sera transformée en :

⊖	B	
**	B	2
*	4.	A
*	temp	C
-	temp	temp
,	temp	
F	RAC2	
+	temp	temp
*	2.	A
/	temp	temp
=	X	temp
<u>ω</u>	temp	

⊖ désigne l'opérateur moins unaire

F désigne l'opérateur de fonction

temp désigne une mémoire temporaire contenant un résultat intermédiaire (nous avons vu que cette indication est nécessaire à moins d'hypothèse sur l'associativité et la commutativité des opérations).

En réalité les adresse symboliques et les constantes sont remplacées par des références aux enregistrements associés à ces éléments.

3,3 Représentation des programmes objet.

Enregistrements d'éléments.

Un élément de programme source peut être une étiquette, une constante, une variable, un tableau ou un nom de fonction.

Tout élément de programme source est représenté par un enregistrement d'élément de longueur fixe contenant l'information attachée à cet élément :

- référence symbolique externe
- valeur
- type, catégorie
- référence numérique interne

Tous les enregistrements d'éléments étant rangés en liste, l'enregistrement d'élément contient aussi l'adresse de l'enregistrement suivant.

Enregistrements d'instructions.

Les enregistrements d'instructions sont également structurés en listes afin de pouvoir optimiser la zone de mémoire affectée au rangement des instructions.

Tout enregistrement d'instruction est composé d'un nombre fixe de mots machine contenant l'information de nature générale et d'un nombre variable de mots contenant la forme objet de cette instruction.

L'information de nature générale comporte :

- un numéro d'ordre affecté par le système et utilisé par le programmeur pour la modification et l'exécution des instructions.
- le type, l'étiquette éventuelle, l'adresse de l'enregistrement d'instruction suivant, l'adresse de l'instruction suivante du même type.

Listes d'enregistrements.

Il y a 28 listes d'enregistrement d'élément : une pour chacune des 26 lettres, une pour les constantes entières et une pour les constantes réelles.

Chaque enregistrement d'élément associé à un identificateur est rangé dans la liste associée avec la première lettre de cet identificateur.

Il y a 2 listes d'enregistrement d'instructions : la première correspond aux numéros d'ordre et est utilisée pour la communication entre le programmeur et son programme ; la seconde correspond aux instructions de même type et est utilisée pour la vérification de l'imbrication correcte des boucles Pour et les ruptures de séquence.

3.4 Décomposition des expressions.

Le compilateur d'expressions arithmétiques utilise un algorithme de double priorité.

Le programme objet est une forme intermédiaire entre la notation postfixée et un langage d'instructions à 2 adresses.

La table de priorité suivante est utilisée dans le système Quiltran.

<u>Opérateur</u>	<u>Priorité de pile</u>	<u>Priorité normale</u>
+ -	5	5
* /	4	4
**	4	3
I (op. indexage)	6	illégal
F (op. de fonction)	6	illégal
=	7	illégal
⊖ (moins unaire)	5	1
,	6	5
(6	illégal
)	illégal	6
<u>ω</u>	illégal	7

3.5 Exécution des programmes objet.

Toutes les instructions sont exécutées interprétativement sous le contrôle d'un programme principal. L'exécution d'une instruction appelle un sous-programme particulier qui peut lui-même appeler d'autres sous-programmes. Par exemple, le sous-programme d'instruction d'affectation et le sous-programme de saut conditionnel utilise le sous-programme évaluation d'expression.

Le sous-programme "instruction faire" utilise une pile spéciale pour la vérification de l'imbrication correcte des "instructions faire" (définition récursive de l'instruction faire).

A la fin d'une instruction, le sous-programme correspondant transmet l'adresse de l'instruction suivante au programme principal.

La mise au point pendant l'exécution est rendue facile par l'utilisation des références symboliques externes pour désigner les quantités.

3.6 Recomposition des expressions et reconstruction des programmes source.

A l'exception des expressions arithmétiques, la reconstruction des programmes source peut être réalisée directement à partir des enregistrements d'instruction qui contient sous forme codée l'information de l'instruction source dans le même ordre. Pour les expressions ; il est nécessaire de procéder à une recomposition (ou décompilation) à partir de la forme objet.

C'est la possibilité de cette recomposition qui caractérise un compilateur symétrique.

Nous allons d'abord étudier un cas simple de recomposition des expressions postfixées en expressions complètement parenthésées. Nous décrirons ensuite une recomposition de la forme objet décrite précédemment en expressions parenthésées minimales (par expression parenthésée minimale nous signifions une expression dans laquelle il n'existe aucune paire de parenthèses superflues).

3.6.1. Recomposition des expressions postfixées en expressions complètement parenthésées à l'aide d'un balayage séquentiel de droite à gauche.

L'expression postfixée est représentée par le tableau POSTFIXE [1 : 100] et, l'expression complètement parenthésée reconstruite par le tableau RECONPOSE [1 : 150].

L'algorithme de transformation utilise une pile pour les opérateurs et les parenthèses gauches, représentée par le tableau PILE [1 : 50].

L'expression postfixée est balayée de droite à gauche et l'expression complètement parenthésée est également reconstruite de droite à gauche.

L'occurrence d'un opérateur provoque le transfert d'une parenthèse gauche sur la pile et le transfert d'une parenthèse droite dans l'expression reconstruite ; on transfère ensuite le contenu de la pile dans l'expression reconstruite jusqu'au transfert d'un opérateur binaire.

On suppose que l'expression postfixée est délimitée à droite par OMEGA.

Par exemple l'expression postfixée :

$$\omega \quad A \oplus B + C \ominus D - \times$$

sera recomposée sous la forme de l'expression complètement parenthésée suivante :

$$(((\oplus A) + B) \times ((\ominus C) - D))$$

DEBUT

```

COMMENTAIRE RECOMPOSITION D UNE EXPRESSION COMPLETEMENT
PARENTHESEE ;
ENTIER I, J, K, PARGAUCHE, PARDROITE, OMEGA ;
ENTIER TABLEAU POSTFIXE[1:100], RECOMPOSE[1:150],
PILE[1:50] ;
ENTIER PROCEDURE NOP(X) ;
DEBUT COMMENTAIRE CETTE PROCEDURE DONNE LE NOMBRE
D OPERANDES ASSOCIE A L OPERATEUR X, OU 0 SI X
EST UNE PARENTHESE ;

```

FIN ;

```

BOOLEEN PROCEDURE OPERATEUR(X) ;
BOOLEEN PROCEDURE OPERANDE(X) ;

```

```

I:=101 ; J:=150 ; K:=1 ;
POUR I:=I-1 TANTQUE POSTFIXE[I] # OMEGA FAIRE

```

DEBUT

SI OPERATEUR(POSTFIXE[I]) ALORS

DEBUT

```

PILE[K]:=PARGAUCHE ;
PILE[K+1]:=POSTFIXE[I] ;
RECOMPOSE[J]:=PARDROITE ;
K:=K+2 ; J:=J-1

```

FIN ;

SI OPERANDE(POSTFIXE[I]) ALORS

DEBUT

RECOMPOSE[J]:=POSTFIXE[I] ;

J:=J-1 ;

E: K:=K-1 ;

RECOMPOSE[J]:=PILE[K] ;

J:=J-1 ;

SI NOP(PILE[K]) # 2 ET K # 1 ALORS

ALLERA E

FIN

FIN

FIN

3.6.2. Recomposition des expressions à partir du langage intermédiaire à 2 adresses.

Cette recombposition est réalisée au moyen de deux balayages successifs.

- un balayage séquentiel de l'expression objet en langage intermédiaire afin de construire séquentiellement une liste d'éléments interconnectés.

- un balayage de la liste d'éléments ainsi construite afin de reconstituer l'expression initiale sous forme minimale.

Construction de la liste d'éléments interconnectés.

L'algorithme de construction est décrit sous la forme d'un programme Algol.

Le programme source en langage intermédiaire est représenté par le tableau à 2 dimensions $\text{MACRO}[1 : N, 1 : 3]$ et utilise trois piles :

- une pile d'opérateurs (PILE OP) pour le réarrangement des opérateurs et deux piles de travail (T PILE GAUCHE, T PILE DROITE) pour le rangement des mots de contrôle. (Un mot de contrôle est composé de deux pointeurs repérant le début et la fin d'une chaîne d'éléments).

Un élément du programme source qui n'est pas une adresse de résultat intermédiaire est d'abord transféré dans une zone commune définie par les tableaux LISTE GAUCHE et LISTE DROITE.

- si cet élément est un opérateur il est rangé sur la pile, ensuite le ou les mots de contrôle sommets de la pile de travail et l'opérateur sont combinés pour former un nouveau mot de contrôle qui est transféré dans la liste d'éléments interconnectés (tableaux CHAINE GAUCHE et CHAINE DROITE).

- si cet élément est un symbole, on forme un mot de contrôle et on le range au sommet de la pile de travail. Si l'élément est une adresse de résultat intermédiaire, le dernier mot de contrôle de la liste d'éléments interconnectés est effacé et placé au sommet de la pile de travail.

Lorsqu'un mot de contrôle est effacé de la liste d'éléments, le sommet de la pile opérateurs est comparé à l'opérateur de l'instruction en cours de traitement pour déterminer si l'on doit insérer des parenthèses au début et à la fin de l'expression désignée au sommet de la pile de travail.

Pour cette comparaison on utilise la table de priorités suivante :

<u>Opérateur</u>	<u>Priorité de pile</u>	<u>Priorité normale</u>
+ -	2	1
*	3	2
**	3	3
I	7	1
F	7	1
=	7	1
⊖	2	2
,	illégal	1

Recomposition de l'expression minimale à partir de la chaîne d'éléments.

Lorsque ce balayage démarre, le premier mot des tableaux CHAÎNE DROITE et CHAÎNE GAUCHE contiennent les valeurs des pointeurs associés au premier et au dernier élément de l'expression à reconstruire.

La recomposition consiste alors simplement à lister les éléments dans l'ordre défini par les pointeurs.

DEBUT COMMENTAIRE RECOMPOSITION DES EXPRESSIONS

A PARTIR DE LA LISTE DES MACROS ;
 ENTIER TABLEAU MACRO[I:N,1:3],
 LISTE GAUCHE,LISTE DROITE,SORTIE[1:150],
 PILE OP[1:30],TPILE GAUCHE,TPILE DROITE[1:20],
 CHAINE GAUCHE,CHAINE DROITE[1:100] ;
 ENTIER I,J,L,OP,OPCHAINE,S,T,PARDROITE,PARGAUCHE,
 VIRGULE,COURANT,R ;

BOOLEEN PROCEDURE SYMBOL(X) ;

DEBUT COMMENTAIRE CETTE PROCEDURE NE PREND LA VALEUR
 VRAI QUE SI X CORRESPOND A UN SYMBOLE ;

FIN ;

BOOLEEN PROCEDURE TEMP(X) ;

DEBUT COMMENTAIRE CETTE PROCEDURE NE PREND LA VALEUR
 VRAI QUE SI X CORRESPOND A UN RESULTAT INTERMEDIAIRE ;

FIN ;

COMMENTAIRE LES PROCEDURES OPUN,OPBIN,OPINDFO,NE PRENNENT

LA VALEUR QUE POUR UN OPERATEUR RESPECTIVEMENT:

UNAIRE,BINAIRE,D INDEXATION OU DE FONCTION ;

BOOLEEN PROCEDURE OPUN(X) ; ... ;

BOOLEEN PROCEDURE OPBIN(X) ; ... ;

BOOLEEN PROCEDURE OPINDFO(X) ; ... ;

ENTIER PROCEDURE PRIORITE DE PILE(X) ; ... ;

ENTIER PROCEDURE PRIORITE NORMALE(X) ; ... ;

PROCEDURE EMPILER OP(X) ; ENTIER X ;
 DEBUT OP:=OP+1 ; PILE OP[OP]:=X FIN ;

ENTIER PROCEDURE CODE(X) ; ... ;

ENTIER PROCEDURE NOM(X) ; ... ;

ENTIER PROCEDURE VALEUR(X) ; ... ;

BOOLEEN PROCEDURE OPERATEUR(X) ; ... ;

BOOLEEN PROCEDURE IDENTIFICATEUR(X) ; ... ;

L:=OP:=S:=T:=0 ;
 POUR I:=1 PAS 1 JUSQUA N FAIRE
 DEBUT
 POUR J:=3,2 FAIRE


```

DEBUT
  SI SYMBOL(MACRO[I,J]) ALORS
    DEBUT
      L:=L+1 ;
      LISTE GAUCHE[L]:=MACRO[I,J] ;
      T:=T+1 ;
      TPILE GAUCHE[T]:=TPILE DROITE[T]:=L ;
    FIN
  SINON SI TEMP(MACRO[I,J]) ALORS
    DEBUT
      OPCHAINE:=PILE OP[OP] ; OP:=OP-1 ;
      SI PRIORITE NORMALE(MACRO[I,1]) >
        PRIORITE DE PILE(OPCHAINE)
      OU
      J=3 ET PRIORITE DE PILE(OPCHAINE)=
        PRIORITE DE PILE(MACRO[I,1])
      ALORS
        INSERTION PAR:
          DEBUT
            L:=L+2 ;
            LISTE GAUCHE[L-1]:=PAR GAUCHE ;
            LISTE GAUCHE[L]:=PAR DROITE ;
            LISTE DROITE[L-1]:=CHAINE GAUCHE[S] ;
            CHAINE GAUCHE[S]:=L+1 ;
            LISTE DROITE[CHAINE DROITE[S]]:=L ;
            CHAINE DROITE[S]:=L
          FIN ;

          T:=T+1 ;
          TPILE GAUCHE[T]:=CHAINE GAUCHE[S] ;
          TPILE DROITE[T]:=CHAINE DROITE[S] ;
          S:=S-1 ;
        FIN
      FIN ;
    SI OPBIN(MACRO[I,1]) ALORS
      DEBUT EMPILEROP(MACRO[I,1]) ;
        L:=L+1 ;
        LISTE GAUCHE[L]:=MACRO[I,1] ;
        LISTE DROITE[L]:=TPILE GAUCHE[T-1] ;
        LISTE DROITE[TPILE DROITE[T]]:=L ;
        S:=S+1 ;
        CHAINE GAUCHE[S]:=TPILE GAUCHE[T] ;
        CHAINE DROITE[S]:=TPILE DROITE[T-1] ;
        T:=T-2
      FIN ;
    SI OPUN(MACRO[I,1]) ALORS
      DEBUT EMPILER OP(MACRO[I,1]) ;
        L:=L+1 ;
        LISTE GAUCHE[L]:=MACRO[I,1] ;
        LISTE DROITE[L]:=TPILE GAUCHE[T] ;
        S:=S+1 ;
        CHAINE GAUCHE[S]:=L ;

```

```

        CHAINE DROITE[S]:=TPILE DROITE[T] ;
        T:=T-1
    FIN ;
SI OPINDFO(MACRO[I,1]) ALORS
CHAINAGE:
    DEBUT
        PROCEDURE LIER(SEP) ; ENTIER SEP ;
            DEBUT L:=L+1 ;
            LISTE GAUCHE[L]:=SEP ;
            LISTE DROITE[TPILE DROITE[T]]:=
                CHAINE DROITE[S]:=L ;
            FIN LIER ;
        EMPILER OP(MACRO[I,1]) ;
        S:=S+1 ;
        CHAINE GAUCHE[S]:=TPILE GAUCHE[T] ;
        SI T=1 ALORS CHAINE DROITE[S]:=TPILE DROITE[T]
            SINON LIER(PAR GAUCHE) ;
    VIDER:
        POUR T:=T-1 TANTQUE T ≠ 0 FAIRE
            DEBUT
                LISTE DROITE[CHAINE DROITE[S]]:=
                    TPILE GAUCHE[T] ;
                LIER( SI T ≥ 2 ALORS VIRGULE
                    SINON PAR DROITE)
            FIN
    FIN ;
    COMMENTAIRE LES OPERATEURS VIRGULE NE SONT JAMAIS EMPILES ;
FIN ;

RECOMPOSITION:
DEBUT I:=1 ;
    COURANT:=CHAINE GAUCHE[I] ;
    A:
        SI OPERATEUR(LISTE GAUCHE[COURANT]) ALORS
            DEBUT
                R:=CODE(LISTE GAUCHE[COURANT]) ;
                ALLERA SORTIR
            FIN ;
        SI IDENTIFICATEUR(LISTE GAUCHE[COURANT]) ALORS
            DEBUT
                R:=NOM(LISTE GAUCHE[COURANT]) ;
                ALLERA SORTIR
            FIN ;
        CONSTANTE R:=VALEUR(LISTE GAUCHE[COURANT]) ;
        SORTIR: SORTIE[I]:=R ; I:=I+1 ;
        SI COURANT ≠ CHAINE DROITE[I] ALORS
            DEBUT COURANT:=LISTE DROITE[COURANT] ;
                ALLERA A
            FIN ;
FIN ;
FIN
FIN

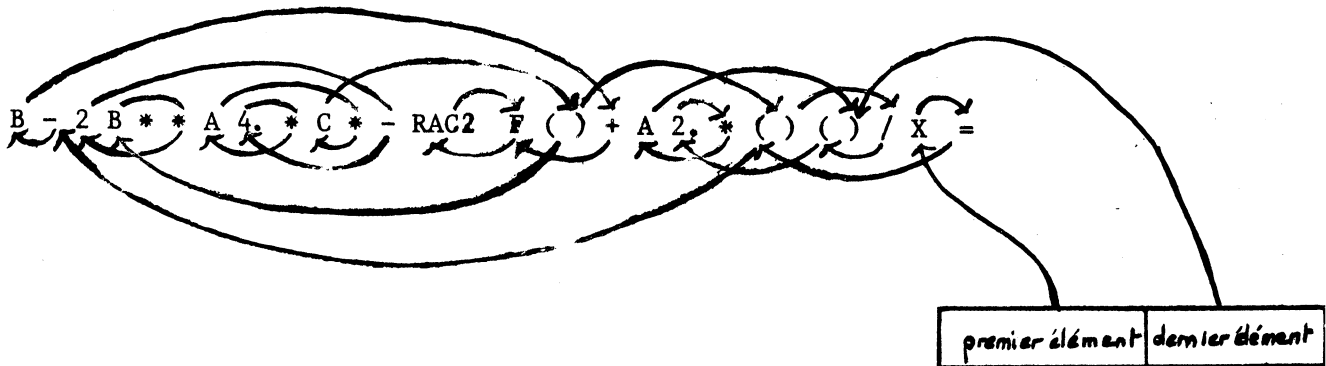
```

Exemple : la liste d'instructions

```

- B
** B 2
* 4. A
+ t c
- t t
, t
F RAC2
+ t t
* 2. A
/ t t
= X t
ω t
    
```

donne la liste d'éléments interconnectés suivante :



qui fournit l'expression minimale :

$$X = (- B + RAC2 (B ** 2 - 4. * A * C)) / (2. * A)$$

C H A P I T R E II

COMPILATEURS ET INTERPRETEURS INCREMENTIELS.1 - LA COMPILATION INCREMENTIELLE.

Il y a deux méthodes de base pour la construction des programmes :

a) la méthode récursive qui consiste à construire un programme en une seule phase en partant du bloc le plus extérieur et en continuant par les blocs intérieurs successifs. La méthode de construction est récursive en ce sens que l'on commence la construction d'un nouveau bloc avant que celle des blocs qui le contiennent soit entièrement terminée. C'est la méthode habituellement utilisée dans la construction des programmes Algol.

b) la méthode itérative qui consiste à construire un programme en autant de phases élémentaires successives qu'il y a de blocs dans le programme. Un programme complexe est ainsi construit par assemblage de blocs élémentaires. Pour un langage comme Algol, où il n'est pas possible de construire et de compiler séparément des blocs ou des procédures, cette méthode ne peut pas être utilisée directement sans une recompilation complète du programme assemblé. On peut cependant définir un certain nombre d'extensions qui permettent d'avoir cette possibilité dans le cadre d'un système à accès multiple.

Une compilation incrémentielle basée sur la méthode récursive ne nécessite que la compilation séparée et indépendante des instructions alors que la méthode itérative exige en plus la compilation séparée et indépendante des blocs et des procédures ainsi que les extensions adéquates notamment pour définir des quantités de type externe et l'équivalence des désignations lors de l'assemblage des programmes.

D'un point de vue purement syntaxique, il n'y a évidemment aucune différence entre un bloc et une instruction ; néanmoins il est logique de considérer un bloc comme un morceau de programme qui se suffit à lui-même, ce qui n'est pas le cas d'une instruction.

2 - L'EVALUATION INCREMENTIELLE.

Les mêmes concepts peuvent être utilisés pour l'évaluation des programmes. Une instruction peut être évaluée indépendamment des autres pourvu que l'on définisse les valeurs des quantités non locales à cette instruction et que cette évaluation isolée ait un sens (on ne peut pas par exemple exécuter séparément une instruction à l'intérieur d'un bloc ou d'une déclaration de procédure).

Un programme peut ainsi être évalué récursivement en une seule phase, ou itérativement en autant de phases qu'il y a de blocs élémentaires.

L'intérêt des techniques itératives dans les systèmes à accès multiple tient précisément aux possibilités de dialogue entre le programmeur et le système qui permettent de travailler à un niveau beaucoup plus fin et aux grandes possibilités de stockage et de recherche des fichiers.

Un programme complexe peut être décomposé en phases élémentaires construites et mises au point indépendamment dans une première étape et recomposées ensuite par assemblage pour la mise au point globale dans une seconde étape.

Nous allons décrire deux systèmes incrémentiels différents, l'un basé exclusivement sur la méthode récursive, l'autre sur une synthèse des deux méthodes : itérative au niveau des blocs et récursive au niveau des instructions.

3 - UN SYSTEME INCREMENTIEL POUR LA CONSTRUCTION ET L'EVALUATION RECURSIVE DES PROGRAMMES ALGOL [42].

3.1. Langage source.

Le langage source est identique à Algol 60 avec la notable exception d'une extension et d'une généralisation de la notion d'instruction et de programme.

Les changements relatifs aux notions de programme et d'instruction sont reflétés dans les définitions suivantes :

```

<programme> ::= <instruction>
<instruction> ::= <instruction non étiquetée> |
                <étiquette> : <instruction>
instruction non étiquetée ::= <bloc> |
                <instruction de déclaration> |
                <instruction d'affectation> |
                <instruction conditionnelle> |
                <instruction pour> |
                <instruction aller a> |
                <vide> |
                <numéro de ligne>
                <bloc> ::= debut <liste d'instructions> fin
<liste d'instructions> ::= <instruction> |
                <liste d'instructions> ; <liste d'instructions> |
                (<numéro de ligne> , <numéro de ligne>)
instruction de déclaration ::= <déclaration>
instruction conditionnelle ::= si <expression booléenne> alors <instruction> sinon <instruction>

```

Les principales différences concernant la définition des instructions sont les suivantes :

- pas de distinction entre bloc et instruction composée :
l'instruction composée devient un cas particulier du bloc.

- les déclarations sont considérées comme des instructions et de ce fait peuvent apparaître n'importe où à l'intérieur d'un bloc : les notions de tête de bloc et de fin d'instruction composée n'existent plus.

- la forme incomplète de l'instruction conditionnelle (instruction si) n'existe plus. Il y a donc identité de structure entre l'expression conditionnelle et l'instruction conditionnelle.

- un numéro de ligne est considéré comme une instruction et une paire de numéros de lignes comme une liste d'instructions. Les deux numéros de ligne correspondent respectivement aux limites inférieures et supérieures délimitant une liste d'instructions déjà construites.

3.2. Interconnection des instructions.

Etant donné que la définition de l'instruction est récursive, c'est-à-dire qu'une instruction peut contenir elle-même d'autres instructions, les liaisons entre les instructions sont beaucoup plus complexes que celles d'un programme en langage machine ou en langage symbolique élémentaire.

L'adresse d'une instruction dans une telle structure de programme ne se réduit pas à un simple numéro d'ordre. Cette adresse est définie par une liste de numéros reflétant la place de cette instruction dans la structure d'ensemble du programme (structure récursive due à l'imbrication des instructions). Dans une telle structure, le déroulement du programme ne peut plus être contrôlé par un simple compteur ordinal mais il est nécessaire d'utiliser une pile qui contient à chaque instant la liste des numéros d'instructions en cours d'exécution.

Considérons le programme Algol suivant dans lequel chacune des instructions est étiquetée numériquement :

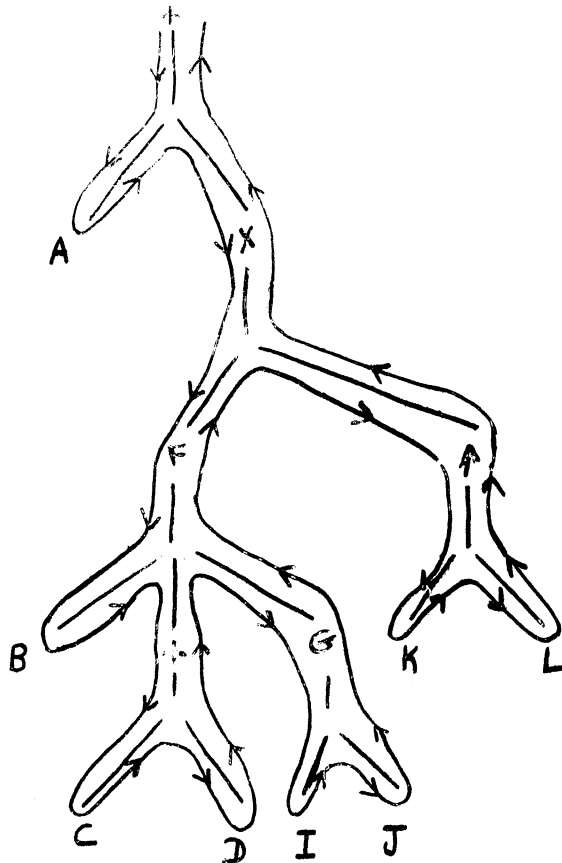
L'exécution directe de ce programme est un processus récursif étant donné que l'on commence l'exécution de nouvelles instructions avant que les précédentes soient terminées. Par exemple lorsque l'on exécute l'instruction étiquetée 8, l'exécution des instructions étiquetées 1, 4, 6 et 7 n'est pas encore terminée. Afin de contrôler cette exécution directe, une pile est nécessaire pour conserver les adresses d'instructions en cours d'exécution.

Ce mécanisme est donc tout à fait analogue à celui de l'évaluation directe d'une expression par la méthode récursive.

Par exemple, l'évaluation de l'expression

$$A + F(B, C+D, G(I, J)) \times K + L$$

serait représentée par le diagramme suivant



qui est tout à fait analogue au précédent : la pile d'évaluation de l'expression qui contient les éléments des opérations non encore terminées est analogue à la pile de contrôle de l'exécution du programme qui contient les adresses des instructions non encore complètement exécutées.

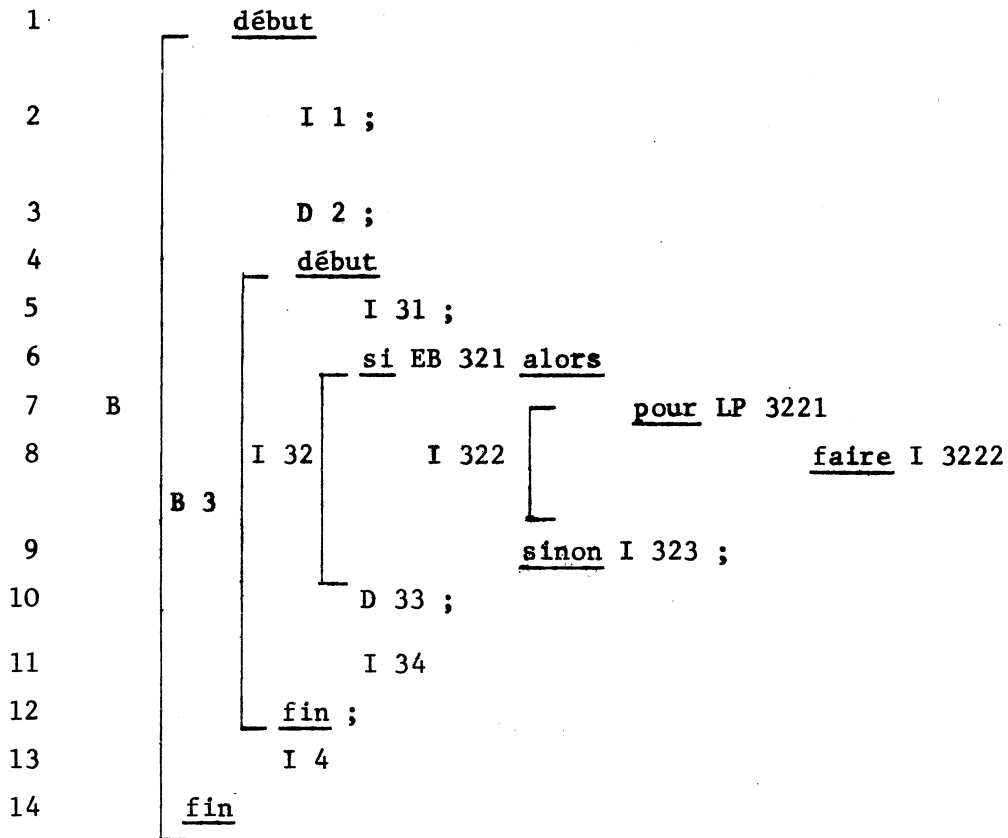
Dans le cas de la compilation incrémentielle, on veut pouvoir changer constamment la structure du programme en cours de construction tout en minimisant le coût de recompilation.

Afin de pouvoir compiler et ranger en mémoire les instructions indépendamment les unes des autres, celles-ci sont représentées sous forme d'un ensemble d'éléments interconnectés entre eux par des pointeurs. Le nombre de pointeurs dépend du type d'instruction : une instruction allera ou une instruction d'affectation ne contient pas d'autres instructions : une instruction pour contient une instruction, une instruction conditionnelle deux instructions, un bloc deux listes d'instructions (liste d'instructions de déclaration, liste des autres instructions).

La structure logique d'un programme est ainsi représentée par l'ensemble des pointeurs associés aux instructions.

Modifier, supprimer ou ajouter une instruction revient alors à changer les éléments de l'instruction correspondante et la valeur des pointeurs associés à cette instruction.

Considérons le programme schématique suivant :



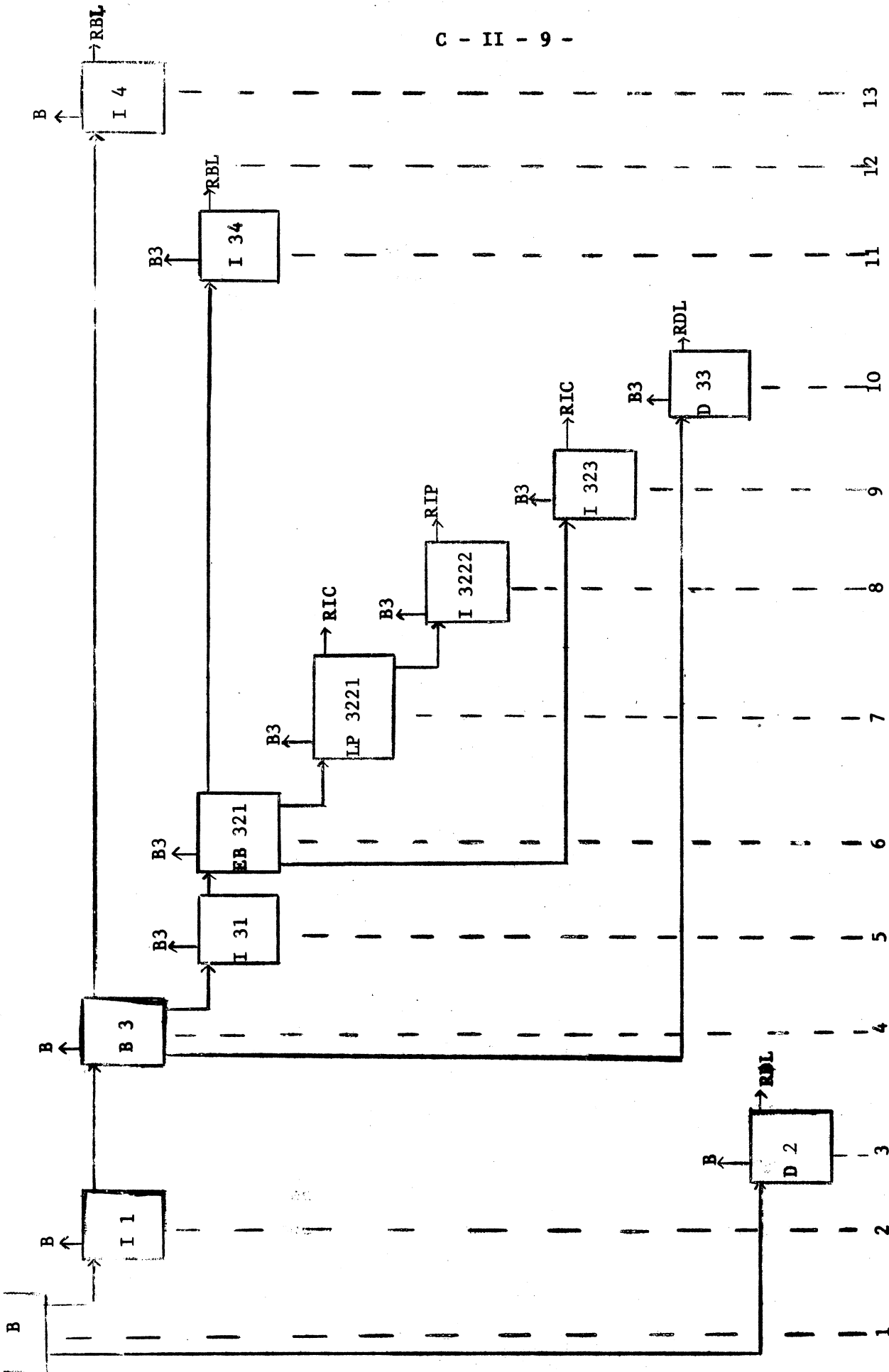
I, D, EB, LP représentent respectivement Instruction, Déclaration, Expression Booléenne, Liste de pour.

X_{ij} représente la j^e composante de l'élément syntaxique X_i .

X_{ijk} la k^e composante de l'élément X_{ij}

etc...

La structure logique de ce programme peut être représentée par le diagramme suivant :



RPR (Retour Programme)
 RBL (Retour Bloc)
 RIC (Retour Instruction conditionnelle)
 RIP (Retour Instruction Pour)
 RDL (Retour Déclaration)

RIP (Retour Instruction Pour)
 RDL (Retour Déclaration)

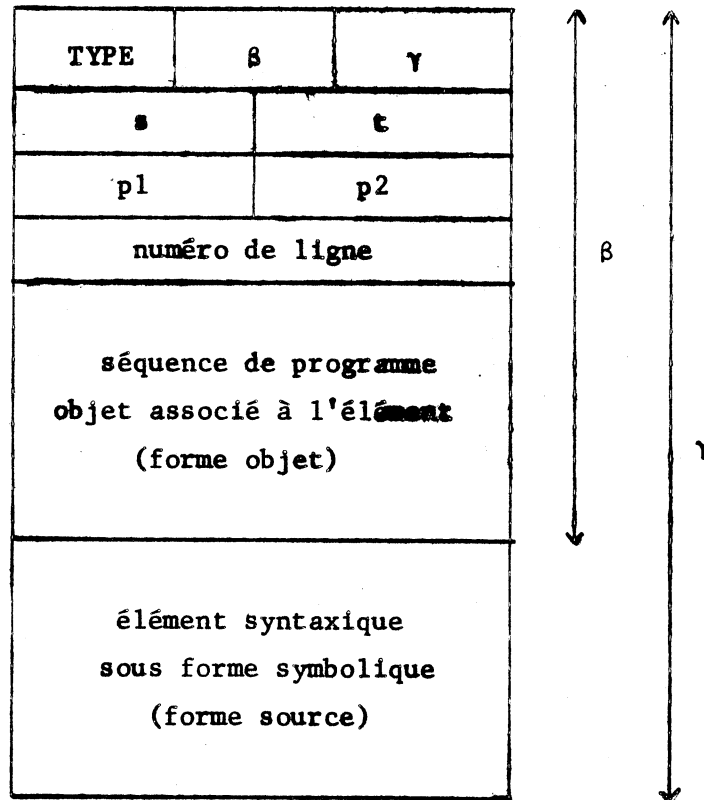
RIP (Retour Instruction conditionnelle)

On voit que l'ordre lexicographique reflété par les nombres entiers à la base du schéma et l'ordre logique reflété par les pointeurs sont très différents.

3.3. Représentation des éléments syntaxiques associés à une structure de programme.

Pour l'exemple donné précédemment, le nombre d'éléments syntaxiques est égal au nombre de rectangles apparaissant dans le diagramme de structure.

Un élément syntaxique est représenté par un nombre de mots variables contenant les indications suivantes :



- TYPE est un code qui indique le type de l'instruction : bloc, déclaration, procédure, instruction conditionnelle, instruction pour, autres instructions.

- s est un pointeur qui repère l'instruction suivante.
- t est un pointeur qui repère le bloc contenant cette instruction.
- p1 et p2 sont des pointeurs qui définissent les éléments d'une instruction qui contient d'autres instructions avec les conventions suivantes :

<u>Type de l'élément.</u>	<u>p1 repère</u>	<u>p2 repère</u>
bloc	liste des déclarations	liste des instructions
déclaration	inutilisé	inutilisé
procédure	"	corps de procédure
instruction conditionnelle	instruction après <u>alors</u>	instruction après <u>sinon</u>
instruction pour	inutilisé	" après <u>faire</u>
autres instructions	"	inutilisé

Le numéro de ligne est un nombre affecté par le système à chacun des éléments et utilisé par le programmeur pour référencer cet élément.

L'intérêt d'une telle représentation réside essentiellement dans la possibilité d'arranger dans un ordre quelconque les diverses instructions d'un même programme aussi bien que celles de nombreux programmes appartenant à des utilisateurs différents. Ceci peut être réalisé en utilisant une seule zone (liste libre) qui en cas d'encombrement trop important sera remis à jour au moyen d'un programme récupérateur destiné à libérer les zones qui ne sont plus utilisées.

3.4. Table des références symboliques.

Deux instructions quelconques d'un programme, outre les connexions logiques qui déterminent dans quel ordre elles doivent être exécutées, utilisent en général des identificateurs communs.

Afin de rendre possible l'utilisation d'identificateurs communs par deux instructions différentes et aussi de permettre au programmeur une communication plus facile au niveau du langage source, tous les identificateurs sont rangés dans une table de références symboliques contenant les désignations des identificateurs utilisés dans un programme. Toutes les références aux identificateurs sont réalisées par adressage indirect à l'intérieur de cette table pour toutes les instructions.

3.5. Modification des programmes.

La modification des programmes est réalisée selon le principe suivant : un nombre entier quelconque d'instructions internes (instructions à supprimer) peut être remplacé par un nombre entier quelconque d'instructions externes (instructions à insérer).

Les instructions internes et externes sont définies comme suit :

```

<instruction interne> ::= <point d'insertion> |
                        (<limite inférieure>, <limite supérieure>)
<point d'insertion> ::= <numéro de ligne> + |
                        <numéro de ligne> -
<limite inférieure> ::= <numéro de ligne>
<limite supérieure> ::= <numéro de ligne>
<numéro de ligne>   ::= <entier sans signe>. |
                        <numéro de ligne> <numéro de ligne>
<instruction externe> ::= <instruction> |
                        <numéro de ligne> |
                        (<limite inférieure>, <limite supérieure>)'
                        <instruction externe> ; <instruction externe>

```

Lorsqu'on insère plusieurs instructions entre deux instructions de numéros de ligne n. et n+1., ces instructions sont numérotées :

n.1., n.2., --- etc

Si le point d'insertion se trouve entre deux numéros n et n+1, il peut être défini comme n.+ ou n+1.-

Toutes les modifications sont exécutées au moyen d'une seule commande de compilation :

<commande de compilation> ::= compiler <instruction interne>, <instruction externe> ω

ω désigne une fin d'enregistrement qui délimite la commande.

Si l'instruction externe définit 2 limites, l'instruction interne initiale définie entre ces limites est remplacée par la nouvelle instruction interne.

Si l'instruction interne définit un point d'insertion, l'instruction externe est insérée en ce point à l'intérieur de l'instruction interne initiale.

S'il n'y a pas d'instruction externe définie dans la commande, on a effacement de l'instruction interne spécifiée.

3.6. Exécution des programmes.

L'exécution des programmes est réalisée par un moniteur qui contrôle le déroulement séquentiel du programme au moyen d'une pile d'adresses.

Chaque instruction est exécutée indépendamment et retourne le contrôle au moniteur d'exécution.

L'exécution est réalisée au moyen de commandes spéciales suivantes :

- arrêter et redémarrer l'exécution en un point déterminé (allera dynamique)
- arrêter et redémarrer l'exécution en un point quelconque (arrêt et reprise dynamiques).

4.2. Compilation incrémentielle itérative.

Dans la méthode de construction itérative des programmes, chaque bloc est construit et compilé indépendamment.

Un programme complet est construit à partir de ces blocs indépendants en utilisant un mécanisme d'inclusion des blocs basé sur les particularités suivantes :

a) bibliothèque de programmes.

Un bloc élémentaire peut être compilé et rangé dans le système de fichiers afin d'être utilisé plus tard ; un bloc est toujours constitué de deux parties : bloc source et bloc objet.

b) quantités externes.

Afin de pouvoir compiler un bloc de façon indépendante, il faut disposer d'un moyen pour désigner les identificateurs non locaux, c'est-à-dire déclarés à l'extérieur, mais utilisés à l'intérieur de ce bloc.

c) équivalence des désignations et structure de blocs.

Dans le processus de construction itérative, on est amené à construire des blocs qui contiennent des blocs de niveau inférieur déjà construits et mis au point. Lorsqu'un bloc de niveau inférieur utilisant des identificateurs non locaux doit être incorporé dans un bloc principal, il faut définir une équivalence de désignations entre les identificateurs non locaux du bloc incorporé et certains identificateurs locaux du bloc principal.

4.3. Evaluation incrémentielle.

Un bloc élémentaire déjà construit peut être évalué indépendamment pourvu que l'on définisse les valeurs des quantités non locales qui interviennent dans ce bloc et que cette évaluation séparée ait un sens dans le cadre du programme auquel il appartient.

Le résultat de l'évaluation, c'est-à-dire les nouvelles valeurs des quantités non locales, peut être utilisé par la suite dans l'évaluation des blocs contenant ce bloc élémentaire.

4.4. Langage de programmation et langage de commande.

Le langage de programmation utilisé est le langage ALGOL 60 auquel on a ajouté un certain nombre d'instructions qui constituent le langage de commande proprement dit.

Le langage de commande permet d'exécuter des opérations sur les quantités du type file.

Une file est une séquence d'éléments de l'un des 4 catégories suivantes : bits, caractères, mots-machine, lignes (c'est-à-dire groupement d'un des 3 types précédents).

Une file est en général trop longue pour être définie explicitement. On utilise habituellement des identificateurs qui désignent des files et des pointeurs qui désignent des points à l'intérieur de ces files.

Il y a 4 opérations fondamentales sur les files :

- insérer (élément, pointeur, file) pour l'insertion d'un élément dans une file au point défini par la valeur du pointeur.

- extraire (pointeur 1, pointeur 2, file) pour l'extraction d'une partie de file comprise entre les limites définies par pointeur 1 et pointeur 2.
- concaténer (file 1, file 2) pour la concaténation de deux files.
- substituer (<liste de files>) pour l'exécution d'opérations de substitution sur des files. Par exemple un assemblage ou une compilation est une opération qui transforme une file source en une file objet au moyen d'une substitution statique. Une exécution qui équivaut à l'interprétation d'une file, en tant que programme auquel sont associées des files d'entrée et de sortie, correspond à une opération de substitution dynamique.

La plupart des opérations sur les files peuvent être construites à partir de ces opérations fondamentales ; dans la pratique il est néanmoins plus commode d'utiliser une liste plus développée d'opérations sur les files.

4.5. Modes d'utilisation du système.

Il y a 3 modes ou niveaux d'utilisation du système :

- le mode système pour l'exécution des commandes de manipulation de files.
- le mode compilation pour l'exécution des commandes de compilation avec deux sous-modes : conversationnel et séquentiel.
- le mode interprétation pour l'exécution des commandes d'exécution avec deux sous-modes : programme et calculateur de bureau. Le langage de commande est défini syntaxiquement par les règles suivantes : (ω joue le rôle de délimiteur de fin de commande).

```

<activité> ::= <activité bibliothèque> | <activité utilisateur>
<activité bibliothèque> ::= debut ω
                                COMMUN (<mot de passe>) ω
                                <liste de travaux> ω
                                fin ω
<activité utilisateur> ::= debut ω
                                NOM (<nom d'utilisateur>, <mot de passe>) ω
                                <liste de travaux> ω
                                fin ω
<liste de travaux> ::= <description de travail> |
                                <liste de travaux> ω <description de travail>
<description de travail> ::= FILE (<nom de file>) ω
                                <liste de commandes>
<liste de commandes> ::= <commande> |
                                <liste de commandes> ω <commande>
<commande> ::= <commande de manipulation de file> |
                                <commande de compilation> |
                                <commande d'interprétation>

```

L'accès au système est réalisé au moyen de la commande début et la sortie au moyen de la commande fin. Ces deux commandes délimitent obligatoirement une activité qui est l'équivalent dans le langage de commande d'un programme dans un langage de programmation. En fait pour un utilisateur travaillant avec un système conversationnel, il n'y a pas à proprement parler de différence entre les deux niveaux de langage et ceci est une des raisons pour lesquelles nous avons essayé de donner au langage de commande une forme similaire à celle du langage de programmation Algol.

Il y a deux sortes d'activités :

- activité bibliothèque qui permet l'accès à l'ensemble des fichiers communs et qui est réservée aux responsables du système.

- activité utilisateur qui donne l'accès aux fichiers propres à chaque utilisateur mais ne donne l'accès aux fichiers communs que pour les opérations de lecture.

La possibilité de démarrer une activité est toujours conditionnée par la validité d'un mot de passe donné comme paramètre de la commande COMMUN pour une activité bibliothèque et de la commande NOM pour une activité utilisateur. Une activité est toujours étiquetée avec le nom d'un utilisateur, ce nom est défini implicitement par le mot de passe dans le cas d'une activité bibliothèque et est donné comme paramètre de la commande NOM pour une activité utilisateur.

Une activité est composée d'une liste de travaux, chaque description de travail comporte une commande FILE avec un paramètre nom de file qui désigne ce travail particulier suivie d'une liste de commandes.

4.6. Mode système et commandes de manipulation des files.

Ce mode a la plus haute priorité c'est-à-dire que l'accès aux modes compilation et interprétation ne peut s'effectuer qu'à l'intérieur de celui-ci.

Le mode système comprend outre les commandes début, fin, COMMUN, NOM et FILE qui permettent l'initialisation d'une activité ou d'un travail, les commandes de manipulation de file proprement dites définies de la façon suivante :

```
<commande de manipulation de file> ::=
CONCAT (<nom de file>)|
INSERER (<pointeur>,<nom de file>)|
EXTRAIRE (<pointeur 1>,<pointeur 2>,<nom de file>,<mode d'accès>)|
LISTER (<pointeur 1>,<pointeur 2>)|
EFFACER (<pointeur 1>,<pointeur 2>)|
COPIER (<nom de file>,<mode d'accès>)|
COPIER (<nom de file>,<nom d'utilisateur>,<mode d'accès>)|
LIRE (<unité externe>)|
```

ECRIRE (<unité externe>)|
RANGER (<mode d'accès>)|
FRANGER|
ENLEVER|
EXEC(ENTREE(<liste de noms de file>),SORTIE(<liste de noms de file>))

Le mode d'accès précise les spécifications d'accès pour une file donnée : LS (Lecture Seule), LE (Lecture Ecriture). Par exemple les trois commandes : EFFACER, CONCAT and INSERER ne peuvent pas être utilisées avec des files dont le mode d'accès est LS.

Au cours d'une activité bibliothèque, on ne peut utiliser que les commandes de manipulation de file excepté la commande EXEC.

Les commandes de manipulation de files permettent l'exécution d'opérations élémentaires sur les files selon les règles suivantes :

Opérations unaires.

La file opérande et la file résultat (si elle existe) ont la même désignation qui est celle de la file définie dans la description de travail associée à cette commande.

Il y a 5 commandes d'opérations unaires de manipulation de files :

- LISTER (<pointeur 1>,<pointeur 2>) : listage de la section de file opérande comprise entre les limites désignées par pointeur 1 et pointeur 2. Cette section de file opérande devient la file résultat et la file opérande initiale est perdue.

- EFFACER (<pointeur 1>, <pointeur 2>) : effacement de la section de file opérande comprise entre pointeur 1 et pointeur 2. La file opérande initiale moins la section supprimée devient la file résultat.
- RANGER (<mode d'accès>)
L'information qui suit cette commande est considérée comme une file et rangée dans le système avec la désignation de la file opérande avec la spécification de son mode d'accès.
- FRANGER n'est pas à proprement parler une commande d'opération. Elle sert à délimiter la fin de la file qui suit la commande RANGER et se trouve obligatoirement associée avec cette commande (parenthèses de délimitation de file).
- ENLEVER suppression de la file opérande.
- LIRE (<unité externe>) et ECRIRE (<unité externe>) permettent l'entrée et la sortie directe d'une file à partir envers une unité externe dont la désignation est celle du premier opérande.

Opérations binaires.

La file premier opérande et la file résultat ont la même désignation qui est celle définie dans la description de travail associée. La file deuxième opérande est définie comme paramètre de la commande.

Il y a 4 commandes d'opérations binaires :

- CONCAT (<nom de file>) : la file premier opérande est concaténée avec la file deuxième opérande et devient la file résultat, la file deuxième opérande reste inchangée.
- INSERER (<pointeur>, <nom de file>) : la file deuxième opérande est insérée à l'intérieur de la file premier opérande au point défini par la valeur du pointeur. La file deuxième opérande reste inchangée et la file premier opérande après l'opération d'insertion devient la file résultat.

- EXTRAIRE (<pointeur 1>, <pointeur 2>, <nom de file>, <mode d'accès>)
la section de la file deuxième opérande comprise entre pointeur 1 et pointeur 2 devient la file résultat (premier opérande) avec le mode d'accès spécifié. La file deuxième opérande reste inchangée.
- COPIER (<nom de file>, <mode d'accès>)
cas particulier de la commande extraire pour une duplication complète de la file deuxième opérande avec comme désignation la file résultat (premier opérande) avec le mode d'accès spécifié. Afin de permettre à un utilisateur de dupliquer une file appartenant à un autre utilisateur, on se sert de la commande :
- COPIER (<nom de file>, <nom d'utilisateur>, <mode d'accès>)
cette commande permet la duplication de la file deuxième opérande appartenant à l'utilisateur désigné en paramètre avec comme désignation la file premier opérande de l'utilisateur en cours d'activité et avec le mode d'accès spécifié.

Opérations n-aires.

Il existe une seule commande de manipulation de file ayant un nombre d'opérandes variable

EXEC(ENTREE(<liste de noms de file>), SORTIE(<liste de noms de file>))

qui permet l'exécution d'un programme défini par la file premier opérande et utilisant comme files d'entrée et de sortie les files désignées dans les listes correspondantes.

4.7. Mode compilateur et commandes de compilation.

L'entrée dans ce mode s'effectue au moyen de la commande COMPILER et la sortie au moyen de la commande FCOMP.

Les commandes de compilation sont définies syntaxiquement de la façon suivante :

```

<commande de compilation> ::= COMPILER(<nom de langage>,<nom de file>) |
                                FCOMP |
                                SCOMP |
                                RCOMP |
                                REMPLACER(<numéro syntaxique>) |
                                FREEMPLACER |
                                INST(<numéro syntaxique>) |
                                TRANSFORMER(<nom de file 1>,<nom de file 2>) |
                                EXTERNE(<partie spécification>) |
                                EQUIV(<liste de paires d'identificateurs>) |
                                INCLURE(<nom de file 1>,<nom de file 2>) |
<liste de paires d'identificateurs> ::= <identificateur> : <identificateur> |
                                         <liste de paires d'identificateurs>,
                                         <identificateur> : <identificateur>

```

Dans le mode compilateur il existe en réalité deux sous-modes :

- mode moniteur
- mode conversationnel

4.7.1. Mode moniteur.

La commande

COMPILER (<nom de langage>,<nom de file>) permet la compilation du programme source écrit dans le langage source défini par le nom de langage, premier paramètre, et désigné par le nom de file, deuxième paramètre.

Le résultat de la compilation (programme objet) est défini par le nom de file, paramètre de la commande FILE qui définit le travail correspondant. Les 3 langages : Algol M (M pour mode Moniteur), Fortran IV et Cobol sont utilisables dans le mode moniteur.

La fin d'une compilation doit être indiquée au moyen de la commande FCOMP.

4.7.2. Mode conversationnel.

Il y a un seul langage : Algol C (C pour Conversationnel) utilisable en mode conversationnel. Le programme source doit être écrit immédiatement à la suite de la commande de compilation conversationnelle et constitue la file désignée dans la commande de compilation (file définissant le programme source). Le programme objet est rangé dans la file paramètre de la commande COMPILER.

- Le compilateur Algol conversationnel est organisé de la façon suivante :
- l'évaluation syntaxique est réalisée séquentiellement à l'aide d'une seule pile (pile d'évaluation syntaxique) et le texte objet est généré au fur et à mesure de l'analyse.

La compilation utilise les files suivantes (mis à part le compilateur lui-même) :

- texte source
- texte objet
- pile d'évaluation syntaxique
- table des références symboliques
- zone de travail du compilateur (réentrant)

Toutes les références aux quantités symboliques sont réalisées indirectement au moyen de la table des références symboliques construite pour chaque niveau de bloc. A la fermeture d'un bloc, la partie de la table correspondant à ce niveau est transférée dans le programme objet.

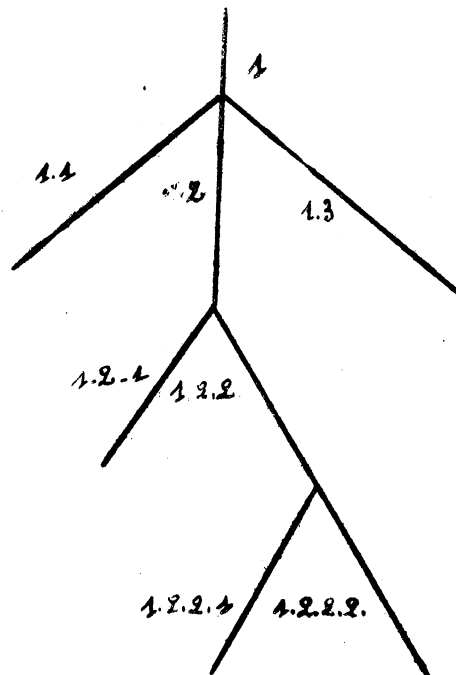
Dans ce mode conversationnel la compilation peut être interrompue à n'importe quel moment et on peut ranger l'état du compilateur au moment de l'interruption au moyen de la commande SCOMP qui préserve les files associées à la compilation interrompue. Cet état peut être restauré au moyen de la commande RCOMP.

4.7.2.1. Numérotation syntaxique et compilation incrémentielle des instructions.

Afin de rendre plus aisée la communication avec le programme au niveau des instructions aussi bien pendant la compilation que pendant l'interprétation, on réalise une numérotation syntaxique des instructions du texte source transmis par les terminaux. Parallèlement à la lecture du texte source caractère par caractère, on effectue une analyse syntaxique au niveau des instructions afin de déterminer le numéro syntaxique attaché à chacune des instructions. Etant donné qu'une instruction Algol est une entité définie récursivement, ce numéro syntaxique ne se réduit pas à un entier mais à une suite structurée d'entiers qui reflète la place de chaque instruction dans la structure d'ensemble du programme. Cette suite structurée sera représentée sous forme d'entiers séparés par des points. Par exemple, dans le programme suivant, les numéros syntaxiques sont écrits juste avant les instructions qu'ils désignent.

```
1  debut
    1.1 A:=B+C ;
    1.2 debut
        1.2.1 X:=Y+Z ;
        1.2.2 si I<J alors
            1.2.2.1 allera E
            sinon
                1.2.2.2 I:=J-K
        fin ;
    1.3 P(X,Y,Z)
fin
```

Sous forme d'arbre, on a la représentation suivante :



Dans cette numérotation syntaxique, les déclarations sont traitées comme des instructions et il en est de même des parties valeur et spécification dans les têtes de déclaration de procédure.

Pour la modification des programmes, il y a une seule règle de remplacement qui correspond à la commande :

REPLACER (<numéro syntaxique>)

et permet le remplacement de l'instruction désignée par le numéro syntaxique, paramètre de la commande remplacer, par la nouvelle instruction délimitée par la fin de commande REPLACER et la commande FREMPACER.

Cette règle simple de remplacement appelle plusieurs remarques :

- l'effacement d'une instruction est équivalent à son remplacement par une instruction vide
- le remplacement d'une instruction par plusieurs instructions ou l'insertion d'instruction entre deux instructions existantes conduit à créer une instruction composée formée par l'ensemble des instructions dans le cas d'un remplacement et par l'une ou l'autre des instructions existantes et de la nouvelle instruction dans le cas d'une insertion.

Lorsque dans une instruction nouvelle qui remplace une instruction existante on utilise à nouveau cette instruction ou des instructions intérieures à cette dernière, on peut utiliser les numéros syntaxiques de ces instructions écrits entre parenthèses (cela revient à dire que le texte source correspondant à une instruction existante n'est détruit que lorsque la nouvelle instruction de même numéro syntaxique a été complètement construite).

Si dans l'exemple précédent, on veut remplacer dans l'instruction de numéro syntaxique 1.2.2 uniquement l'expression booléenne on écrira :

```
REPLACER (1.2.2) ω
1.2.2 si I>J alors (1.2.2.1) sinon (1.2.2.2)
FREMPPLACER ω
```

Le numéro syntaxique de la nouvelle instruction est renvoyé automatiquement sur le terminal comme résultat de l'exécution de la commande REPLACER.

Dans le cas d'une instruction à insérer par exemple entre les instructions de numéros 2.4.2 et 2.4.3 on écrira :

```
REPLACER (2.4.2) ω
2.4.2 debut
      2.4.2.1 (2.4.2) ;
      2.4.2.2 Instruction à insérer
fin
FREMPPLACER ω
```

On aurait pu également utiliser REPLACER (2.4.3) et écrire l'instruction à insérer comme première instruction de l'instruction composée.

Ces remplacements d'instruction sont réalisés librement par le programmeur ou réclamés par le système en cas de détection d'erreur.

La commande

INST (<numéro syntaxique>) permet la sortie de l'instruction associée au numéro syntaxique défini comme paramètre.

4.7.2.2. Mécanisme d'inclusion des blocs.

La commande la plus importante du mode compilateur est la commande permettant l'inclusion d'un bloc déjà construit dans un bloc en cours de construction.

INCLUDE (<nom de file 1>,<nom de file 2>)

Les deux paramètres de cette commande désignent les files représentant le texte source et le texte objet du bloc à inclure. Il ne s'agit pas d'une inclusion physique (recopie) mais d'une inclusion logique (positionnement de pointeurs). La commande INCLUDE est en réalité considérée comme une instruction du texte source à laquelle on affecte un numéro syntaxique.

Il en est de même, des commandes EXTERNE et EQUIV ; EQUIV suivent immédiatement la commande INCLUDE et EXTERNE est la première instruction du bloc à inclure.

La commande INCLUDE permet la compilation d'un appel de sous-programme réentrant étant donné que le bloc à inclure peut être partagé par plusieurs utilisateurs simultanément.

4.7.2.3. Transformation des programmes mode conversationnel mode moniteur.

Cette opération est réalisée au moyen de la commande :

TRANSFORMER (<nom de file 1>,<nom de file 2>)

et permet la transformation d'un programme écrit en mode conversationnel en un programme écrit en mode moniteur.

Cette transformation est réalisée en deux étapes :

a) transformation du texte source écrit en mode conversationnel et désigné par le nom de file 1 premier paramètre de la commande TRANSFORMER, en un texte source écrit en mode moniteur et désigné comme paramètre de la commande FILE associée à ce travail.

b) compilation du texte source ainsi obtenu et production d'un texte objet désigné par le nom de file deuxième paramètre et exécutable en mode moniteur.

Dans la première étape, les instructions spécifiques du mode conversationnel sont éliminées du texte source après que les changements nécessaires aient été réalisés.

Dans cette transformation, les commandes d'inclusion peuvent être considérées comme des macros d'édition : elles sont remplacées par les textes source correspondants. Ce mécanisme de transformation est récursif étant donné que le texte source inséré peut à son tour contenir des commandes d'inclusion.

Les commandes EXTERNE et EQUIV peuvent être considérées comme liées implicitement aux commandes INCLURE correspondantes et permettent le remplacement correct des identificateurs de type externe.

Un dernier point concerne l'élimination des instructions STOP conditionnelles et des clés associées servant à la mise au point des programmes (cf. 4.8.2).

4.8. Mode interpréteur et commandes d'interprétation.

L'entrée dans ce mode s'effectue au moyen de la commande INTER et la sortie au moyen de la commande FINITER. Ce mode n'est valable que pour des programmes écrits en mode conversationnel.

Les commandes d'interprétation sont définies syntaxiquement de la façon suivante :

```

<commande d'interprétation> ::= INTER(ENTREE(<liste de noms de file>),
                                     SORTIE(<liste de noms de file>))|
INITIAL(<liste de paires nom-valeur>)|
VALEUR(<liste de variables>)|
<variable> := <expression>|
ADR|
INST(<numéro syntaxique>)|
CLE 1 := <constante booléenne>|
CLE 2 := <constante booléenne>|
CLE 3 := <constante booléenne>|
STOP|
REPRISE|
aller a <étiquette>|
aller a <numéro syntaxique>|
FINTER|
SINTER|
RINTER
<liste de paires nom-valeur> ::= <paire nom-valeur>|
                                <liste de paires nom-valeur>,<paire nom-valeur>
<paire nom-valeur> ::= <variable arithmétique> : <constante arithmétique>|
                       <variable booléenne> : <constante booléenne>|
                       <identificateur de tableau> : (<liste de constantes>)|
                       <identificateur de procédure> : (<nom de file>,<
                       <liste des valeurs des quantités externes>)

```

La commande INTER permet l'exécution du texte objet défini par la file opérande (de la commande FILE associée à ce travail) et utilisant les files données et résultats définis dans les paramètres ENTREE et SORTIE de cette commande.

Si l'on demande l'interprétation d'un bloc dans lequel on utilise des quantités non locales, les valeurs initiales de ces quantités doivent être définies dans une commande d'initialisation qui précède cette commande d'interprétation.

Dans ce mode d'exécution, l'interprétation peut être interrompue à n'importe quel moment et on peut ranger l'état de l'interpréteur au moment de l'interruption au moyen de la commande SINTER qui préserve les files associées à l'interprétation interrompue. Cet état peut être restauré au moyen de la commande RINTER.

L'interruption d'une interprétation peut être réalisée de 3 façons différentes :

- la fin du programme objet
- une instruction STOP écrite par le programmeur à l'intérieur du texte source
- une commande STOP issue du terminal sur lequel a lieu l'interprétation.

La reprise normale de l'interprétation après l'exécution d'une commande ou d'une instruction STOP est réalisée par la commande REPRISE.

4.8.1. Lecture et affectation dynamiques.

Après une interruption, des renseignements concernant l'interprétation en cours peuvent être obtenus au moyen des commandes suivantes :

VALEUR (<liste de variables>), sortie des valeurs courantes des variables désignées dans la liste.

<variable> := <expression>, affectation dynamique de la valeur courante d'une expression à une variable.

4.8.2. Contrôle dynamique de l'interprétation.

Il existe une classe de commandes d'interprétation qui permettent le contrôle de l'interprétation au niveau du programme source.

a) Les commandes :

ADR et
INST (<numéro syntaxique>)

permettent respectivement d'obtenir le numéro syntaxique de l'instruction Algol en cours d'exécution et le texte de l'instruction désignée par son numéro syntaxique.

b) Les commandes :

CLE 1 := <constante booléenne>
CLE 2 := <constante booléenne>
CLE 4 := <constante booléenne>

permettent le positionnement d'indicateurs booléens associés aux instructions STOP conditionnelles 'STOP1', 'STOP2', '-----', 'STOP7'.

Ces instructions STOP conditionnelles peuvent être écrites à l'intérieur du programme source comme des instructions Algol ordinaires, c'est-à-dire partout où cela est syntaxiquement possible. Elles sont écrites entre apostrophes afin de permettre une élimination plus facile dans la transformation d'un programme source écrit en mode conversationnel en un programme source écrit en mode moniteur.

L'instruction 'STOP i' ($1 < i \leq 7$) est interprétée de la façon suivante :

A tout instant la valeur courante de k est définie de la façon suivante :

```
k := (si CLE1 alors 1 sinon 0) +  
      (si CLE2 alors 2 sinon 0) +  
      (si CLE4 alors 4 sinon 0)
```

Si $i = k$ alors l'instruction STOP i est effective sinon elle est équivalente à une instruction vide.

c) Les commandes :

```
aller a <étiquette>  
aller a <numéro syntaxique>
```

permettent le transfert du contrôle d'interprétation au point du programme défini par l'étiquette ou le numéro syntaxique. Dans le cas où l'étiquette ou le numéro syntaxique ne sont pas valides, il y a renvoi d'un message d'erreur.

4.8.3. Mode calculateur de bureau.

Après la commande NOM on peut rentrer directement dans le mode interpréteur en utilisant la commande INTER sans paramètres et sans définir préalablement une description de travail au moyen d'une commande FILE.

On ne peut alors utiliser que les deux commandes

```
<variable> := <expression>  
VALEUR (<liste de variables>)
```

qui définissent dans ce contexte particulier le mode d'interprétation en calculateur de bureau.

C O N C L U S I O N G E N E R A L E

Dans le développement des langages de programmation et des techniques de compilation et d'interprétation qui permettent leur mise en oeuvre sur les machines, on peut discerner quelques tendances d'importance fondamentale :

- du côté de la définition des langages, les techniques de macro-définition et de macroévaluation qui permettront aux programmeurs à partir de quelques notions de base de définir les structures syntaxiques et les significations associées en fonction de leur besoin et de leur expérience.

- du côté de la mise en oeuvre des méthodes, les techniques de microprogrammation qui permettront la conception et la réalisation de systèmes efficaces. Il est clair en effet que pour tous les langages évolués caractérisés par leurs propriétés dynamiques, le langage machine ne peut plus jouer le rôle de langage objet comme cela fut le cas jusqu'à la mise en oeuvre des compilateurs Algol. La microprogrammation permet précisément à la fois une grande souplesse dans la définition des langages objet intermédiaires et une réelle efficacité d'exécution.

- il faudra enfin faire disparaître les différences souvent artificielles qui subsistent entre des notions similaires comme celles d'expression et d'instruction, de programme et de donnée, de langage de commande et de langage de programmation.

B I B L I O G R A P H I E

- [1] Sheridan, P.B., The Arithmetic Translator Compiler of the IBM Fortran Automatic Coding System, *Comm. ACM* 2, 2 (Février 1959), 9-21.
- [2] Chomsky, N., On certain Formal Properties of Grammars, *Information and Control*, 2 (1959), 137-167.
- [3] Backus, J.W., The Syntax and Semantics of the Proposed International Algebraic Language of the Zürich ACM-GAMA Conference, *Information Processing, Proceedings of the International Conference on Information Processing, UNESCO, Paris, 1960*, 125-132.
- [4] Perlis, A.J. et Smith, J.W., A Mathematical Language Compiler, *Journal of the Franklin Institute, Philadelphia*, (Avril 1957), 87-102.
- [5] Arden, B. et Graham, R., On GAT and the Construction of Translators, *Comm. ACM* 2, 8 (Aout 1959), 24-26.
- [6] Taylor, W., Turner, L. et Waychoff, R., A syntactical chart of Algol 60, *Comm. ACM* 4, 9 (Septembre 1961), 393.
- [7] COBOL - 1961, Revised Specifications for a Common Business Oriented Language, U.S. Government Printing Office, 1961.
- [8] IBM Operating System/360, PL/I : Language Specifications, IBM Corporation, SRL File N°. S360-29 Form C28-6571-3.
- [9] Brooker, R.A., et al., The Compiler compiler, *Annual Review in Automatic Programming*, Vol. 3, Pergamon Press, 229-275.
- [10] Floyd, R.W., Syntactic Analysis and Operator Precedence, *J.ACM* 10 (1963), 316-333.
- [11] Ershov, A.P., Hozhukhin, G.I. et Voloshin, U.M., Input Language for Automatic Programming Systems, *A.P.I.C. Studies in Data Processing*, N°. 3, Academic Press, 70 p.

- [12] Iverson, K.E., A Method of Syntax Specification, Comm. ACM 7, 10 (Octobre 1964), 588-589.
- [13] Church, A., The Calculi of Lambda Conversion, Annals of Mathematics Studies, N° 6, Princeton (1941), Princeton University Press.
- [14] Samelson, K., Functionals and Functional Transformations, Algol Bulletin, N° 20, (Juillet 1965), 27-28.
- [15] Mc Carthy, J., A Formal Description of a Subset of Algol, Formal Language Description Language for Computer Programming, Amsterdam (1966), North Holland Publishing Company, 1-12.
- [16] Landin, P.J., The Mechanical Evaluation of Expressions, Comp. J. 6, 4 (Janvier 1964), 308-320.
- [17] Van Wijngaarden, A., Recursive Definition of Syntax and Semantics, Formal Language Description Language for Computer Programming, Amsterdam (1966), North Holland Publishing Company, 13-24.
- [18] De Bakker, J.W., Formal Definition of Algorithmic Languages with an application to the definition of Algol 60, Report N° MR 74, Mathematisch Centrum, Amsterdam (1961).
- [19] Van Wijngaarden, A., Generalized Algol, Proceedings of the ICC Symposium on Symbolic Languages in Data Processing, New York and London (1962), Gordon and Breach, 409-419.
- [20] Perlis, A.J. et Iturriaga, R., An Extension to Algol for Manipulating Formulae, Comm. ACM 7, 2 (Février 1964), 127.
- [21] Cohen J. et Nguyen Huu Dung, Définition de procédures LISP en Algol, Exemple d'utilisation, R.F.T.I., Chiffres, Vol. 8., N° 4, (1965), 271-293.
- [22] Lawson, H.W., PL/I List Processing, IBM Technical Report.
- [23] Van Wijngaarden, A., The SC Proposal for Algol X, Working Document, W2, IFIP/WG2.1, Warsaw.
- [24] Abrahams, P.W. et Weissman, C., The Lisp 2 Programming Language and System, FJCC 1966 Proceedings.
- [25] Strachey, C., A General Purpose Macrogenerator, The Computer Journal, Vol. 8., N° 3 (Octobre 1965), 225-241.

- [26] Rutishauser, H., Automatische Rechenplanfertigung bei programmgesteuerten Rechenmaschinen, Verlag-Birkhäuser, Basel (1952).
- [27] Backus, J.W., et al., The Fortran Automatic Coding System, WJCC 1957 Proceedings, 188-198.
- [28] Bauer, F.L. et Samelson, K.S., Verfahren zur automatischen Verarbeitung von Rodierten Daten und Rechenmaschine zur Ausübung des Verfahrens, Deutsche Patentauslegeschrift 1094019, 30, 3 (1957).
- [29] Dijkstra, E.W., Recursive Programming, Numerische Mathematik 2 (1960), 312-318.
- [30] Leroy, H., A Proposal for Macro-Facilities in Algol, Algol Bulletin, N°. 22 (Février 1966), 15-26.
- [31] Hoare, C.A.R., Record Handling, A series of lectures delivered at the NATO Summer School on Programming Languages, Villard de Lans, France (Septembre 1966).
- [32] Galler, B.A. et Perlis, A.J., A Proposal for Definitions in Algol, ACM Proceedings of the 21st Conference, à paraître.
- [33] Leavenworth, B.M., Syntax Macros and Extended Translation, Comm. ACM 9, 11 (Novembre 1966), 790-793.
- [34] Johansen, P., Construction of Recognition Devices for Regular Languages from their BNF Definition, BIT 6 (1966), 294-309.
- [35] Moore, E.F., Sequential Machines, Selected Papers, Addison-Wesley Publishing Co. (1964).
- [36] Grau, A.A., Recursive Processes and Algol Translation, Comm. ACM 4, 1 (Janvier 1961), 10-15.
- [37] Bauer, F.L. et Samelson, K., Sequential Formula Translation, Comm. ACM 3, 2 (Février 1960), 76-83.
- [38] Werner, G., Etude de la syntaxe d'Algol, Application à la compilation, Thèse présentée à la Faculté des Sciences de Grenoble (Juin 1964).
- [39] Colmerauer, A., Recherche d'erreurs syntaxiques, Note technique IMAG, Grenoble (Octobre 1964).

- [40] Auroux, A. et Bellino, J., Système en Temps Partagé 1401/7044 en mode moniteur et mode conversationnel, Colloque AFIRO sur le traitement à distance et l'utilisation des calculateurs en temps réel et en temps partagé, Grenoble (31 Mai - 3 Juin 1966), à paraître chez Dunod, Paris.
- [41] Keller, J.M. et al., Remote Computing, an Experimental System, Part 2 : Internal Design, SJCC 1964 Proceedings, 426-443.
- [42] Lock, K., Structuring Programs for Multiprogram Time-Sharing On-Line Applications FJCC (1965), Vol. 27, Part 1, 457-472.
- [43] Auroux, A. Bellino, J. et Bolliet, L., Diamag : A Multi-Access System for On-Line Algol Programming, à paraître dans SJCC 1967 Proceedings.