

Contribution à l'étude des mécanismes de traduction des langages de programmation : application au traitement des structures définies dynamiquement

Laurent Trilling

▶ **To cite this version:**

Laurent Trilling. Contribution à l'étude des mécanismes de traduction des langages de programmation : application au traitement des structures définies dynamiquement. Génie logiciel [cs.SE]. Université Joseph-Fourier - Grenoble I, 1967. Français. NNT : . tel-00280779

HAL Id: tel-00280779

https://theses.hal.science/tel-00280779

Submitted on 19 May 2008

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

T H E S E S

présentées
à la Faculté des Sciences de l'Université de Grenoble
pour obtenir le grade de

Docteur - Ingénieur

par

Laurent TRILLING

Ingénieur IMAG. Licencié es Sciences

Première Thèse :

CONTRIBUTION A L'ETUDE DES MECANISMES DE TRADUCTION
DES LANGAGES DE PROGRAMMATION. APPLICATION AU TRAITEMENT
DES STRUCTURES DEFINIES DYNAMIQUEMENT

Deuxième Thèse :

PROPOSITIONS DONNEES PAR LA FACULTE

Thèses soutenues le 1er Décembre 1967, devant la Commission d'examen

Monsieur	VAUQUOIS	Président
Messieurs	BACCHUS	
	BOLLIET	Examinateurs
	BOUSSARD	

n° d'ordre

T H E S E S

présentées
à la Faculté des Sciences de l'Université de Grenoble
pour obtenir le grade de

Docteur - Ingénieur

par

Laurent TRILLING

Ingénieur IMAG. Licencié es Sciences

Première Thèse :

CONTRIBUTION A L'ETUDE DES MECANISMES DE TRADUCTION DES LANGAGES DE PROGRAMMATION. APPLICATION AU TRAITEMENT DES STRUCTURES DEFINIES DYNAMIQUEMENT

Deuxième Thèse :

PROPOSITIONS DONNEES PAR LA FACULTE

Thèses soutenues le 1er Décembre 1967, devant la Commission d'examen

Monsieur	VAUQUOIS	Président
Messieurs	BACCHUS	
	BOLLIET	Examinateurs
	BOUSSARD	

FACULTE DES SCIENCES

LISTE DES PROFESSEURS

DOYENS HONORAIRES :

M. MORET

M. WEIL

DOYEN :

M. BONNIER E.

PROFESSEURS TITULAIRES :

MM.	NEEL Louis	Chaire de Physique Expérimentale
	HEILMANN René	Chaire de Chimie
	KRAVTCHENKO Julien	Chaire de Mécanique Rationnelle
	CHABAUTY Claude	Chaire de calcul différentiel et intégral
	BENOIT Jean	Chaire de Radioélectricité
	CHENE Marcel	Chaire de Chimie Papetière
	WEIL Louis	Chaire de Thermodynamique
	FELICI Noël	Chaire d'Electrostatique
	KUNTZMANN Jean	Chaire de Mathématiques Appliquées
	BARBIER Reynold	Chaire de Géologie Appliquée
	SANTON Lucien	Chaire de Mécanique des Fluides
	OZENDA Paul	Chaire de Botanique
	FALLOT Maurice	Chaire de Physique Industrielle
	KOSZUL Jean-Louis	Chaire de Mathématiques M.P.C.
	GALVANI O.	Mathématiques
	MOUSSA André	Chaire de Chimie Nucléaire
	TRAYNARD Philippe	Chaire de Chimie Générale

	SOUTIF Michel	Chaire de Physique Générale
	CRAYA Antoine	Chaire d'Hydrodynamique
	REULOS R.	Théorie des Champs
	BESSON Jean	Chaire de Chimie
	AYANT Yves	Physique Approfondie
	GALLISSOT	Mathématiques
Melle	LUTZ Elisabeth	Mathématiques
MM.	BLAMBERT Maurice	Chaire de Mathématiques
	BOUCHEZ Robert	Physique Nucléaire
	LLIBOUTRY Louis	Géophysique
	MICHEL Robert	Chaire de Minéralogie et Pétrographie
	BONNIER Etienne	Chaire d'Electrochimie et d'Electrométallurgie
	DESSAUX Georges	Chaire de Physiologie Animale
	PILLET E.	Chaire de Physique Industrielle et Electrotechnique
	YOCCOZ Jean	Chaire de Physique Nucléaire Théorique
	DEBELMAS Jacques	Chaire de Géologie Générale
	GERBER R.	Mathématiques
	PAUTHENET R.	Electrotechnique
	VAUQUOIS B.	Chaire de Calcul Electronique
	BARJON R.	Physique Nucléaire
	BARBIER Jean-Claude	Chaire de Physique
	SILBER R.	Mécanique des Fluides
	BUYLE-BODIN Maurice	Chaire d'Electronique
	DREYFUS B.	Thermodynamique
	KLEIN J.	Mathématiques
	VAILLANT F.	Zoologie et Hydrobiologie
	ARNOUD Paul	Chaire de Chimie M.P.C.
	SENGEL P.	Chaire de Zoologie
	BARNOUD F.	Chaire de Biosynthèse de la Cellulose
	BRISSONNEAU P.	Physique
	GAGNAIRE Didier	Chaire de Chimie Physique

Mme	KOFLER L.	Botanique
MM.	DEGRANGE Charles	Zoologie
	PEBAY-PEROULA J.C.	Physique
	RASSAT A.	Chaire de Chimie Systématique

PROFESSEURS SANS CHAIRE :

MM.	GIDON P.	Géologie et Minéralogie
	GIRAUD P.	Géologie
	PERRET R.	Servomécanismes
Mme	BARBIER M.J.	Electrochimie
Mme	SOUTIF J.	Physique
MM.	COHEN J.	Electrotechnique
	DEPASSEL R.	Mécanique des Fluides
	GASTINEL N.	Mathématiques Appliquées
	ANGLES-d'AURIAC P.	Mécanique des Fluides
	DUCROS P.	Minéralogie et Cristallographie
	GLENAT R.	Chimie
	LACAZE A.	Thermodynamique
	BARRA J.	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

MAITRES DE CONFERENCES :

MM.	DODU J.	Mécanique des Fluides
	LANCIA Roland	Physique Automatique
Mme	KAHANE J.	Physique
MM.	DEPORTES C.	Chimie
Mme	BOUCHE L.	Mathématiques
MM.	SARROT-RAYNAUD J.	Géologie Propédeutique
Mme	BONNIER M.J.	Chimie
MM.	KAHANE A.	Physique Générale
	DOLIQUE J.M.	Electronique
	BRIERE G.	Physique M.P.C.
	DESPRE P.	Chimie S.P.C.N.
	LAJZEROWICZ J.	Physique M.P.C.
	VALENTIN P.	Physique M.P.C.
	BERTRANDIAS J.P.	Mathématiques Appliquées T.M.P.
	LAURENT P.	Mathématiques Appliquées T.M.P.
	CAUBET J.P.	Mathématiques Pures
	PAYAN J.J.	Mathématiques
Mme	BERTRANDIAS F.	Mathématiques Pures M.P.C.
MM.	LONGEQUEUE J.P.	Physique
	NIVAT M.	Mathématiques Appliquées
	SOHM J.C.	Electrochimie
	ZADWORNY F.	Electronique
	DURAND F.	Chimie Physique
	CARLIER 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
	BLOCH D.	Electrotechnique
	BRUGEL L.	I.U.T.
	SIBILLE R.	I.U.T.

Que Monsieur le Professeur KUNTZMANN, Directeur de l'Institut de Mathématiques Appliquées de GRENOBLE, reçoive mes remerciements les plus sincères pour m'avoir donné la possibilité de préparer et de présenter cette thèse.

Je tiens à exprimer ma profonde reconnaissance à

Monsieur le Professeur VAUQUOIS, Directeur du Centre d'Etudes pour la Traduction Automatique, qui a bien voulu me faire l'honneur de présider le jury,

Monsieur le Professeur BACCHUS, Directeur du Laboratoire de Calcul de l'Université de LILLE, qui a bien voulu faire partie du jury,

Monsieur BOLLIET, Maître de Conférences à l'Institut Universitaire de Technologie de GRENOBLE, qui s'est intéressé à cette étude,

Monsieur BOUSSARD, Maître de Conférences à l'Institut Polytechnique, qui m'a guidé et considérablement soutenu dans mes travaux,

Monsieur COHEN, Professeur Assistant au Massachussets Institute of Technology, qui fut un conseiller ardent et avisé et qui a collaboré en partie à ce travail.

Je tiens à remercier également toutes les personnes du Laboratoire, qui par leurs entretiens ou leurs études, ont contribué à la réalisation de cette thèse, en particulier Alain COLMERAUER qui fut un attentif compagnon de travail et Philippe JORRAND.

Je ne manquerai pas de mentionner l'aide que m'ont apportée le service de perforation et les opérateurs du calculateur.

Enfin, je remercie vivement pour sa célérité et son effécacité le service du Secrétariat qui a participé à la réalisation matérielle de cet ouvrage.

A ma Mère,

A mon Père,

TABLE DES MATIERES

I INTRODUCTION.

II GENERATION .

1 - Un essai de description du langage objet.

2 - Un langage de génération.

1 - Description.

2 - Inclusion dans ALGOL 60.

3 - Exemples d'applications.

1 - Expressions arithmétiques.

2 - Schéma simple pour la définition de nouveaux opérateurs.

4 - Environnement.

1 - Relation avec une phase AMC.

2 - Relation avec l'analyse.

5 - Conclusion.

III COMPILATION D'UN LANGAGE DERIVE D'ALGOL X.

1 - Description du langage source.

2 - Description du langage objet.

3 - Principaux problèmes.

1 - Déclarations.

2 - Variables libres.

3 - Expressions

4 - Instructions - Générateur d'enregistrement.

5 - Relations.

4 - Gestion de la mémoire.

1 - Organisation générale.

2 - Récupération.

3 - Retassement.

5 - Réalisations pratiques.

IV CONCLUSION.

-I- INTRODUCTION

La nécessité de langages de programmation indépendants d'un calculateur donné est apparue très tôt depuis les débuts d'utilisation des ordinateurs.
Un grand nombre de ces langages ou "dialectes"(c'est à dire différent de peu de certains de leurs prédécesseurs) ont été créés et ainsi dès 1962, un rapport [ISO] sur ce sujet comptait plus de trois cents compilateurs... La construction d'un compilateur se révélant longue et ardue, de nom - breuses méthodes ont été mises en oeuvre pour en limiter le nombre ou en faciliter l'élaboration. Parmi celles-ci, on peut distinguer trois courants principaux :

1) Un langage algorithmique universel

ALGOL et FORTRAN en sont des exemples dans le domaine scientifique et COBOL dans le domaine commercial: ils sont très largement répandus. Il reste qu'un langage de cette sorte n'est pas adapté à tous les problèmes et qu'il ne peut rendre compte, au moment où il est conçu, de progrès technologiques et des nouveaux domaines d'application des ordinateurs. CPL, ALGOL X, PL/1,...etc, sont les représentants les plus récents de cette tendance.

2) Un langage machine universel.

En 1958 l'organisation SHARE proposa un langage intermédiaire universel appelé UNCOL.
UNCOL devrait être tel que tout programme écrit dans un langage à tout niveau puisse être transformé en un programme UNCOL équivalent et tout programme UNCOL traduit en tout langage machine. JOVIAL [Sh] fut compilé dans cet esprit. Notons simplement qu'il est difficile pour un tel langage de tenir compte de la diversité des codes machines.

3) Le compilateur paramétrisé.

On montre de plus en plus d'intérêt à des "Systèmes" de compilation admettant comme données non seulement un programme écrit dans un certain langage, mais aussi une description de ce langage objet (celui-ci pouvant être ou non le langage de la machine sur laquelle le programme source doit être exécuté). Les problèmes qui se posent sont évidemment ceux des formalismes à adopter pour représenter des descriptions et ceux relatifs à la mise en oeuvre de leur exploitation.

Nous nous intéressons ici à cette dernière proposition.
Le travail que nous présentons se divise en deux parties.
La première est essentiellement consacrée :

- d'une part, à l'étude d'un essai de description du langage objet dans un tel système.

- d'autre part, à l'étude d'un langage destiné à travailler sur une structure arborescente (dérivée directement de l'analyse syntaxique ou non) déduite de la chaîne en langage source. Un programme écrit dans un tel langage fournit la traduction en langage objet de cette chaîne.

La seconde partie concerne la résolution d'un problème concret : celui de la compilation d'un langage inspiré en partie d'ALGOL X [VW1, VW2]. Les emprunts faits à ce langage sont ceux ayant trait à la création de nouveaux types et aux variables de types "référence à". Ces concepts donnent au programmeur la possibilité d'utiliser des variables dont il définit lui-même la nature en fonction de certains modes de base (entier, booléen,...etc) ou de types déjà définis; il peut aussi déclarer des variables dont la valeur est un "nom", par exemple, celui d'une autre variable.

On remarque que ces extensions font d'ALGOL X un langage mieux adapté qu'ALGOL 60 aux traitements de problèmes appartenant à des domaines divers : en effet, ces nouvelles notions permettront de traiter plus commodément des structures de natures très variées. Ainsi, on pourra utiliser des nouvelles variables par représentation par exemple, des éléments de matrice creuse ou les noeuds d'un graphe.

Le langage objet choisi est un langage d'interprèteur et nous utilisons pour effectuer la compilation le langage proposé dans la première partie.
Ce dernier facilite grandement l'écriture du compilateur, mais il reste évidemment un outil.
En fait, nous verrons que les difficultés principales ressortent du choix de la représentation machine des nouvelles variables, de l'organisation générale de la mémoire et de la gestion de celle-ci à l'exécution.

-II- GENERATION

-GENERALITES-

Un assez grand nombre de systèmes de compilation ont été mis sur pied ou projetés. Un récent rapport sur ce sujet [ShF] en compte environ douze. Indirectement, ceci laisse supposer un développement important des langages spécialisés.
On trouvera en [Co] une description de quatre des principaux systèmes dus respectivement à IRONS [Ir] , BROOKER [Br], FELDMANN [Fl] et GARWICK [Gr] En [ShF] M. SHAW et J. FIERST donnent un aperçu très complet de ce qui a été fait à ce jour et présentent d'une façon succinte les particularités de chacun d'entre eux.

Nous nous bornerons ici à présenter deux parmi les plus importants de ces systèmes :
C G S, mis en oeuvre par Computer Associates [Was] et T M G/TROL du à C G S (Compiler Generator System).

Dans cette proposition, un compilateur est divisé en cinq phases (qui ne procèdent pas nécessairement à un examen complet de la chaîne) :

1 - Un analyseur syntaxique auquel est fournie une grammaire sous forme proche de la Forme Normale de Backus décrivant le langage source et qui convertit une chaîne (et certaines de ses sous-chaînes) écrite dans ce langage en un arbre conforme à la grammaire.

2 - Un générateur qui transforme cet arbre en macro-instructions à n-adresses, examinant un certain contexte avant de décider de la production.

3 - Une "optimiseur" qui réduit ces macro-instructions, reconnaît et élimine les calculs d'expressions communes, déplace les invariants dans une boucle et même affecte certaines quantités à des registres spéciaux (index).

4 - Ur "sélecteur" de code qui transforme les macros en "syllabes" de code-machine et effectue une gestion optimale des registres.

5 - Un assembleur qui met en forme ces syllabes de code de façon compatible avec le mode adopté : symbolique, absolu, translatable...etc.

Cette dernière phase est dépendante de la machine utilisée. Les quatre autres sont écrites dans des langages spécialisés.

L'analyseur est "descendant" et, à l'aide d'un langage dit BNF (Backus Normal Form) on peut effectuer des tâches qui ne sont pas purement de reconnaissance syntaxique : par exemple, passer le contrôle à la phase suivante ou ajouter certaines informations à l'arbre en cours de construction.

La seconde phase est écrite en langages G S L (Generation Strategy Language) : elle est composée d'une série de règles identifiant le type de noeuds d'arbres traités et indiquant les actions à exécuter lorsqu'ils sont rencontrés. Une instruction G S L débute par un prédicat de la forme :

I F < nom du noeud > AND σ_1 (t_1)..:AND σ_i (ti...AND σ_n (t_n)

dans lequel les σ_i sont des expressions booléennes satisfaites ou non aux noeuds de l'arbre spécifiés par t_i. Ce prédicat est suivi d'un court programme indiquant les "actions" associées au noeud considéré : examiner un autre noeud, déclencher une sortie..etc.

Une particularité de ce système est que le même noeud peut être examiné plusieurs fois; à chaque fois, des actions différentes lui sont associées.

La phase de génération de code objet est réalisée à l'aide du langage M D L (Macro Description Language) : essentiellement, c'est son sous-ensemble C S L (Code Sélection Langage) qui permet d'exprimer les codes-objets et gérer les registres spéciaux du calculateur.
Ce système est opérationnel sur les calculateurs IBM 7090, Burroughs D-285 et C D C 1604.
Il a été utilisé pour compiler les langages JOVIAL et CXA (un "dialecte" d'ALGOL).
On note à ce propos qu'une expérience intéressante a été faite pour transplanter automatiquement ce système d'une machine sur une autre disposant :

- de compilateurs écrits dans un langage Lo et transformant des programmes écrits dans les langages employés par le système en des programmes équivalents en Lo.

- d'un compilateur pour Lo sur l'ordinateur IBM 7090.

Il a été possible, en utilisant un procédé d'autocompilation (bootstrapping) d'implanter ce système sur la machine CDC 1604. On a ainsi pu obtenir le compilateur du langage CXA sur cette machine. Cette approche est très séduisante, mais les auteurs eux-mêmes font remarquer qu'elle est loin d'être aisée à mettre en oeuvre.

En conclusion, ce système est un des plus complets que nous connaissions : notons simplement que les "descriptions" des langages sources et objets sont en fait des programmes. En particulier la stratégie adoptée pour générer le langage objet peut s'exprimer d'une façon fort complexe. Enfin, peu d'indications sont données sur les caractéristiques des langages qui doivent faciliter la tâche du programmeur-compiliste.

T M G / T R O L.

Mc CLURE a construit, à la Texas Instruments, un système appelé T M G que KNUTH a utilisé comme base pour son langage de génération dénommé T R O L.. T M G a été employé récemment pour traiter un sous-ensemble de PL/I.
KNUTH utilise T R O L et son interpréteur associé T R O L L pour illustrer des techniques de construction de compilateurs [non encore publié].

Un programme est composé (en plus des déclarations) d'un ensemble de lignes, chacune d'elle contenant une spécification syntaxique et une spécification sémantique, celle-ci devant être examinée si la première est satisfaite. Elles sont précédées d'un champ contenant une étiquette.
La partie syntaxique comporte une série d'actions, chacune d'elles étant associée à deux étiquettes de sortie. Une action peut être elle-même une étiquette (impliquant l'évaluation de la ligne associée) ou un groupe de tests ou d'actions tels que tester la présence d'une lettre ou d'un bit, repositionner le pointeur de la chaîne d'entrée ou encore passer le contrôle à la partie sémantique. Après chaque action une valeur est disponible, vrai, faux ou échec; la partie syntaxique interprète cette valeur de la façon suivante :

- au début d'une ligne, à la fin d'une action, s'il ne reste plus d'actions, exécuter la partie sémantique.

- si une action reste, l'exécuter : ceci fournit une valeur. Si deux étiquettes sont associées à l'action, on poursuit dans le cas où la valeur fournie est vrai, on se renvoie à la première si la valeur est faux, à la seconde dans le cas d'un échec : si les étiquettes correspondantes ne sont pas présentes, le pointeur de la chaîne d'entrée reprend la valeur qu'il avait à l'entrée dans la ligne et le contrôle revient à l'appel de la ligne : la valeur reste inchangée.

En fait, les tests retournent des valeurs booléennes, les autres actions (manipulation de pointeurs ou de tables) retournent généralement la valeur vrai. La seule action retournant échec est FALL : elle permet d'écourter les recherches quand une erreur a été définitivement détectée.
En ce qui concerne la partie sémantique; on dispose pour l'exprimer d'un assortiment usuel d'instructions, affectation, instruction con itionnelle, entrées-sorties...et transfert de contrôle soit à une partie sémantique soit à une partie syntaxique.
On voit que dans ce système la syntaxe est uniquement prise comme guide d'écriture pour l'écriture d'un compilateur. Tous les outils sont fournis pour construire un analyseur syntaxique, mais le paramètre grammaire (sous forme normale de Backus) n'est pas considéré.

Nous abordons quant à nous, le problème de la construction d'un système de compilation de la façon suivante.
Etant donnée une chaîne de symboles, appelons "reconnaissance" un processus qui, d'une part vérifie la correction de cette chaîne par rapport à certaines conventions d'écritures, et d'autre part est capable d'y distinguer des sous-chaînes auxquelles l'auteur attache une signification particulière.
Dans les mêmes conditions, appelons production un procesuus fournissant, à partir de telles sous-chaînes, une autre chaîne de symboles que nous nommerons "transformée" de la chaîne initiale par le couple reconnaissance-production ainsi défini.
La plupart des compilateurs actuellement opérationnels [Bou , le P, Na] sont réalisés à l'aide d'une succession de tels couples reconnaissance-production, où la chaîne transformée issue d'un couple constitue la chaîne initiale de son sucesseur éventuel. On constate que très souvent, le nombre de ces couples est 2. Dans ce cas nous nommerons le premier analyse morphologique-codification (AMC) le second analyse syntaxique-génération (ASG).

Les rôles de la phase AMC sont essentiellementles suivants :

- obtenir de la chaîne initiale, constituée par le programme source, une transformée acceptable par la phase suivante, que celle-ci soit du type ASG ou de nouveau du type AMC.

- détecter, au choix de l'utilisateur, les erreurs que celui-ci juge importantes et qui se trouveraient hors de portée de la phase suivante.

Un langage adapté à ce travail a été développé au Laboratoire de Calcul de l'Université de Grenoble par P. Jorrand [Jor] .
Un programme dans ce langage admet comme paramètre de la partie reconnaissance la description d'un automate d'états finis à l'aide de sa table d'états. Un compilateur pour ce langage est opérationnel sur IBM 7044.

Nous nous intéressons ici à la seconde phase. Ses tâches principales sont :

- transformer la structure linéaire de la chaîne fournie par la phase précédente en une nouvelle structure, en général parenthésée : ceci constitue la partie reconnaissance de cette phase, déterminant en particulier si la chaîne d'entrée est syntaxiquement correcte et détectant éventuellement la source de l'erreur.

- obtenir de la chaîne ainsi restructuréeune transformée de nouveau linéaire et le mieux adaptée possible à une exécution sur machine (ceci n'interdit pas de considérer comme langages objets des langages évolués).
La forme normale de Backus (FNB) est un formalisme bien connu de description de langages et il est couramment adopté pour abordé pratiquement la partie reconnaissance de cette phase. Une description du langage source est donc introduite sous cette forme comme paramètre principal de la phase ASG.

Ceci permet d'obtenir un arbre syntaxique associé à la chaîne en langage source fournie.

En II.1., nous rappelons un moyen, que nous avons proposé en Co T_1 pour introduire la description du langage-objet (langage-machine ou d'assemblage) comme deuxième paramètre. Signalons dès maintenant que le formalisme proposé reste incomplet : nous indiquons quelques unes des difficultés rencontrées pour l'étendre.

En II.2, nous présentons un langage destiné à exprimer la partie production de la phase ASG. Un programme dans ce langage exploite :

- soit directement l'arbre syntaxique

- soit un arbre déduit de l'arbre syntaxique, plus condensé en général et se prêtant mieux à la génération (arbre de dépendance). Cette transformation peut s'effectuer à l'aide d'un mécanisme décrit par J. Cohen [Co] : des règles sont associées à certaines métavariables de la grammaire et permettent d'indiquer la transformation à accomplir sur l'arbre syntaxique pour obtenir ce nouvel arbre.
L'ensemble de ces règles forme d'ailleurs un autre paramètre possible.

Une inclusion de ce langage dans ALGOL 60 à été a été en partie réalisée et nous décrivons brièvement les principes qui nous ont guidés pour ce faire.
Deux exemples d'application , l'un consacré aux expressions arithmétiques, l'autre à la définition de nouveaux opérateurs dans un langage de programmation aideront ensuite à comprendre comment fonctionne un programme écrit dans ce langage de génération.

Enfin, les caractéristiques souhaitables de l'analyseur et les conditions à satisfaire pour une bonne liaison des phases AMC et ASG sont examinées et discutées.

II - 1 - Un essai de description du langage objet.

Nous rappelons ici brièvement le fonctionnement d'un transducteur décrit plus complètement en [Co, Col] . Un programme ALGOL 60 simulant ce transducteur est présenté ainsi qu'un exemple d'application à un répertoire d'instructions.

On fournit à ce transducteur :

- une chaîne d'entrée dans laquelle les opérateurs sont distingués et placés devant les opérandes associés (forme préfixée). Le nombre d'opérandes d'un opérateur étant fixé, on peut représenter une chaîne sous cette forme par un arbre. On supposera un tel arbre obtenu à partir d'un arbre syntaxique par exemple à l'aide du mécanisme décrit en [Co] [cf.II.2.3.1] .
- une description des instructions du langage objet.

Il produit une séquence d'instructions représentant une traduction de la chaîne d'entrée en langage objet.

Description de la chaîne d'entrée.

Syntaxe :

<chaîne> ::= <opérateur> <suites d'opérandes>|<atome>

<suite d'opérandes> ::= <chaîne> | <chaîne> <suites d'opérandes>

Sémantique :

<opérateur> désigne les opérateurs mis en évidence lors de la transformation de l'arbre syntaxique.

<atome> désigne un opérande élémentaire.

Exemple :

:= A + * B C / D E

Description des règles de réécriture associées aux instructions objet :

Syntaxe :

<règle> ::= <partie gauche> $\rightarrow$ <partie droite>

<partie gauche> ::= <expression>

<partie droite> ::= λ | <suite d'expressions>

<suite d'expressions> ::= <expression> |
<expression> et <suite d'expressions>

<expression> ::= <registre> = <terme> | <terme>

<terme> ::= <opérateur> <suite d'opérandes 1>

<suite d'opérandes 1> ::= <opérande 1> <suite d'opérandes 1>

<opérande 1> ::= T | M | ω_i

Sémantique :

ω_i désigne formellement une chaîne du type défini ci-dessus,

M désigne un atome d'une telle chaîne

T désigne une mémoire de travail

<registre> représente les registres sur lesquels opèrent les instructions

$\rightarrow$ veut dire "se réécrit en"

et est un opérateur de concaténation pour les expressions

λ représente l'expression vide

= est un symbole permettant uniquement une meilleure lisibilité de la règle.

A chaque instruction correspond une règle de ce type.

Exemple :

A l'instruction :

ADD M

on attache la règle :

Acc = + Mω_o $\rightarrow$ Acc = ω_o

que l'on interprète ainsi : "pour que l'instruction soit utilisée, il faut que l'un des opérandes (ici ω_o) soit rangé dans le registre Acc; le but de l'opération est d'additionner M à ω_o".

Fonctionnement du transducteur.

Définissons tout d'abord l'applicabilité d'une partie gauche β sur une séquence de symboles α : β est applicable à α si à chaque élément de β on peut faire correspondre dans le même ordre de succession une partie de α

- soit égale à l'élément de β
- soit désignée par l'élément de β

Exemple : la partie gauche

$$Acc = + M\omega_o$$

est applicable à : Acc = + A * B C

Etant donnée une expression fournie au transducteur, celui-ci recherche parmi les règles de réécriture celle (S) dont la partie gauche est applicable à cette expression. Dans un cas favorable, une double tâche est dévalue au transducteur :

a) Transformation de l'expression donnée

La nouvelle expression considérée est obtenue en faisant prendre aux symboles ω_i et M de la partie droite de la règle utilisée leurs valeurs trouvées lors du calcul d'applicabilité.

Exemple : l'emploi de la règle :

$$Acc = + M\omega_o \rightarrow A_{cc} = \omega_o$$

transforme l'expression Acc = + A * B C en Acc = * B C

Si l'expression transformée est une suite d'expressions concaténées par le symbole et, celles-ci seront considérées de gauche à droite.

b) Production d'une instruction

Si une instruction est associée à la règle utilisée, la valeur de ses constituants sont celles prises dans le calcul d'applicabilité par les éléments précisés comme constituants de l'instruction.

Exemple : l'instruction ADD M attachée à la règle de l'exemple précédent aboutit à la génération de l'instruction ADD A.

La première expression fournie au transducteur est la chaîne d'entrée. Elle est successivement réduite par application des règles de réécriture. La traduction en langage objet est terminée lorsque l'expression considérée est l'expression vide. Les instructions sont évidemment produites dans l'ordre inverse de celui de leur exécution.

Signalons que plusieurs traductions peuvent être obtenues si des règles différentes peuvent être appliquées à la réduction d'une même expression : ce transducteur est donc non-déterministe.

Extensions

Ce transducteur ne permet de traiter que des expressions relativement simples. Deux extensions ont été abordées, l'une concerne l'usage des mémoires de travail, l'autre l'optimisation du programme généré.

1. - On peut introduire les mémoires de travail à l'aide de règles de réécriture analogues à celles décrites précédemment. Aucune instruction ne leur correspond.

Nous avons utilisé des règles de deux types :

a) $R = op\ \omega_1\ \omega_2 \rightarrow R = op\ \omega_1,\ T\ \underline{et} := T\ \omega_2$

R et op désingnent respectivement un resgistre et un opérateur (binaire) quelconques.

L'opérande ω_2 est stocké dans une mémoire temporaire. Cette

règle permet d'aboutir lorsque l'expression ne peut être réduite du fait qu'un opérande n'est pas atomique.

b) $R = \omega_0 \rightarrow R = T$ et $:= T\ \omega_0$

Cette règle donne la possibilité de transférer le contenu d'un registre dans une mémoire temporaire. Elle est utilisée dans le cas où l'expression examinée ne peut être réduite parce que le registre est incompatible avec le terme de l'expression. Une telle transformation est toutefois "dangereuse" : son utilisation peut en effet aboutir à un traitement ultérieur de la même expression, d'où un "bouclage", comme nous le verrons dans l'exemple considéré plus loin.

2. - Le critère d'optimisation considéré pour le programme objet est sa rapidité d'exécution. Un coût est attribué à chaque instruction et le transducteur recherche parmi les solutions possibles celles dont le coût accumulé des instructions est minimal. Pour cette recherche, il est intéressant de remarquer que dans le cas où une expression est formée d'expressions concaténées, le programme objet minimal est composé des programmes objets minimaux correspondant à celles-ci.

Description du programme ALGOL simulant ce transducteur

On a utilisé l'inclusion de LISP en ALGOL 60 présentée en [Co N] En effet, les règles et les expressions sont aisément représentables sous forme de listes.

Il se compose principalement de deux procédures récursives :

- réduction totale (suite d'expression)

  qui isole les expressions dans une suite d'expressions.

- réduction (expression, coût) :

  qui a deux tâches :

a) déclencher le calcul d'applicabilité, c'est-à-dire : déterminer les règles qui peuvent s'appliquer à la réduction de l'ex-

pression fournie. Dans le cas où aucune règle du répertoire d'instructions n'est convenable, une règle introduisant des mémoires temporaires du premier type est employée : c'est seulement en cas d'insuccès de cette tentative qu'est appliquée une règle du second type.

b) - Comparer le coût minimal obtenu pour les réductions antérieures de l'expression à celui obtenu en produisant l'instruction correspondant à la règle appliquée. Si ce dernier est supérieur au coût minimal, cette voie de réduction est abandonnée.

Le calcul d'applicabilité est effectué par la procédure :

application (règle, expression).

Le rôle de celle-ci est de construire la transformée de l'expression si la règle s'y prête. Dans le cas où la règle est "dangereuse", l'expression est retenue dans une pile. Si ultérieurement, cette expression est de nouveau considérée, la transformation en est jugée impossible.

Exemple :

On présente partiellement dans les pages qui suivent le déroulement du traitement de la chaîne d'entrée :

F := (A+B) * C/D

A chaque règle de réduction correspond une liste dont les éléments pris de gauche à droite sont :

- la partie gauche de la règle
- la partie droite de la règle
- l'instruction associée à la règle
- le coût attribué à la règle
- un indicateur représenté par P ou NP suivant que, dans le cas où M représente une mémoire temporaire, il faille la créer ou la libérer.

Deux étapes sont décrites :

- la première est le début de la réduction de la chaîne donnée. Celle-ci est initialement transformée en :

  MQ = * + AB /CD

  qui nécessite l'emploi d'une règle "dangereuse". On en arrive à traiter ensuite l'expression :

  := T * +AB / CD

  qui conduira à traiter de nouveau l'expression :

  MQ = * + AB / CD

  Cette voie de réduction est alors abandonnée. C'est la règle associée à l'instruction "STO M" qui est alors essayée. Elle aboutit à la considération de la suite d'expressions :

  Acc = * + ABT et := T / CD

- la seconde étape concerne la réduction de la deuxième expression de cette suite d'expression. L'examen de la première produit le programme objet.

```
CLA B
ADD A
STO T1
LDQ T1
MULT TO
```

cle coût total étant égal à 19.

L'expression := T / CD est alors traduite en

```
CLA C
DIV D
STQ TO
```

ce qui donne un coût total de 29.

Une autre possibilité est ensuite essayée. Elle est abandonnée avantde parvenir à une réduction totale car son coût est trop élevé.

Le programme objet finalement obtenu est :

```
CLA  C
DIV  D
STQ  TO
CLA  B
ADD  A
STØ  T1
LDQ  T1
MULT TO
STO  F
```

On note que l'emploi d'une autre règle introduisant des mémoires temporaires, en l'espece :

$$R = op\ \omega_1\ \omega_2 \rightarrow R = op\ T\omega_2 \quad \underline{et} \quad := T\omega_1$$

aurait produit le programme objet plus intéressant :

```
CLA  B
ADD  A
STO  TO
CLA  C
DIV  D
MULT TO
STO  F
```

Remarque :

On trouve en [Wa S] un exemple d'optimisation (écrit en langage CSL cf. § II.0) de l'affectation grâce à l'utilisation de l'instruction

RAD M

Cette instruction a pour effet d'additionner le contenu de la mémoire M à celui de l'accumulateur Acc et de ranger le résultat en M. Ainsi l'expression :

A := A + B

est traduite en :

CLA B

RAD A

Les règles de réécriture correspondant à cette instruction et conduisant à l'optimisation décrite en [Wa S] seraient :

$:= M + M\omega_0 \rightarrow Acc = \omega_0$

$:= M + \omega_0 M \rightarrow Acc = \omega_0$

Conclusion

Ce transducteur présente, à notre avis, deux aspects intéressants :

- les instructions du langage objet sont explicitement décrites
- la recherche de toutes les solutions correctes et l'introduction d'un coût associé à chaque instruction permet de déteminer les solutions de coût minimal.

Il reste toutefois très incomplet pour permettre de traiter un répertoire entier d'instructions. Parmi les principales limitations, on peut citer :

- absence d'un dispositif interdisant l'utilisation ultérieure d'un registre.

En effet, il peut arriver que des suites d'expressions soient du type :

$$R_1 = \omega_1 \ \underline{et} \ R_2 = \omega_2$$

Si la réduction de l'expression ω_1 nécessite l'emploi du registre R_2, le programme objet correspondant sera erroné.

Dans ce cas, il faut donc prévoir de ranger le contenu du registre menacé dans une mémoire de travail.

- absence d'un moyen permettant de générer des règles de réécriture, par exemple à la déclaration d'une procédure.

- absence d'un mécanisme pour produire des instructions étiquetées.

Plusieurs voies sont possibles pour poursuivre cette étude :

- recherche de critères permettant, au vue des règles définissant les instructions, d'éliminer des essais infructueux.

- développement de méthodes heuristiques afin d'examiner en premier lieu les branches les moins coûteuses. On pourrait, par exemple, poursuivre "simultanément" la recherche de toutes les solutions et éliminer suivant certains critères les branches paraissant mener à des solutions trop coûteuses.

- introduction d'un dispositif permettant une gestion optimale des éléments spéciaux sur lesquels opèrent les instructions objets, registres ou index.

REGLES DE REDUCTION :

((MQ = M) (()) ((LDQ M) 1) P)

((:= M W0) (MQ = W0) ((STO M) 1) NP)

((ACC = M) (()) ((CLA M) 2) P)

((:= M W0) (ACC = W0) ((STO M) 5) NP)

((ACC = + W0 M) (ACC = W0) ((ADD M) 2) P)

((ACC = + M W0) (ACC = W0) ((ADD M) 1) P)

((ACC = - M W0) (ACC = W0) ((SUB M) 5) P)

((ACC = * W0 M) (MQ = W0) ((MULT M) 3) P)

((ACC = * M W0) (MQ = W0) ((MULT M) 2) P)

((MQ = / W0 M) (ACC = W0) ((DIV M) 7) P)

REGLES D EQUIVALENCE :

((R = OP W2 W3) ((R = OP W2 T) ET (:= T W3)))

((R = OP W1 W2) ((R = T) ET (:= T (OP W1 W2))))

CHAINE DE DEPART :

(:= F (* (+ A B) (/ C D)))

L EXPRESSION :

(:= F (* (+ A B) (/ C D)))

PEUT ETRE REDUITE PAR LA REGLE :

((:= M WO) (MQ = WO) ((STQ M) 1) NP)

CECI PRODUIT LA NOUVELLE EXPRESSION :

(MQ = * (+ A B) (/ C D))

ET L INSTRUCTION :

(STQ F)

A CE STADE,LE COUT DU PROGAMME EST DE: 1

L EXPRESSION :

(MQ = * (+ A B) (/ C D))

N EST PAS DIRECTEMENT REDUCTIBLE

L APPLICATION DE LA REGLE D EQUIVALENCE :

((R = OP W2 W3) ((R = OP W2 T) ET (:= T W3)))

A CETTE EXPRESSION PRODUIT LA SUITE D EXPRESSIONS :

((MQ = * (+ A B) T) ET (:= T (/ C D)))

PROGRAMME PARIBL GENERE POUR UNE SUITE D EXPRESSIONS :

( )

CETTE TENTATIVE A ECHOUE
L APPLICATION DE LA REGLE D EQUIVALENCE :

((R = OP W1 W2) ((R = T) ET (:= T (OP W1 W2))))

A CETTE EXPRESSION PRODUIT LA SUITE D EXPRESSIONS :

((MQ = T) ET (:= T (* (+ A B) (/ C D))))

L EXPRESSION :

(MQ = T)

PEUT ETRE REDUITE PAR LA REGLE :

((MQ = M) (( )) ((LDQ M) 1) P)

CECI PRODUIT LA NOUVELLE EXPRESSION :

(( ))

ET L INSTRUCTION :

(LDQ T0)

A CE STADE,LE COUT DU PROGAMME EST DE: 2

L EXPRESSION CONSIDEREE EST COMPLETEMENT REDUITE
DERNIER COUT : 2

PROGRAMME PARTIEL GENERE POUR UNE SUITE D EXPRESSIONS :
(((LDQ TO)))

L EXPRESSION :
(:= T (* (+ A B) (/ C D)))
PEUT ETRE REDUITE PAR LA REGLE :
((:= M W0) (M0 = W0) ((STQ M) 1) NP)
CECI PRODUIT LA NOUVELLE EXPRESSION :
(M0 = * (+ A B) (/ C D))
ET L INSTRUCTION :
(STQ T0)

A CE STADE,LE COUT DU PROGAMME EST DE: 3

L EXPRESSION :
(M0 = * (+ A B) (/ C D))
N EST PAS DIRECTEMENT REDUCTIBLE
L APPLICATION DE LA REGLE D EQUIVALENCE :
((R = OP W2 W3) ((R = OP W2 T) ET (:= T W3)))
A CETTE EXPRESSION PRODUIT LA SUITE D EXPRESSIONS :
((M0 = * (+ A B) T) ET (:= T (/ C D)))

PROGRAMME PARTIEL GENERE POUR UNE SUITE D EXPRESSIONS :
( )
CETTE TENTATIVE A ECHOUE

LA REGLE D EQUIVALENCE SUIVANTE:
((R = OP W1 W2) ((R = T) ET (:= T (OP W1 W2))))
NE PEUT S APPLIQUER A L EXPRESSION,CAR CELLE-CI A ETE TRAITEE PRECEDEMMEN

. . .

. . .

PROGRAMME PARTIEL GENERE POUR UNE SUITE D EXPRESSIONS :
(((MULT T0) (LDQ 1) STQ T1) (ADD A) (CLA B)))

L EXPRESSION :

(:= T (/ C D))

PEUT ETRE REDUITE PAR LA REGLE :

((:= M WO) (MQ = WO) ((STO M) 1) NP)

CECI PRODUIT LA NOUVELLE EXPRESSION :

(MQ = / C D)

ET L INSTRUCTION :

(STO TO)

A CE STADE,LE COUT DU PROGAMME EST DE: 20

L EXPRESSION :

(MQ = / C D)

PEUT ETRE REDUITE PAR LA REGLE :

((MQ = / WO M) (ACC = WO) ((DIV M) 7) P)

CECI PRODUIT LA NOUVELLE EXPRESSION :

(ACC = C)

ET L INSTRUCTION :

(DIV D)

A CE STADE,LE COUT DU PROGAMME EST DE: 27

L EXPRESSION :

(ACC = C)

PEUT ETRE REDUITE PAR LA REGLE :

((ACC = M) (()) ((CLA M) 2) P)

CECI PRODUIT LA NOUVELLE EXPRESSION :

(())

ET L INSTRUCTION :

(CLA C)

A CE STADE,LE COUT DU PROGAMME EST DE: 29

L EXPRESSION CONSIDEREE EST COMPLETEMENT REDUITE

DERNIER COUT : 29

PROGRAMME EN COURS DE FORMATION :

(((DIV C) (CLA C)))

PROGRAMME EN COURS DE FORMATION :

(((STO TO) (DIV C) (CLA C)))

L EXPRESSION :

(:= T (/ C D))

PEUT ETRE REDUITE PAR LA REGLE :

((:= M W0) (ACC = W0) ((STO M) 5) NP)

CECI PRODUIT LA NOUVELLE EXPRESSION :

(ACC = / C D) .

ET L INSTRUCTION :

(STO T0)

A CE STADE,LE COUT DU PROGAMME EST DE: 24

L EXPRESSION :

(ACC = / C D)

N EST PAS DIRECTEMENT REDUCTIBLE

L APPLICATION DE LA REGLE D EQUIVALENCE :

((R = OP W2 W3) ((R = OP W2 T) ET (:= T W3)))

A CETTE EXPRESSION PRODUIT LA SUITE D EXPRESSIONS :

((ACC = / C T) ET (:= T D))

PROGRAMME PARTIEL GENERE POUR UNE SUITE D EXPRESSIONS :

()

CETTE TENTATIVE A ECHOUE
L APPLICATION DE LA REGLE D EQUIVALENCE :

((R = OP W1 W2) ((R = T) ET (:= T (OP W1 W2))))

A CETTE EXPRESSION PRODUIT LA SUITE D EXPRESSIONS :

((ACC = T) ET (:= T (/ C D)))

L EXPRESSION :

(ACC = T)

PEUT ETRE REDUITE PAR LA REGLE :

((ACC = M) ((I)) ((CLA M) 2) P)

CECI PRODUIT LA NOUVELLE EXPRESSION :

((I))

ET L INSTRUCTION :

(CLA TO)

A CE STADE,LE COUT DU PROGAMME EST DE: 26

L EXPRESSION CONSIDEREE EST COMPLETEMENT REDUITE

DERNIER COUT : 26

PROGRAMME PARTIEL GENERE POUR UNE SUITE D EXPRESSIONS :

(((CLA TO)))

L EXPRESSION :

(:= T (/ C D))

PEUT ETRE REDUITE PAR LA REGLE :

((:= M WO) (MQ = WO) ((STQ M) 1) NP)

CECI PRODUIT LA NOUVELLE EXPRESSION :

(MQ = / C D)

ET L INSTRUCTION :

(STQ TO)

A CE STADE,LE COUT DU PROGAMME EST DE: 27

L EXPRESSION :

(MQ = / C D)

PEUT ETRE REDUITE PAR LA REGLE :

((MQ = / WO M) (ACC = WO) ((DIV M) 7) P)

LE COUT ACTUEL 34
EST SUPERIEUR AU DERNIER COUT ENREGISTRE

II.2. Un langage de génération.

Ce langage est destiné à l'écriture de programmes permettant d'exploiter un arbre syntaxique ou un arbre de dépendance au sens de [Co] [cf§ II.2.3.1.] . Il est largement tributaire d'ALGOL 60 en ce qu'il contient en partie

- les expressions arithmétiques
et les expressions booléennes.
- les instructions conditionnelles
- les instructions allera
- les instructions composées

de ce langage

L'originalité de ce langage de génération tient principalement :

- à la multiplicité des points d'entrée dans un programme. En effet, tout programme est divisé en deux chapitres, chacun d'eux correspondant à un sommet possible pour les arbres considérés. Certain de ces chapitres sont distingués de telle sorte qu'ils sont exécutés automatiquement si l'arbre donné possède un sommet correspondant au chapitre. Ceci reflète les possibilités d'alternance entre reconnaissance et génération dans une phase ASG. En d'autres termes, le programmeur a la possibilité de choisir ses "points de génération", c'est à dire de décider des arbres qui peuvent être traités indépendamment.

- à l'introduction d'un nouveau type, le type segment. Un segment est un support pour une chaîne objet générée. Des instructions sont prévues pour inclure un segment dans un autre.

- à la possibilité d'utiliser des variables dites "locales" qui désignent en fait des sommets de piles. Ceci permet de retenir des informations dans le cas où la grammaire du langage source est récursive.

- à l'introduction de descriptions de structure analogue à celle des listes. Ceci donne un moyen commode de stocker et de retrouver des informations et d'effectuer des calculs.

Ce langage est présenté d'une façon devenue classique depuis la parution du rapport ALGOL 60 Nau : à chaque description syntaxique sous Forme Normale de Backus (FNB) est associée une partie dite sémantique explicitant la signification des chaînes décrites précédemment.
Nous commencerons par préciser ce que nous entendons par arbre et arbre syntaxique.

II.2.0. Structure arborescente et arbre syntaxique.

Cette présentation est inspirée de [V V V]

1 n-graphes

Un graphe est représenté par le couple (Γ, x) où :

- X est un ensemble (ses éléments sont appelés noeuds).
- Γ une application multivoque de X dans X

si $\Gamma_1, \Gamma_2, \ldots, \Gamma_n$ sont des applications multivoques de X dans X on appelle n-graphe le n+1-uple :

$$(X, \Gamma_1, \Gamma_2, \ldots, \Gamma_n)$$

considéré comme le graphe union des graphes (X, Γ_1) $(X, \Gamma_2)\ldots$ (X, Γ_n) dans lequel tous les arcs sont désignés selon leur appartenance à l'application d'où ils proviennent.

2 Structure

Soit un n-graphe $(X, \Gamma_1, \Gamma_2, \ldots, \Gamma_n)$, V un ensemble de symboles (vocabulaire ou alphabet) et Δ une application univoque de X dans V.

On appelle structure le n+3-uple :

$$(X, V, \Gamma_1, \Gamma_2, \ldots, \Gamma_n, \Delta)$$

On déduit une structure d'un n-graphe en attribuant à chaque noeud xi du n-graphe un élément de V à l'aide de Δ. Des noeuds distincts de la structure pourront être désignés par le même élément : celui-ci sera dit "nom du ou des noeuds.

3 Structure arborescente

Une structure arborescente est un 5-uple $(X,V, \Gamma_1, \Gamma_2, \Delta)$, associé comme il est dit ci-dessus au 2-graphe (X, Γ_1, Γ_2), celui-ci étant tel que :

- le graphe (X, Γ_1) soit une arborescence, c'est à dire :

a) il existe un noeud $x_o \in X$ tel que $\Gamma_1 x_o = \emptyset$.
Ce noeud est appellé sommet.

b) pour tout noeud $x \neq x_o$, il existe un noeud y unique tel que $\Gamma_1 x = y$

c) le graphe est connexe.

- l'application Γ_1 étant donnée, l'application Γ_2 satisfasse aux conditions suivantes :

a) à tout sous-ensemble $\Gamma_1^{-1} x$ correspond une composante connexe de (X, Γ_2)

b) pour chacun de ces sous-ensembles $\Gamma^{-1}x$, la relation d'ordre associée à Γ_2 est totale. Ainsi, dans tout sous-ensemble $X_i = \Gamma^{-1}$ xi, il existe un $x_A \in X_i$ unique tel que $\Gamma_2^{-1} x_A = \varphi$ et un x_B unique tel que $\Gamma_2 x_B = \ell$.

Le graphe (X, Γ_1) peut être interprèté comme une relation de paternité :

$\Gamma_1 x = y \Longleftrightarrow$ y est père de x

De même, le graphe (X, Γ_2) exprime un ordre entre les fils d'un même père.

$\Gamma_2 x = y \Longleftrightarrow$ y est frère puiné de x

(nous dirons aussi de x est à gauche de y)

dans la suite, nous nous intéresserons aux couples (x, y) tels que :

$\Gamma_1 x = y$ et $\Gamma_2^{-1} x = \emptyset$

nous dirons alors que x est le fils ainé de y.

Nous emploierons aussi le terme d'arbre pour désigner une structure arborescente.

4 Arbre syntaxique associé à une grammaire context-free

Soit G une grammaire context-free (V_T, V_N, S, R) ; une structure arborescente $(X, V, \Gamma_1, \Gamma_2, \Delta)$ associée à G est telle que :

- $V = V_T \cup V_N$
- soit $T = \left\{ x \in X \mid \Gamma_1^{-1} x = \emptyset \right\}$

alors $\Delta = \Delta_1 \cup \Delta_2$

Δ_1 étant une application univoque de T dans V_T Δ_2 étant une application univoque de X-T dans V_N avec Δ_2 (xo) = S

- si Δ (xi) : ψ , $\psi \in V_N$

et si

pour yo $\in$ X , Γ_{yo} = xi et Γ_2^{-1} yo = $\emptyset$, Δ(yo) = ψ_o

pour yj $\in$ X, yj = Γ_2 yj-1, Δ(yj) = ψ j $1 < j < n$

pour $y_n \in X$, yn = Γ_2 yn-1 et Γ_2 yn = $\emptyset$, Δ(yn) = ψ n

il existe dans R la règle

$$\psi \longrightarrow \psi_o \psi_n \ldots \psi_n$$

II.2.1. Description

-1- Programme génération

syntaxe :

<programme génération> : : = début <déclaration>
<initialisation>
<corps>
fin

sémantique :

Un programme de génération opère sur une donnée qui est une structure arborescente. Il fournit en résultat les valeurs des segments déclarés dans le programme (cf 8 et 2).

-2- Déclaration - Initialisation.

Syntaxe :

<déclaration> : : = <déclaration simple> ; |
<déclaration simple> ; <déclaration>

<déclaration simple> : : = <type> <liste d'identificateurs>

<type> : : = local <type de base> | <type de base>

<type de base> : : = entier | booléen | segment

<liste d'identificateurs> : : = <identificateurs> |
<identificateur>,<liste d'identificateurs>

<initialisation> : : = <suite d'instructions>

Sémantique :

Les variables de type ent r ou booléen ont la même signification qu'en ALGOL 60.

La valeur d'une variable de type segment est une suite d'instructions objet (cf 7). Cette suite est formée à l'aide des actions générer et inclure (cf 5).

La notation adoptée pour représenter la suite attachée au segment p, à un instant donné de l'exécution, est la suivante :

$$\mathcal{U}_1^p \; \mathcal{U}_2^p \; \ldots\ldots \mathcal{U}_i^p \ldots \mathcal{U}_m^p$$

où $\mathcal{U}_i^p$ représente une instruction objet, m est "l'indice maximale" de la suite.

Si la variable déclarée est spécifiée local, sa valeur est déterminée par l'emploides actions ancien et nouveau (cf 5). La suite d'instructions représentée par <initialisation> est exécutée préalablement à tout traitement d'arbre par un chapitre.

-3- Corps

Syntaxe :

<corps> : : = <chapitre> | <chapitre> # <corps>

<chapitre> : : = <liste d'en-têtes> <corps de chapitre>

<liste d'en-têtes>: : = <en-tête> | <en-tête> <liste d'en-têtes>

<en-tête> : : = e <opérateur> : | * e <opérateur> :

<corps de chapitre> : : = <suite d'instructions> |
<corps de chapitre 1>

<corps de chapitre 1> : : = <sous-chapitre> |
<sous-chapitre> <corps de chapitre1>

<sous-chapitre> : : = <liste d'en-têtes-1> <suite d'instructions>

<liste d'en-tête 1> : : = <en-tête1> | <en-tête 1> <liste d'en-têtes 1>

<en-tête 1> : : = e : | e <profil> :

<opérateur> : : = <symbole> | <symbole> <opérateur>

<profil> : : = <o1 > | <01> <profil >

<01> : : = 0 | 1

<suite d'instructions> : : = <instruction > | <instruction >;<suite d'instructions>

Sémantique :

Le corps de chaque chapitre représente le travail à effectuer si l'arbre "courant" possède un sommet dénoté par l'opérateur associé au chapitre. Nous désignons par arbre courant soit celui fourni en donnée, soit celui spécifié par une action traiter (cf 5). Dans ce dernier cas, l'exécution du chapitre est toujours effectuée : dans le premier elle ne l'est que si le chapitre possède une en-tête débutant par le symbole "*" (chapitre d'entrée).

Un chapitre peut être divisé en sous-chapitres à chacun desquels sont associés ou non un ou plusieurs profils. Seule la suite d'instruction correspondant à un de ces sous-chapitres est exécutée lorsqu'on atteint le noeud considéré. Ces instructions se déroulent séquentiellement.

Le sous-chapitre en question est :

- soit celui dont l'en-tête contient un profil "acceptable"
- soit celui dont l'en-tête ne contient pas de profil.

Ce sous-chapitre doit être unique.

Le profil est dit acceptable s'il est identique à la suite de 0 et de 1 obtenue de la façon suivante :

considérant de gauche à droite les fils (cf § II.2.0.) du sommet traité, on construit une suite de 0 et de 1, créant un 0 si l'élément considéré n'a pas de descendant (cf 6) et un 1 dans le cas contraire.

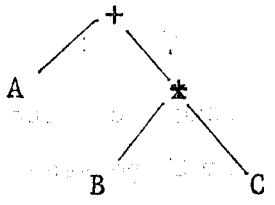
Exemple :

<chapitre> :

* e +:

e o o : <suite d'instructions>
e o 1 : < " " >
e 1 o : < " " >
e 11 : < " " >

arbre



Le sous-chapitre distingué est celui dont l'en-tête est e o 1.

-4- Instruction

Syntaxe :

<instruction> : : = <instruction conditionnelle> | <instruction inconditionnelle> |
<instruction parcourant>

<instruction inconditonnelle> : : = <instruction de base> |
<instruction composée>

<instruction de base> : : = <instructions non étiquetée de base> |
<étiquette> : <instruction de base>

<instruction non étiquetée de base> : : = <instruction d'affectation> |
<instruction allera> |
<action>

<instruction d'affectation> : : = <partie gauche> : = <expression>

<expression> : : = <expression entière> | <expression booléenne> |
<expression description> |

<partie gauche> : : = <composant> | <variable entière> | <variable booléenne> |

<instruction allera> : : = allera <étiquette>

<instruction composée> : : = <composée non étiquetée> |
<étiquette> : <instruction composée>

<composée non étiquetée> : : = début <suite d'instructions> fin

<instruction conditionnelle> : : = <instruction S I> |
<instruction SI> sinon <instruction > |
<proposition si> <instruction parcourant> |
<étiquette> : <instruction conditionnelle>

<proposition SI> : : = si <expression booléenne> alors

<instruction si> : : = <proposition si> <instruction inconditionnelle>

<instruction parcourant> : : = parcourant <expression description>
avec <variable entière> faire <instruction> |
<étiquette> : <instruction parcourant>

Sémantique :

Les instructions composées et conditionnelles (avec les restrictions apportées par la syntaxe) et les instructions d'affectation ont essentiellement la même signification qu'en ALGOL 60. Pour ces dernières, les types à gauche et à droite doivent être compatibles lorsque le type gauche est imposé (cas des variables entières et booléennes (cf 11) et segment (cf 8)). L'effet d'une affectation entre segments est le suivant : le segment gauche désignera la même suite d'instructions objet que le segment droit et non une copie de cette suite. Un composant (cf 10) peut prendre une valeur de type quelconque, entier, logique, segment, description ou nom (cf 9).

L'attribution à un nom d'une description n'est pas possible au moyen d'une affectation : nous verrons (cf 5) qu'une action spéciale (établir) est prévue à cet effet.
La partie d'une étiquette est restreinte au chapitre ou, le cas échéant, au sous chapitre dans lequel se trouve une instruction allera correspondante (ou encore à la partie initialisation.
L'instruction parcourant entraine la répétition de l'exécution de l'instruction correspondante tant que la variable entière, initialisée à zéro et incrémentée de 1 à chaque exécution possède une valeur : telle que le ième composant de la description existe.

Exemples : consulter aussi les exemples des § 5, 9 et 10.

1) Affectation d'une valeur entière à un composant :

* i ri *[3] : = 4

2) Affectation d'une description à un composant

$\langle a \rangle [6] := \{ 3, \{ 0, * i r 2 i * \} \}$;

* ri * [2] : = $\{ \langle a \rangle [6,2] \}$

La valeur du deuxième composant de la description associée à * r i * est la description $\{0, * i r 2 i *\}$

3) * i r i * : = <------>

est une écriture non permise.

4) parcourant $\langle a \rangle$ avec i faire

$\langle a \rangle [i] : = 1$;

est une instruction donnant à tous les composants de la description associée à $\langle a \rangle$ la valeur 1.

5) Actions.

syntaxe :

<action> : : = traiter <descendant> |
générer <instruction objet> sur |
prévoir dans |
inclure dans |
annuler
ancien <variable locale> | nouveau <variable locale>

établir <nom> = <description> |

initialiser <nom> |

supprimer <expression entière> de <expression description>

<variable locale> : : = <variable entière locale> |

<variable booléenne locale> |

<segment local>

Sémantique :

L'action traiter permet de suspendre momentanément l'exécution du chapitre courant et de se référer au chapitre désigné par le descendant (cf 6) associé à l'action. Une fois ce chapitre terminé, on revient exécuter le chapitre primitif. Si le descendant ne désigne aucun noeud ou s'il désigne le sommet d'un arbre déjà traité l'action est ineffective.

L'action générer a pour but d'ajouter l'instruction objet indiquée à la suite représentée par le segment associé à cette action. Si la suite est dénotée (cf 2) :

$$u_1^p u_2^p \ldots u_m^p ,$$

on ajoute l'instruction objet u_{m+1}^p pour que la suite devienne :

$$u_1^p \; u_2^p \ldots u_m^p \; u_{m+1}^p$$

inclure est une action destinée à intercaler la suite désignée par le premier segment cité, dans la suite désignée par le second. L'indice où débute ce placement est l'indice maximal possédé par la suite associée au second segment à l'exécution de la précédente action prévoir s'y rapportant. Cette action prévoir ne peut plus être ainsi utilisée par la suite. Dans le cas où plusieurs de telles actions prévoir ont été activées inclure considère celle qui l'a été le plus récemment et qui n'a pas été utilisée.

Soit po le premier segment, p le second et i l'indice maximal dont il est question. Si les dénotations des suites po et p1 avant l'action incluse les concernant sont respectivement :

$$u_1^{po} \ldots u_i^{po} \; u_{i+1}^{po} \ldots u_m^{po}$$

et

$$u_1^{p1} \ldots u_k^{p1}$$

l'effet de celle-ci est de transformer la suite po en :

$$v_1^{po} \ldots v_i^{po} \; v_{i+1}^{po} \ldots v_{i+k}^{po} \; v_{i+k+1}^{po} \ldots v_{m+k}^{po}$$

dans laquelle v_j^{po} est une instruction objet telle que :

$$v_j^{po} = u_j^{po} \quad \text{pour } 1 \leqslant j \leqslant i$$

$$v_{i+j}^{po} = u_j^{p1} \quad \text{pour } 1 \leqslant j \leqslant k$$

$$v_{i+k+j}^{po} = u_{i+j}^{po} \quad \text{pour } 1 \leqslant j \leqslant m$$

annuler rend vide la suite représentée par le segment associé.
L'action nouveau indique que l'identificateur représentant une variable spécifiée locale et déclarée d'un type donné désigne désormais une autre variable de ce type. L'identificateur désignera à nouveau la première variable de ce type à la prochaine action exécution de l'action ancien s'y rapportant. établir associe à un nom une description (cf 9). Il est ensuite possible à l'aide de ce nom d'obtenir la valeur d'un composant de cette description.
Supprimer a pour but de détruire le composant de la description désignée par l'expression description et déterminé par la valeur n de l'expression entière. Les composants de cette description sont réordonnés, c'est à dire que le m-ième composant (m>n) devient le m-1ième.
Initialiser supprime le lien existant entre un nom et sa description.

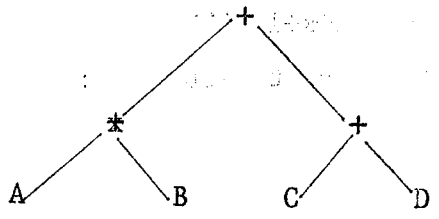
Exemples :

1) On peut écrire le chapitre :

et :

e 11 : traiter x i * ;
traiter * r i * ;
générer "add" sur segment 1

l'action de ce sous-chapitre, ayant à traiter l'arbre :



conduit à considérer les deux sous-arbres dépendant du sommet "+", à exécuter les sous-chapitres correspondants et à ajouter dans la suite désignée par segment 1 l'instruction objet "add".

2) Soient α et β deux opérateurs. Supposons qu'un arbre de sommet α soit traité avant celui-de sommet β .
On désire générer pendant tout le traitement sur le segment po, puis à l'occurence de β , générer sur le segment p1 et intercaler le contence de p1 en po à l'endroit correspondant à la première génération du chapitre de α .
On peut écrire :

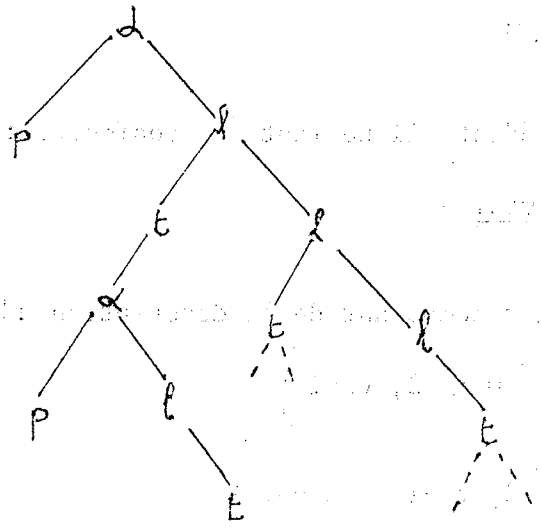
eα :

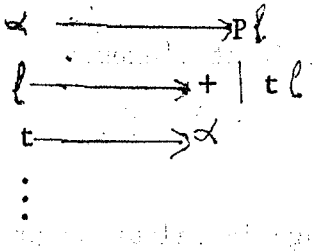
prévoir p1 dans po ;
générer <------> sur po ;
traiter * <----> * #

e β :

```
générer <-----> sur p1 ;
inclure p1 dans po ;
traiter * <----> *      #
```

3) Soit à traiter un arbre au sommet duquel est attribué l'opérateur α , possédant des sous-arbres aux sommets desquels est attribué ce même opérateur. Si on suppose que le traitement de α nécessite l'emploi d'une variable entière par exemple, celle-ci doit être nouvelle à chaque traitement de α . Elle sera alors déclarée locale. Considérons par exemple l'arbre suivant (pouvant être associé à la grammaire ci-contre) :





Si on désire que la variable entière et locale k prenne dans le chapitre correspondant à α une valeur égale au nombre de t non contenus dans un sous-arbre de sommet α , on peut écrire :

```
e α :
 e o1 : nouveau k ;
        traiter * ri * ;
           :
        ancien  k       #
```

```
et :

    traiter * i * ;
        .
        .
        .
    k : = K + 1      #
```

4) établir ‘a’ = { 1, * i r i * }

attribue au nom ‘a’ une description telle que :

‘a’ [1] = 1

et ‘a’ [2] = * i r i *

supprimer 1 de ‘a’ conduit à :

‘a’ [1] = * i r i *

5) Considérant l'exemple précédent, il ne faut pas confondre :

‘a’ [1]: = { 2, vrai }

qui change la valeur du premier composant de la description adjointe à ‘a’ et établir ‘a’ [1] = { 2, vrai }

qui attribue la description { 2, vrai au nom }

* i r i * qui est la valeur de ‘a’ [1] .

6) Descendant

Syntaxe :

<descendant> : : = * <suite i r > *

<suite i r> : : = <composition i r> | <composition i r> <suite i r>

<composition i r> : : = <fils a frère p> | <fils a frère p><expression entière>

<fils a frère p> : : = i | r

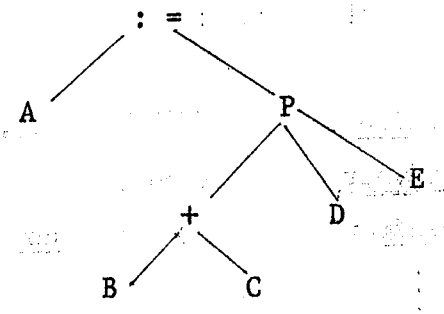
Sémantique :

Un descendant représente un noeud dans l'arbre considéré dans le chapitre.Un fils a représenté par i est une fonction fournissant le fils ainé (cf II.2.0) du noeud considéré. Un frère p représenté par r fournit le frère puiné du noeud considéré. Le premier noeud considéré est le sommet de l'arbre courant.

Une composition i r est la composition de n fils a ou de n frère p, n étant la valeur de l'expression entière. Une suite i r est une composition des précédentes fonctions.

Exemple :

Soit l'arbre :

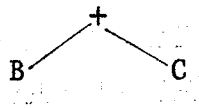


Dans le chapitre consacré à ":="

à " : = "

* i * représenté le noeud A

* i r i * représente le sommet de l'arbre



* r 2 i r i * représente le noeud E.

7) Instruction objet

Syntaxe :

<instruction objet> : : = <élément> | <élément> <instruction objet>
<élément> : : = <élément 1> | .<élément 1>
<élément 1> : : = "<chaîne>" | <expression entière> | <nom>

Sémantique :

Une instruction objet peut être formée de la concaténation de chaînes représentant des éléments du langage objet, des valeurs d'expressions entières et des noms (cf 9). Toutefois, la concaténation de valeurs d'expressions entières entre elles nécessite l'emploi du connecteur " " (chaîne vide).

UN "." figurant en tête d'un élément signifie qu'on augmente de 1 le compteur (cf 12) associé au segment considéré.

Exemples :

1) "si t =" n+3 "alors t ["x"] : = 1 "

2) Soient α et β deux opérateurs. Supposons qu'un noeud α soit traité avant un noeud β .

Soient les deux chapitres :

eα :

```
générer. "␣␣␣␣␣␣␣   CLA" sur po ;
prévoir   p1 sur po ;
générer   "␣␣␣␣  X " sur po ;
  .
  .
  .
  #
```

eβ :

```
  .
générer " * " sur p1 ;
inclure p1 dans po ;
  .
  .
  .
  #
```

Après l'exécution du chapitre β on obtient sur le segment po l'instruction :

␣␣␣␣␣␣␣CLA *␣␣␣␣␣ X

8) Segment

Syntaxe :

 : : = <segment global> |
<segment local> | <composant segment>
<segment local> : : = <identificateur> |
<identificateur> moins<expression entière>
<segment global> : : = <identificateur>

Sémantique :
Un segment local est un segment déclaré local. On a vu (cf 5) qu'il était possible à l'aide de l'action nouveau de créer dynamiquement un nouveau segment.
A l'aide de l'expression entière (toujours positive) attachée (éventuellement) à un segment local, on pourra atteindre un segment crée auparavant. Plus précisément, la valeur n de l'expression permet d'atteindre le segment depuis lequel on en a créé n autres.

Exemple :

Soient les déclarations :

segment po ;
local segment pl ;

et les chapitres :

e α : ....

nouveau p 1 ;
prévoir p 1 dans po ;
traiter * < >* ;
ancien p 1
... #

e β : .....

inclure p 1 dans po ;
inclure p 1 moins 1 dans po ;
..... #

Si on examine successivement deux sommets α et ensuite un noeud β, le contenu des deux derniers segments p 1 créés dans le chapitre α sont intercalés dans p o au cours du traitement associé au sommet β.

9) Expression description

Syntaxe :

<expression description> : : = <nom> | <description>
<nom> : : = (<chaîne>) | <descendant> | <composant nom>
<description> : : = { <description 1> }
<description 1> : : = <élément 2> | <élément 2>, <description 1>
<élément 2> : : = description <expression description> |
copie <expression description> |
<composant description> |
<expression>

Sémantique :

Un nom peut être, soit une chaîne choisie par le programmeur, soit le nom d'un noeud d'arbre (cf § II.2.0), soit encore la valeur d'un composant. A l'aide de l'action établir il est possible de lui associer une description.

Une description est un ensemble ordonné d'éléments qui peuvent être soit de type entier, booléen, segment soit un nom ou une description. On atteint les éléments d'une description à l'aide d'un composant.

L'opérateur description fournit la description associée à un nom.

Supposons que cette description contienne n éléments. Deux remarques sont à faire lorsqu'on ajoute m éléments à cette description :

- la description résultante possède m + n éléments
- les éléments ajoutés à droite (c'est à dire de numéro i : i> n dans la description résultante) sont aussi ajoutés à droite de la description adjointe au nom. Il en est de même s'il s'agit d'une description désignée par un composant description.

L'opérateur copie, appliqué à un nom fournit une reproduction de sa description, appliqué à une description, une reproduction de celle-ci. Cette recopie ne corcerne que les composants de la description et non les descriptions que pourraient désigner ses composants.

Exemples :

1) Descriptions :

$$\{\, n + 1,\ *\ \underline{i}\ \underline{i}\ *\ ,\ \{\, \underline{vrai},\ 3\,\}\,\}$$

$$\{\, \text{'liste 1'}\ ,\ 5\,\}$$

2) On associe au nom ‘liste’ une description, soit :

établir ‘liste’ = { 1, * i r i *, vrai }

l'action :

établir ‘liste 0’ = { 5, ‘liste’ }

implique :

‘liste o’ [1] = 5,
‘liste o’ [2,1] = 1,
‘liste o’ [2,2] = * i r i * ,
‘liste o’ [2,3] = vrai

Dans le cas où on effectue :

établir ‘liste 1’ = { 5, description ‘liste’ }

on obtient :

‘liste 1’ [2] = 1,
‘liste 1’ [3] = * i r i *,
‘liste 1’ [4] = vrai

et toujours : ‘liste’ [1] = 1

Si on effectue::

établir ‘liste 2’ = { description ‘liste’ , 5 }

on obtient :

‘liste 2’ [4] = 5 mais aussi :

‘liste’ [4] = 5

On a donc modifié la description adjointe à liste . Dans les deux cas précédents, elle serait aussi modifiée par les affectations :

‘liste 1’ [2] = 7 ou

‘liste 2’ [2] = 7

Enfin les actions :

établir ‹liste 1› = { 5, copie ‹liste› }

établir ‹liste 2› = { copie ‹liste›, 5 }

ont le même effet que celles décrites ci-dessus quant aux valeurs des composants des descriptions de ‹liste 1› et ‹liste 2› . Toutefois, la description de ‹liste› n'est pas modifiée et il devient impossible de le faire à partir des noms ‹liste1› et ‹liste 2›.

10) Composant

Syntaxe :

<composant> : : = <composant entier> |
<composant booléen> |
<composant description> |
<composant nom> |
<composant segment>

<composant entier> : : = < f composant>
<composant booléen> : : = <f composant>
<composant description> : : = < f composant>
<composant nom> : : = < f composant>
<composant segment> : : = <f composant>
<f composant> : : = <expression description> [<suite d'indices>]
<suite d'indices> : : = <expression entière> |
<expression entière>, <suite d'indices>

Sémantique :

Un composant est un élément d'une description : dans le cas où l'expression description est un nom la description considérée est celle associée à ce nom.

On accède à cet élément à l'aide d'une suite d'indices. On considère d'abord le premier indice (le plus à gauche dans la suite d'indices) : sa valeur détermine un élément de la description. Si un indice suivant est présent et si cet élément désigne un nom ou une description, on recommence le processus en considérant cet indice et cette description. Dans le cas contraire, le composant est l'élément ainsi déterminé.

Exemple :

Soit

établir ⌜liste⌝ = { n + 1, * i i * , { vrai, 3 } }

établir * i i * = { ⌜a⌝ , 5 }

⌜liste⌝ [3,2] a pour valeur 3

⌜Liste⌝ [2,2] a pour valeur 5

On remarque que l'écriture précédente est une contraction de ⌜liste⌝ [2] [2].

11) Expressions entières et booléennes.

Syntaxe :

<variable entière locale> : : = <identificateur> |
<identificateur> moins<expression entière>

<variable entière globale> : : = <identificateur>

<variable booléenne locale> : : = <identificateur> |
<identificateur>moins<expression entière>

<variable booléenne globale> : : = <identificateur>

Les expressions booléennes et entières sont analogues respectivement aux expressions booléennes et arithmétiques (avec opérandes et résultats entiers) d'ALGOL et :

<primaire arithmétique> : : = <valeur entière> | <variable entière> | (<expression entière>)

<variable entière> : : = <variable entière globale> | <variable entière locale> | <composant entier>

<primaire booléen> : : = <valeur booléenne> | <variable booléenne> | <relation> | (<expression booléenne>)

<variable booléenne> : : = <variable booléenne globale> | <variable booléen locale> | <composant booléen>

Les relations sont aussi identiques à celle d'ALGOL 60, de plus :

<relation> : : = <nom> eq <nom> |
présence de <expression> dans <expression description>

Sémantique :

Les opérateurs arithmétiques et booléens ont la même signification qu'en ALGOL 60.

La relation eq permet de tester l'égalité de deux noms, la relation présence de est vraie si l'expression description contient un élément même valeur que l'expression associée.

De même que les segments locaux, les variables entières et booléennes locales permettent d'accèder à la valeur d'une variable créée auparavant à l'aide de l'action nouveau.

Exemple :
Soit établir 'a' = {vrai, x + 2, 'b'}
présence de 'b' dans 'a' et
'b' eq 'a' [3]
sont alors vraies.

12) Valeurs entières et logiques.

Syntaxe :

<valeur entière> : : = nombre |
compteur |
<nombre entier>

<valeur logique> : : = vrai | faux |
terminal <descendant> |
nul <expression description>

Sémantique :

nombre fournit, dans chaque chapitre, le nombre de fils du sommet de l'arbre courant.

compteur donne la valeur du compteur ordinal des instructions objet créées sur un segment donné. On augmente la valeur de ce compteur en plaçant un point dans une instruction objet (cf 7). Ce compteur est remis à zéro par une action annuler correspondant au segment.

terminal prend la valeur vraie si le descendant considéré n'a aucun fils, faux dans le cas contraire.

Nul est vraie si la description désignée ne contient aucun élément.

Exemple :

Supposons que "alléra" et "." soient des opérateurs possèdant des étiquettes comme opérandes. On désire traduire une instruction alléra par une instruction "tra" opérant un transfert de contrôle à l'instruction dont le numéro lui est indiqué par sa partie adresse.

On peut écrire :

```
Segment po ;                                              arbres :

  ...
e : établir * i * = { compteur po } ;                        :
    traiter * r i *                                         / \
....                                                       f
e allera : eo : générer "tra"  * i * [1] sur po ;                alléra
                                                                  /
                                                                 f
```

13) Nombres entiers, étiquettes, symboles, identificateurs, chaînes.

Les nombres entiers et les identificateurs sont définis comme en ALGOL 60. Les étiquettes sont représentées comme des identificateurs. Les symboles représentent les éléments de l'alphabet dont on dispose. Une chaîne est une suite de symboles ne contenant pas les symboles ' ,) et " .

II . 2. 2. Inclusion du langage de génération en ALGOL 60..

Les éléments de ce langage traités ici ont essentiellement trait :

- à la liaison entre chapitres à l'aide de l'action traiter et des descendants -

- à la représentation d'un segment et des opérations afférentes -

- aux variables locales

- à la représentation des noms, des descriptions et des opérations associées.

Le langage hôte en fait utilisé est ALGOL-LISP [CoK]

1 - Organisation générale -

L'arbre fourni au programme de génération est représenté par une liste LISP décrite sous FNB de la façon suivante :

<arbre >: = (<noeud> u <suite de branches>)
<suite de branches> :: = <branche> | <suite de branches> u <branche>
<branche >:: = <élément terminal > | <arbre >
<noeud > :: = <élément non terminal>

Pour illustrer ceci , on voit que dans un arbre syntaxique <élément terminal> représente un symbole terminal de la grammaire et <élément non terminal> une métavariable de la grammaire.

L'action traiter est représentée par un appel à une procédure récursive appelée "traiter" admettant un parametre entier ? désignant un arbre (on sait qu'en LISP-ALGOL un entier peut désigner une liste) -

Dans cette procédure, un aiguillage appelé "chapitre" est déclaré dont les étiquettes se réfèrent à des débuts de chapitres. Une entière procédure appelée "numéro" permet de déterminer le numéro de l'étiquette correspondant au chapitre du <noeud> de l'arbre considéré (ce numéro est préalablement inclus dans la liste de propriétés de cet atome). A l'entrée dans "traiter" une instruction allera renvoie au chapitre recherché.

A la fin de chaque chapitre une autre instruction alléra renvoie à la fin du corps de la procédure traiter.

Exemple :

Si les atomes ALPHA, BETA et GAMMA représentent des noeuds et si leurs numéros associés sont respectivement 1, 2 et 3, la procédure traiter a la structure suivante :

```
procédure traiter  (1) ;
   valeur 1 ; entier 1 ;
   aiguillage chapitre : = e1, e2, e3 ;
   allera  chapitre [numéro] ;

e1 : <chapitre de ALPHA> ; alléra termine ;
e2 : <chapitre de BETA>  ; alléra termine ;
e3 : <chapitre de GAMMA> ;

termine :
fin ;
```

Pour déterminer un descendant, on utilise l'entière procédure "descendant", déclarée dans traiter et définie comme suit :

```
entier procédure descendant  (n) ;
    valeur n ; entier n ;
```

```
début entier z, i, x ;
    x : = 1 ;
    pour z : = n - (n'/'10) * 10 tant que n ≠ 0 faire
        début
            pour i := 1 pas 1 jusqu'à z faire
              x : = cdr (x) ;
              x : = car (x) ; n :=n'/' 10
            fin ;
descendant : = x ;
    fin ;
```

Deux restrictions sont introduites par l'emploi de cette procédure :

- Dans un chapitre donné, les "frères" du noeud associé sont inaccessibles, c'est à dire que tout descendant possède au moins un i à son extrême droite.

Le nombre maximum de "frères" est limité à 8.

Pour déduire le paramètre n de l'expression d'un descendant, on procède ainsi :

- n est initialisé à zéro
- considérant un entier N initialisé à zéro et les éléments i^m et r^k ($k < 10$) du descendant pris de droite à gauche :
  - pour un i^m, n prend la valeur $\sum_{i:=0}^{i:=m-1} 10^{i+N} + n$

    et N la valeur N + m - 1
  - pour un r^k, n prend la valeur $k10^N + n$

Exemple :

traiter * ri3 * s'écrit traiter (descendant (2 11))

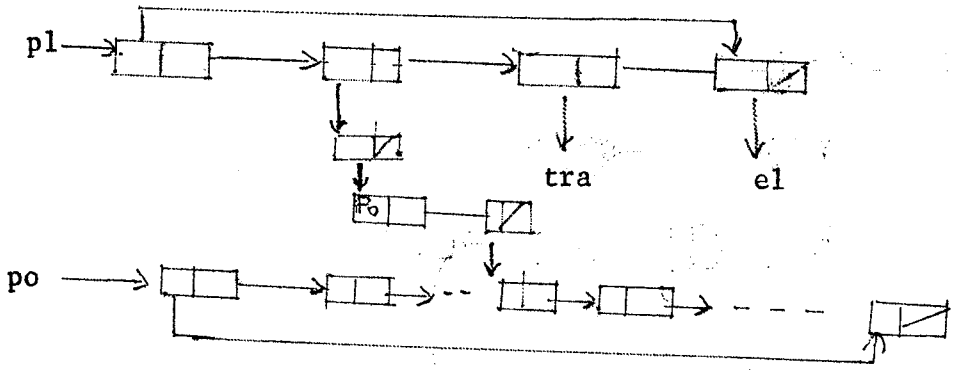
traiter * r3i * s'écrit traiter (descendant (4))

traiter * ir3i * s'écrit traiter (descendant (14))

2. Segments et variables locales.

Un segment est représenté par une variable entière qui désigne une liste LISP. Le premier élément de cette liste repère la fin du programme généré attenant au segment. Le second élément retient l'emplacement où éventuellement ce segment doit être inclus. Les autres éléments contiennent des références aux instructions objet générées sur ce segment.

Exemple : Soit pl un segment sur lequel on a généré "tra" puis "el" et devant être inclus dans le segment po. Il peut être représenté par :



L'action <u>générer</u> est impossible à utiliser telle quelle en ALGOL 60. Les chaînes qui font partie des instructions objet doivent être mises sous forme d'atomes. Les expressions entières seront aussi converties en atomes après leur évaluation. On utilise alors la procédure "générer" qui possède deux paramètres ; le premier est une liste d'atomes représentant les instructions objet, le second un segment. Elle place en bout de la liste du segment la liste d'atomes.

Exemple :

<u>générer</u> "CLA␣␣␣␣␣ = n <u>sur</u> po s'écrit

générer (cons (cla, atent (n)), po)

- cla est l'atome CLA ␣␣␣␣␣ =
- atent (x) délivre un atome représentant la valeur de x.

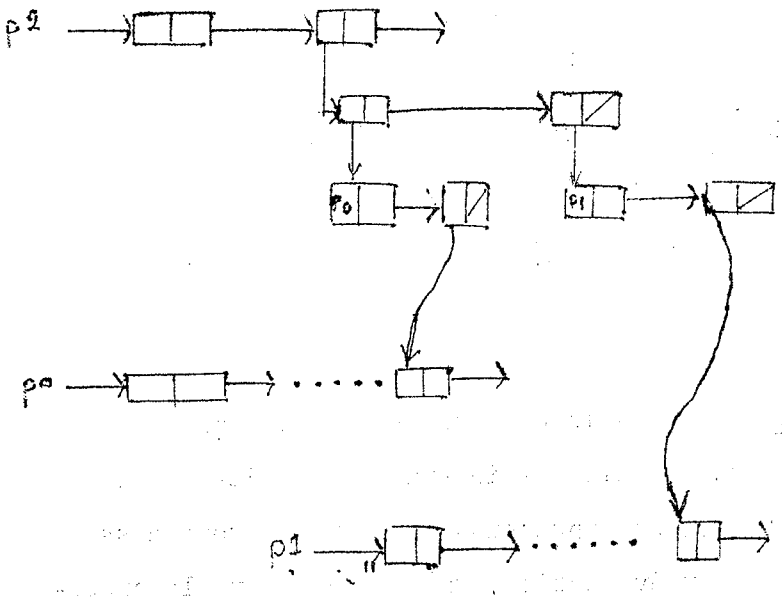
prévoir et inclure sont aussi traitées comme des procédures, ayant deux segments comme paramètres.

La première complète le second élément de la liste du premier segment : cet élément a pour charge de retenir le second segment et son indice maximal actuel.

Exemple :

```
prévoir   p2 dans po ;
prévoir   p2 dans p1 ;
```

produit :



La procédure "inclure" utilise les informations stockées dans "prévoir" pour intercaler une copie des instructions objet du premier segment dans le second.

Une variable locale possède une valeur située sur le sommet d'une pile. Elle est représentée par un entier dont la valeur se refère au sommet de cette pile : cette dernière est représentée en liste. Chaque niveau est une sous-liste à deux éléments : le premier repère le niveau

précédent, le second la valeur de la variable. La procédure "local" permet d'obtenir cette valeur. Une autre procédure "affecter local" permet l'affectation à une telle variable.

Les actions nouveau et ancien sont représentées par des procédures de même nom : étant donné une variable locale en paramètre, elles ont pour but respectivement d'augmenter ou de rabaisser la pile d'un niveau.

Exemple : Si on effectue :

```
i:= 3;
nouveau  i ;
    i:=4 ;
nouveau  i ;
    i:=5 ;
```

i désigne la liste :

(((( ) 3) 4) 5)

Si on effectue

```
ancien  i
```

i devient : ((( ) 3) 4).

3. Descriptions.

Un nom est représenté par un atome auquel on a adjoint, dans sa liste de propriétés, une valeur entière repérant une liste. Cette dernière figure la description associée au nom. Une telle liste a pour éléments soit des atomes de type entier , logique ou nom, soit une sous-liste, et est atteinte à partir du nom à l'aide de la procédure "description".

établir est simulée par une procédure affectant à la liste de propriétés du nom considéré un pointeur sur la liste à adjoindre.

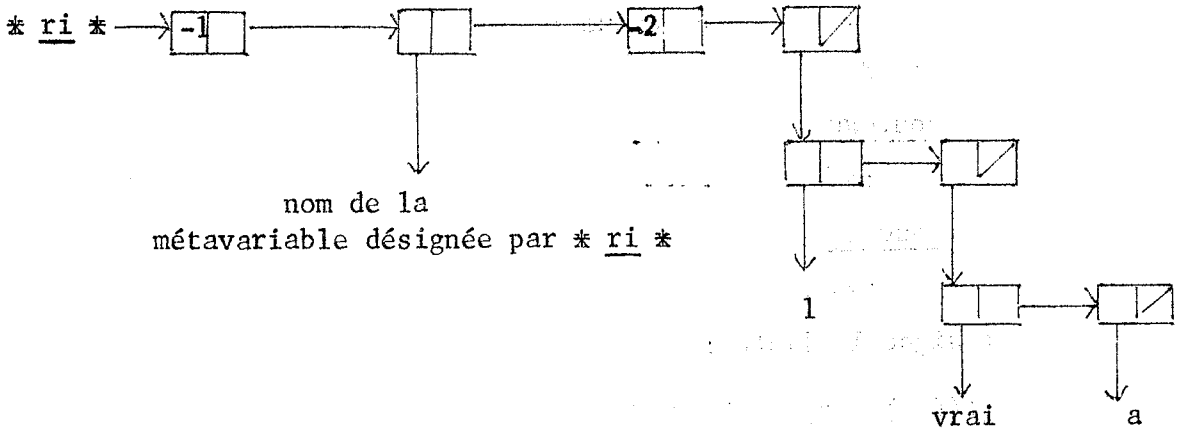
Exemple :

établir * ri * = { 1, { vrai , ' a ' } }

s'écrit

établir (descendant (2), cons (1,cons(cons(vrai,a),nil)))

et produit :



Une description est formée à l'aide de l'entier procédure q à deux paramètres représentant des listes. Cette procédure forme une liste dont les premiers éléments sont ceux de la première liste et les seconds ceux de la seconde.

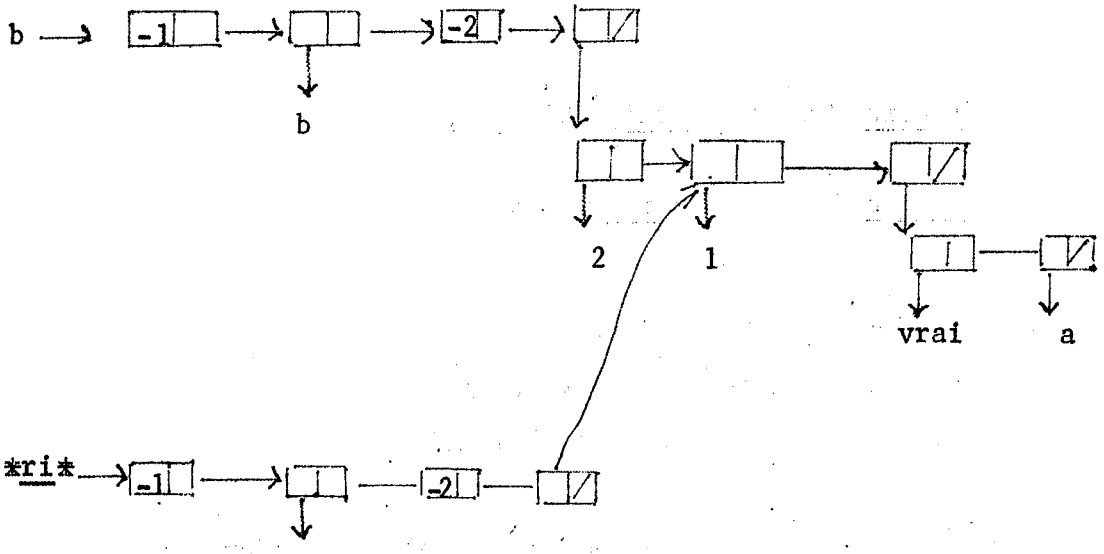
Exemple : * ri * étant tel que dans l'exemple précédent

1) établir ' b ' = { 2, description * ri * }

s'écrit

établir (b, q (2, description (descendant (2))))

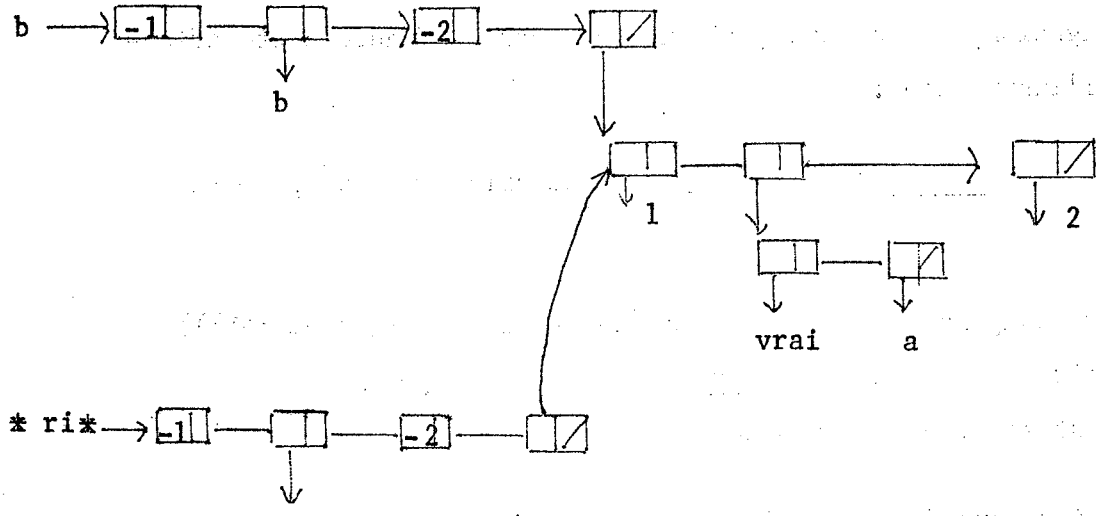
et fournit la configuration :



2) L'instruction

établir 'b' = { description * ri * , 2

aurait produit :



On remarque ici que la description de * ri * est modifiée.

Un composant est accessible à l'aide de l'entière procédure composant (n,x) où x est une liste ou un nom et n un nombre formé à partir de la suite d'indices du composant. En supposant que g soit le nombre maximal de composants des descriptions, on obtient n en faisant la somme des produits des indices (pris de gauche à droite) par g élevé à la puissance du rang de l'indice (le rang de l'indice le plus à gauche étant 0). Lorsque x est un nom, on considère sa description.

Exemple :

Soit

établir ‘a’ = { vrai, ‘b’ , { 3, 4 } }

établir ‘b’ = { * iri * , 5 }

- x : = ‘a’ [3, 2] s'écrit
  x : = car (composant (3+2 * g, a))
  et a pour effet d'affecter 4 à x.

- ‘a’ [2,1] : = { description ‘c’ , n +1 } s'écrit :

affecter car (composant (2 + g * 1,a), q (description (c), atent (n+1)) et a pour effet de remplacer le premier composant de b par la liste indiquée.

Signalons de nouveau qu'il ne faut pas confondre cette affectation et l'instruction:

établir ‘a’ [2,1] = { description ‘c’ , n+1 }

qui s'écrit :

établir (composant (2+g *1, a), q (description (c) ,atent(n+1)))

Cette dernière associe au nom * iri * la liste indiquée.

copie est simplement traduite par :

```
entier procédure copie (x) ;
       valeur x ; entier x ;
       copie : = si x = nil alors nil sinon
              cons (car (x), copie (cdr (x))) ;
```

Si ‘a’ et i sont respectivement le nom et la variable entière considérés, l'instruction parcourant est équivalente à :

```
i : = 0;
pour  i := i+1 tant que composant (i,a) ≠ nil faire <instruction>
```

Les deux relations présence de et eq sont aisément traitées. La première devient une procédure analogue à la fonction LISP "member". La seconde est remplacée par l'égalité puisque les noms sont représentés par des atomes.

Autres éléments.

Ces éléments sont ceux qui n'ont pas été utilisés dans l'application du chapitre II, essentiellement parce que nous utilisons directement l'arbre syntaxique et que le langage objet est symbolique.

Il s'agit des entiers nombre et compteur, du booléen terminal, des <profils>, de l'opérateur moins et de l'élément d'instructions objet "."

terminal, du fait de la représentation en liste de l'arbre fourni, est équivalente à la fonction LISP "atom".

Un <profil> peut être considéré comme une expression booléenne dont les termes sont des booléens terminal.

Exemple : soit le sous-chapitre :

e l o l l : < > ;

Il s'écrit :

si atom (cad (1)) et atom (cad (3)) et atom (cad (4)) alors
début < > fin ;

Le compteur propre à chaque segment est une valeur contenue dans la liste correspondant à ce segment. Il est incrementé d'autant de 1 qu'une instruction objet générée sur ce segment contient de "." ou du contenu du compteur d'un segment lorsqu'il subit une inclusion. Compteur est une procédure fournissant cette valeur.

Une entière procédure "moins" fournit à partir de la pile associée à une variable locale et d'une valeur entière n l'élément de pile situé à n positions au-dessous du sommet de la pile.

Exemple :

générer "'si' x = " k sur pl moins 2

s'écrit :

```
générer (cons ( atome ("'si'x="), atent (k)),
              local (moins (k,2)) ;
```

II.2.3. Exemples d'applications -

II. 2.3.1. Expressions arithmétiques simples -

Langage source

La grammaire context-free fournie à l'analyseur est la suivante :

⟨affectation⟩ ::= ⟨variable⟩ := ⟨expression⟩
⟨expression⟩ ::= ⟨expression⟩ + ⟨terme⟩
⟨terme⟩ ::= ⟨terme⟩ * ⟨primaire⟩
⟨primaire⟩ ::= (⟨expression⟩) | ⟨variable⟩
⟨variable⟩ ::= A | B | C .... | Z

Nous utilisons ensuite un dispositif décrit en [Co] pour transformer un arbre syntaxique transmis par l'analyseur en un arbre dit "de dépendance".

Brièvement, cette transformation s'effectue à l'aide de listes d'entiers positifs attachées à chaque règle de la grammaire syntaxique - Soit la liste d'entiers positifs :

$$\ell\varphi = (n_0, n_1, \ldots, n_i, \ldots, n_m), \forall i \quad n_i < \text{longueur}(\varphi)$$

attachée à la règle :

$$A \longrightarrow \varphi$$

Une correspondance est établie entre le $n_i$ème élément de φ compté à partir de la gauche et le ième élément de la liste d'entiers: nous dirons que le second dénote le premier. L'arbre de dépendance déduit de la règle est formé suivant le schéma :

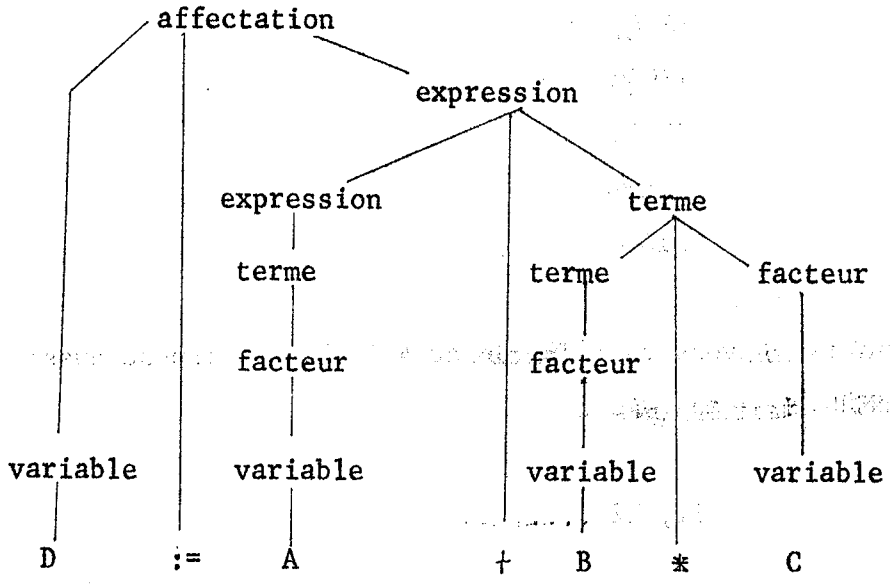
- l'élément dénoté par no est son noeud
- Sa ieme branche est l'élément dénoté par n_i s'il est terminal ; si cet élément est une métavariable B, la ieme branche est l'arbre obtenu par le même processus appliqué à la règle $B \longrightarrow \psi$ et à sa liste d'entiers $\ell\psi$

Exemple : Aux règles de la grammaire considérée, nous associons les listes d'entiers :

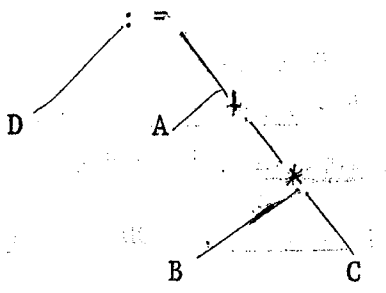
⟨affectation⟩	::= ⟨variable⟩ := ⟨expression⟩	(213)
⟨expression⟩	::= ⟨terme⟩	(1)
⟨expression⟩	::= ⟨expression⟩ + terme⟩	(213)
⟨terme⟩	::= ⟨facteur⟩	(1)
⟨terme⟩	::= ⟨terme⟩ * facteur⟩	(213)
⟨variable⟩	::= A | B | .... Z	(1)

Ainsi l'arbre syntaxique de la chaîne

D : = A + B * C :



est transformé en l'arbre :



Langage objet -

Il est composé des instructions :

```
CLA   M
LDQ   M
ADD   M
MULT  M
STO   M
```

dont nous avons vu l'action au § II.1. On dispose aussi de N mémoires auxiliaires appelées

T1, T2 ..... TN

Programme de génération

Un programme simple P1 traduisant l'arbre de dépendance d'une affectation en langage objet pourrait être ::

```
début segment po ; entier k ; k :=0 ;

    e+ :
       eoo : générer . "CLA  " * i *
                     . "ADD  " * ri *  sur  po ;
       e01 : traiter * i * ; générer . "ADD " * ri * sur po ;
       e01 : traiter * ri * ; générer . "ADD  " * i * sur po ;
       e11 : traiter * i * ;
             k := k+1 ; générer . "STO  T" k sur po ;
             traiter  * ri *
             générer. "ADD  T" k  sur  po ;  k:=k-1 #
```

```
e * :

    eoo : générer . "LDQ  " * i *
                  . "MULT" * ri * sur  po ;

    eol : traiter * i * ; générer . "LDQ " * i *
                                  . "MULT  " * ri * sur  po ;

    e10 : traiter * ri * ; générer . "LDQ  " * ri *
                                  . "MULT  " * i * sur po ;

    e 11 : traiter * i * ;
           k : = k + 1  ; générer . "STO  T" k  sur  po ;
           traiter * ri * ;
           générer . "LDQ  T" k
                   . "MULT  T" k  sur  po ;  k:= k-1 #

* e : = :

    eoo : générer . "CLA  " * ri * . "STO  " * i * sur  po

    eol : traiter * ri * ;
          générer . "STO  " * i * sur po

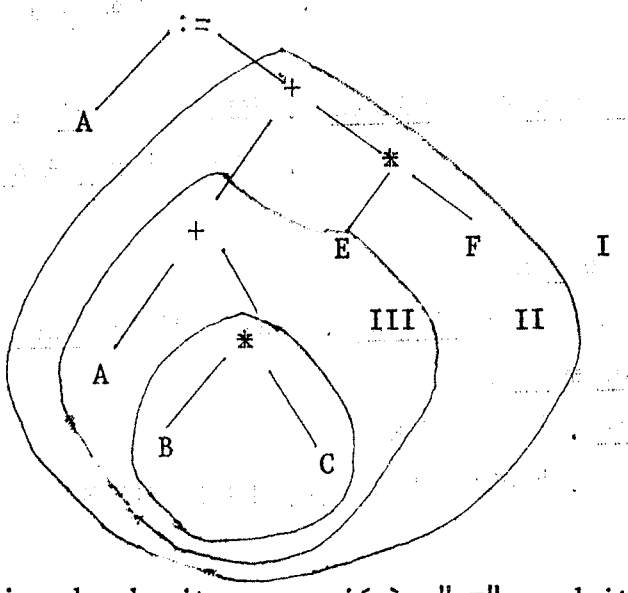
fin
```

Examinons le fonctionnement de ce programme à l'aide d'un exemple .

Soit la chaîne d'entrée

A : = A + B * C + E * F

L'arbre de dépendance obtenu est le suivant :



L'action du chapitre associé à ":=" conduit à traiter l'arbre I, puis successivement les arbres II et III par l'intermédiaire des sous-chapitres ell et eol du chapitre "*". C'est à ce niveau qu'est généré dans le sous-chapitre e o o de "* " :

```
LDQ  B
MPY  C
```

sur le segment po.

Le retour au sous-chapitre correspondant à II produit ensuite :

```
ADD A
```

La première branche de I a été examinée - Un rangement dans une mémoire auxiliaire devient nécessaire puisque la seconde branche n'est pas terminale. Ces mémoires sont gérées en pile, c'est à dire qu'une kième mémoire n'est demandée que si les contenus des mémoires d'indices inférieures à k sont effectivement intéresssants pour la suite du programme objet. L'ordre engendré est ici :

```
STO  T1
```

La seconde branche de I se traite alors comme l'arbre III ; il en résulte :

```
LDQ  E
MPY  F
```

Pour parachever l'examen de I, il reste à générer

```
ADD  T1
```

L'indice k est décrémenté de 1, signalant ainsi que la mémoire t1 est à nouveau disponible.

Enfin, le retour au chapitre ":=" produit

```
STO  A
```

On remarque que le traitement d'un arbre fournit toujours un programme objet tel que si ce dernier était exécuté, son résultat serait contenu dans l'accumulateur Acc.

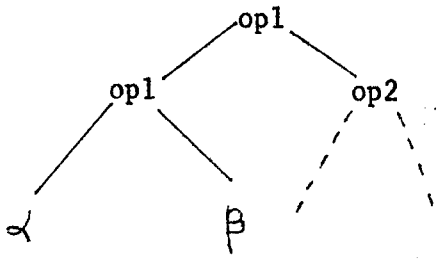
Une première amélioration du programme objet concerne le nombre des mémoires auxiliaires utilisées. On remarque en effet qu'il n'est pas nécessaire de sauvegarder le résultat d'une opération entre deux variables si ce résultat est opérande d'un même type d'opération. Cette dernière propriété n'est pas exploitée par Pl. Il suffit dans ce

cas d'intervertir l'ordre d'évaluation des opérandes dans la dernière opération citée. On obtient un programme P effectuant cette optimisation en modifiant les sous-chapitres e11 des chapitres e+ et e* . Ils deviennent :

```
e + :
    :
    e11 :
          si * i * eq '+' et
              terminal * ii * et terminal * iri * alors
                 début
                    traiter * ri * ;
                      générer . "ADD " * ii *
                              . "ADD " * iri * sur po
                 fin
          sinon début <corps du sous-chapitre correspondant
dans P1 > fin #

e *:
    :
    e11 :
          corps déduit de celui de e11 dans e+ en remplaçant +
          par '*' et le code ADD par MUL> #
```

Le programme P2 distingue donc les arbres de la forme :



où op1 et op2 sont des opérateurs et α et β des variables. Il prend d'abord en considération l'arbre de noeud op2 et ensuite produit deux opérations op1 sur α et β .

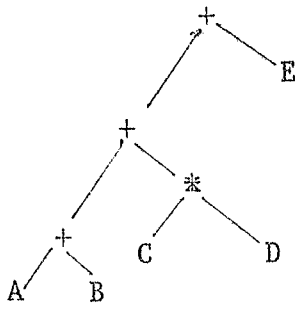
On note qu'un examen du "frère" du "fils" du noeud considéré (c'est à dire de la branche droite de l'arbre) pourrait aussi conduire à une optimisation du même genre. On ne l'envisage pas ici parce qu'étant donné la grammaire et la transformation effectuée sur l'arbre syntaxique, ceci reviendrait à chercher à améliorer le programme objet d'expressions telles que

$$\ldots W_o \ \text{op1} \ (W_1 \ \text{op1} W_2) \ \ldots$$

où W_o , W_1 et W_2 sont des expressions (les parenthèses sont inutiles).

Exemple :

L'arbre



produit à partir de l'expression

A + B + C * D + E

fournit le programme objet :

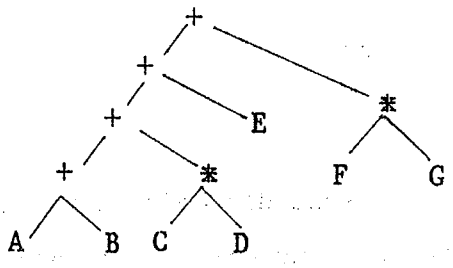
```
LDQ  C
MPY  D
ADD  A
ADD  B
ADD  C
```

dans lequel aucune mémoire auxiliaire n'est utilisée.

Nous envisageons maintenant d'améliorer un programme objet en fonction de l'instruction RAD M supposée introduite - Rappelons (Cf § II 1) qu'elle est équivalente à la séquence :

```
ADD  M
STO  M
```

Une optimisation possible concerne les opérations sur les mémoires auxiliaires. On remarque en effet que si le résultat de l'addition du contenu d'une mémoire auxiliaire et de l'accumulateur doit être sauvegardé, il le sera dans cette même mémoire auxiliaire. L'instruction RAD peut être employée lorsque ce résultat est utilisé pour une nouvelle addition. Considérant, par exemple, la configuration :



résultat du traitement de l'expression

... A * B + C * D + E + F * G

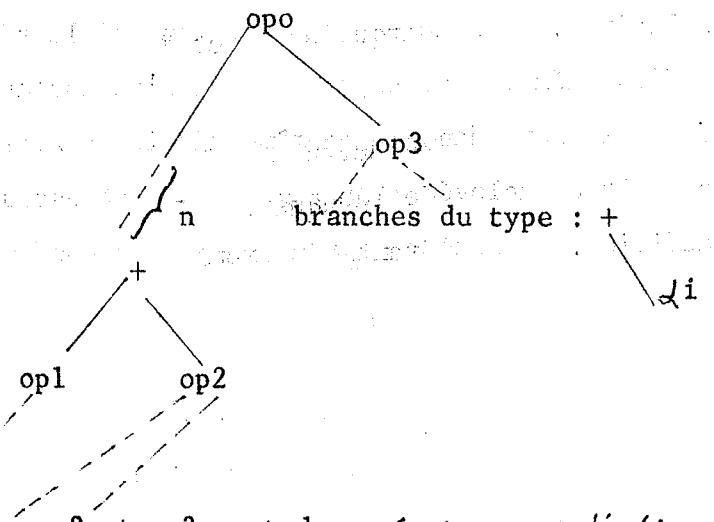
le programme P2 fournit le programme objet :

```
LDQ  A
MPY  B
STO  Ti
LDQ  C
MPY  D
ADD  Ti   }
ADD  E    } I
STO  Ti   }
LDQ  F
MPY  G
ADD  Ti
```

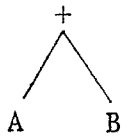
dans lequel la séquence I pourrait être remplacée par :

```
ADD  E
RAD  Ti
```

Pour effectuer cette optimisation il faut distinguer les arbres du type :



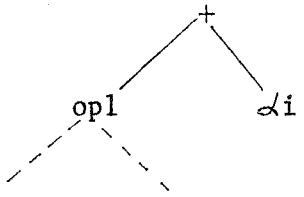
où opo, op1, op2 et op3 sont des opérateurs et α_i ($i \geq o$) des terminaux, tels que l'arbre de noeud op1 ne soit pas de la forme :



Cette dernière contrainte intervient du fait que la première optimisation est prioritaire.

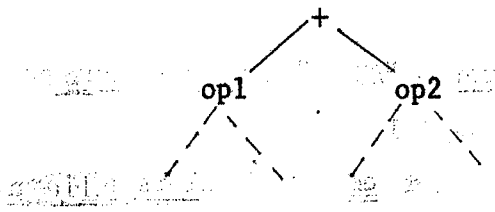
La reconnaissance a lieu dans les sous-chapitres e11 de e+ et de e* en procédant ainsi :

- tant que les descendants "gauche" de l'arbre considéré sont du type



une addition est générée sur un nouveau segment pa ; l'examen des descendants "gauche" ne se poursuit que si op1 est l'opérateur +.

- Au cas où le descendant "gauche" possède une configuration



satisfaisante, les arbres dépendants de op1 et op2 sont traités -

- Si l'arbre initial s'avère impropre à l'optimisation le dernier arbre examiné est traité.-

- Enfin, dans tous les cas, les instructions produites sur pa sont replacées convenablement sur le segment po.

Le sous-chapitre e11 modifié peut s'écrire :

```
e11 : si
        <optimisation introduite par P2>
      sinon
      si *i* eq '+' alors

        début
            nouveau m ; m : = 1 ; nouveau pa ; annuler pa ;
            f : si ¬ terminal *iim * et ¬ terminal *riim * et
                ¬ (terminal *iiim * et terminal *riiim * et
                                *iim * eq '+' ) alors

                  début
                     traiter *iim* ;
                     k : = k + 1 ; générer."STO  T""k sur po ;
                     traiter * riim *;;
                     générer. "RAD  T" k sur pa ; k : = k - 1
                  fin
```

```
                sinon si terminal * riim * alors
                        début
                          générer - "ADD  " * riim * sur pa ;
                          m : = m + 1 ;
                          si * im * eq '+' alors alléra f
                    sinon  traiter * im *
            fin ;

    prévoir pa dans po ; inclure pa dans po ;
    ancien  pa ; ancien m

fin

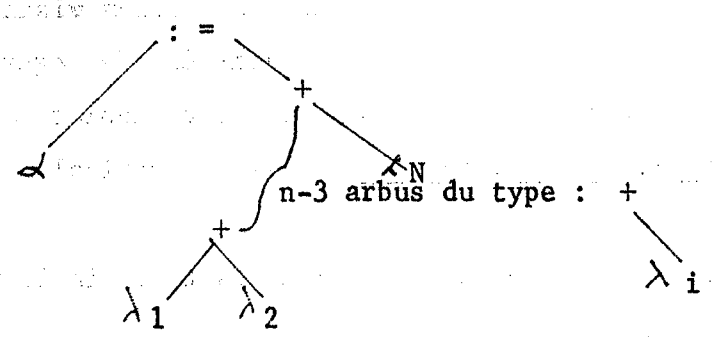
    sinon début <corps du sous-chapitre correspondant dans Pl> fin #
```

La variable entière m et le segment pa sont déclarés locaux puisqu'il est possible que les expressions soient imbriquées.

On remarque que les branches droites qui pourraient amener à une optimisation ne sont pas examinées : ceci, pour la même raison que précédemment.

Signalons qu'il est aussi possible d'optimiser à l'aide de l'instruction RAD l'affectation proprement dite (Cf § II.2.1) : on peut l'employer lorsque l'expression est le résultat d'une addition dont la partie gauche est l'un des opérandes - -

Il faut alors distinguer les configurations du type :



où λ_i ($i > 1$) est une variable ou un opérande et telles qu'il existe au moins un λ_i identique à α .

II.2.3.2. Shéma simple pour la définition de nouveaux opérateurs.

On trouve dans [VW2] une proposition visant à donner la possibilité au programmeur de déclarer lui-même des opérateurs (unaires ou binaires) ayant comme opérandes soit des variables de types de base (entier, reel, booleen, caractere) soit des variables de types nouveaux (cf § III.1).
Ces opérateurs sont définis à l'aide des trois indications suivantes :

- une priorité
- le type des opérandes
- une expression (cf [VW 2]) qui à partir de ces opérandes fournit une valeur d'un type donné.

Exemple :

```
struct complexe = (reel pr, pi) ;
priorité + = 5(*) ;
opérateur complexe + = (complexe a, complexe b)
                    expr complexe (pr de a + pr de b,
                                   pi de a + pi de b) ;
```

Ayant déclaré le nouveau type complexe, on définit l'opérateur "+" sur des variables de ce type. La valeur fournie est une valeur complexe dont la partie réelle est la somme des parties réelles des opérandes et la partie imaginaire la somme des parties imaginaires.
Ainsi, on peut maintenant écrire :

```
complexe x, y, z ;
        :
        :
x : = y + z  ;
```

(*) En fait, la notation "priorité + = 5" n'est pas prévue dans [V W 2], mais apparaîtra certainement dans le rapport final d'Algol X (Réunion du groupe Algol AFIRO à La Gaude, Septembre 1967).

II.2.3.2.1.

Nous nous posons le problème suivant : étant donné des expressions parenthésables contenant de tels opérateurs binaires, comment les traduire en un programme en langage interpréteur. Nous entendons par langage interprèteur un langage formé d'instructions pouvant opérer sur des opérandes situés au sommet d'une pile.
L'intérêt d'utiliser une analyse syntaxique dans ce cas est manifeste. En effet, elle permet de déterminer aisément les opérandes d'un opérateur si sa priorité est précisée : conjointement, la hiérarchie dans laquelle les opérations doivent être effectuées est spécifiée.

Exemple :
Si "α" est un opérateur sur des entiers, de priorité inférieure à la multiplication, l'expression

a * b α c

sera interprètée :

(a * b) α c

Pour pouvoir employer l'analyse syntaxique à la résolution de ce problème, il faut donc disposer d'un mécanisme permettant d'introduire, en cours de compilation, des règles dans la grammaire. Nous supposons ici qu'à l'aide de l'ordre

introduire regle <regle>

nous avons la possibilité, dans un programme de génération, de fournir à la grammaire des règles lexicographiques (la syntaxe des règles est analogue à celle des instructions objets).
Signalons que l'usage d'un tel procèdé est délicat : en effet, il faut se garder d'introduire des règles rendant la grammaire ambigue.

Description syntaxique.

Soit N la priorité maximale. La syntaxe des expressions fournie à l'analyseur est la suivante :

< expression > : : = < t_1 >

< t_1 > : : = < t_1 > < op 1 > < t_2 > | < t_2 >

⋮

< ti > : : = < ti > < opi > < ti_{+1} > | < T_{i+1} >

< t_{N+1} > : : = (< + 1 >) |

< variable > | < parametre > |

< nombre entier >

avec par exemple :

< op 1 > : : = ⊃

< op 2 > : : = ∨

< op 3 > : : = ∧

< op 4 > : : = ≠

< op 5 > : : = +

< op 6 > : : : = *

La syntaxe de la déclaration d'un opérateur binaire pourrait être la suivante :

< déclaration-d-opérateur> : : = < déclaration de priorité > #
< déclaration d'opération >

< déclaration de priorité > : : = priorité <opérateur> = <nombre entier >

< déclaration d'opération > : : =

opérateur <type> <opérateur> = (<spécification d'opérandes>)

expr < t_1 >

< spécification d'opérandes > : : = < type > < paramètre > ,
< type > < paramètre >

On suppose que la relation entre les déclarations ou spécifications de variables et leur apparition dans les expressions a été effectuée (éventuellement, dans une phase AMC) de telle sorte qu'on puisse considérer que la grammaire comporte, par exemple, les règles lexicographiques suivantes :

< variable > : : = a | b
< paramètre > : : = x | y
< type > : : = entier | logique | complexe
< opérateur > : : = -

De même, on suppose que les nombres entiers présents dans les expressions ont été reconnus et qu'ils sont désignés par <nombre entier>.

Programme de génération.

Ce programme utilise directement l'arbre syntaxique. Il ne décrit pas les actions entreprises pour traiter les déclarations de variables et de types. Les chapitres consacrés à ces éléments ont eu pour effet:

- d'adjoindre à chaque variable une description contenant comme premier élément le nom de son type et comme troisième son adresse d'implantation dans la pile à l'exécution : le deuxième pourrait contenir, par exemple, son niveau (cf §III.3.1.)

- d'adjoindre à chaque type une description contenant en premier élément la taille des variables de ce type, c'est à dire l'encombrement en mémoire de leur représentation.

Les opérateurs de base sont munis dans la partie initialisation d'une description comportant le type des opérandes, le type résultant et le programme objet correspondant.

Exemple :

nouveau p; générer. "multiplication" sur p ;
établir '*' = {'entier' , 'entier' , 'entier' , p}
nouveau p ; générer. "Sup" sur p ;
établir '<' = {'entier' , 'entier' , 'booleen' , p}

Le programme s'écrit shématiquement :

début
local segment po , p , ps ;
< < initialisation >
* e déclaration de priorité :
introduction règle " <op " * i r 3 i*" > : : = " *iri * #
* e déclaration d'opération :
nouveau p ; nouveau po ;
prévoir po dans p ;
générer. "initialiser lo" * i2 r5 i * [1] + ir 3ir5i * [1] sur po ;
établir * i r i r 5 i * = {o, * i 2r5 i *} ;
établir * i r 4 i r 5 i * = {* i 2 r 5 i * [1] , * i r 3 i r 5 i * ;
traiter * r 8 i * ;
inclure po dans p ; ancien po ;
générer. "affectation" * i r i * [1]
. "désempiler" * i 2 r 5 i * [1] + * i r 3 i r 5 i * -
* i r i * [1] sur p ;
établir * i r 2 i * = {* i 2 r 5 i *,* i r 3 i r 5 i *, * i r i * ,
p} #

```
* e expression :
        traiter * i * #

e + 1 :
  .
  .
  .
e ti  :
  .
  .
  .
e t_N :

        traiter * i * ;
        si nombre = 3 alors
           début
             si ¬ (⌜type considéré⌝ [1] eq * r i * [13) alors
                 message "erreur type gauche" ;

        traiter * r r i * ;
        si ¬ (⌜type considéré⌝ [1] eq * r i * {2} alors
                 message "erreur type droit " ;
                 prévoir * r i [4] * dans po ;
                 inclure * r i [4] * dans po ;
                 établir ⌜type considéré⌝ = { * r i * [3]

           fin #

e t_N+1 :
        si nombre = 3 alors traiter * r i * sinon
                 début
                   si * i * eq ⌜paramètre⌝ alors
                        générer. "valeur paramètre" * i i * [1] sur po
                   sinon générer. "valeur variable" * i i * [3] sur po

                 fin #
fin
```

Décrivons maintenant le fonctionnement de ce programme à l'aide d'un exemple simple. On désire définir la "soustraction propre" sur des opérandes entiers (toujours positifs) et traiter l'expression :

a * 3 $\dot{-}$ b

l'opérateur $\dot{-}$ est défini par :

priorité $\dot{-}$ = 5 ;

opérateur entier $\dot{-}$ = (entier x, entier y)
expr si x > y alors x-y sinon o ;

Le langage objet est composé des instructions suivantes :

- empiler x

La valeur x est placée au sommet de la pile

- valeur variable x

x est une adresse de pile. Le contenu de l'adresse est placé au sommet de la pile

- valeur paramètre x

La valeur x est additionnée à la valeur de lo (cf initialiser lo). La contence de l'adresse ainsi indiquée est placée au sommet de la pile.

- initialiser lo x

L'adresse de pile située à x positions au dessous du sommet de la pile est affectée à lo. Cette adresse est placée au sommet de la pile.

- sup

si le contenu de la position immédiatement en arrière du sommet de la pile est supérieure au contenu du sommet il y a saut d'une instruction sinon le programme se déroule séquentiellement. Les deux premières positions de la pile sont détruites.

- soustraction

soustraction usuelle entre le second et le premier élément élément de la pile. Le résultat figure à la place du premier opérande.

- multiplication

même processus que la précédente instruction appliquée à la multiplication.

- saut x

transfert de contrôle à la xième instruction suivant ou précédant celle-ci selon que x est positif ou négatif.

- affectation x

affectation de x éléments supérieurs de la pile à l'adresse indiquée par le contenu de l'élément immédiatement en arrière des précédents. Ce dernier devient le premier élément disponible de la pile.

- désempiler x

les x éléments supérieurs de la pile sont détruits. Si x est négatif ou nul , cette instruction est sans effet.

Examinons dans ce cas les actions de chaque chapitre du programme de génération.

Le chapitre consacré à la déclaration d'opération

- génère sur un nouveau segment po l'instruction initialiser lo 2

On a supposé ici que la taille d'un entier est 1

- associe aux noms "x" et "y" les descriptions :

$$\{o, \langle \text{entier} \rangle\}$$

$$\{1, \langle \text{entier} \rangle\}$$

- déclenche la production sur le segment po. du programme objet traduction de :

si x > y alors x - y sinon o

Etant donnés les nouveaux éléments que nous avons introduits en considérant cet exemple d'opérateur et pour en faciliter la présentation, nous n'exposons pas ici comment s'effectue cette production. Nous en verrons le détail au § II.2.3.2.2.

Ce programme objet est le suivant :

valeur paramètre 0
valeur paramètre 1
sup
saut 4
valeur paramètre 0
valeur paramètre 1
soustraction
saut 1
empiler 0
affectation 1
désempiler 1

- associe au nom "___" la description :

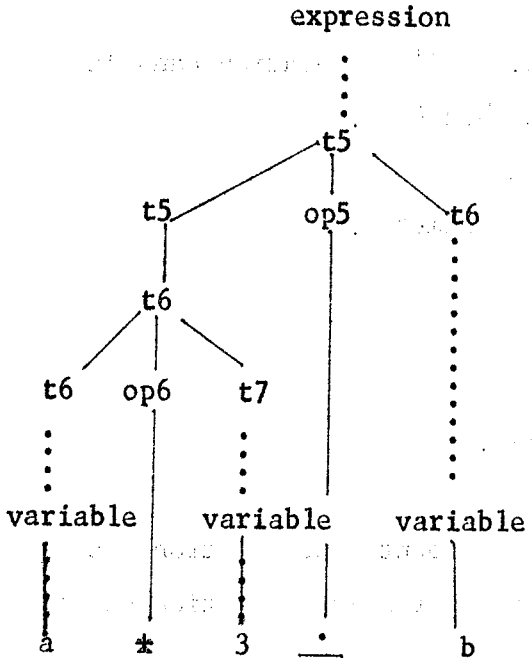
{⌜entier⌝ , ⌜entier⌝ , ⌜entier⌝ , p}.

Le chapitre consacré à la déclaration de priorité introduit la règle :

<op 5 > : : = ___

dans la grammaire

L'arbre syntaxique de l'expression figurant dans le texte est de la forme :



Le chapitre consacré à t5 traite cet arbre et successivement,

- provoque la génération sur le segment po des programmes objets correspndants aux deux opérandes de la soustraction propre, c'est à dire :

  valeur variable a
  empiler 3
  multiplication
  valeur variable b

- vérifie que chacun des opérandes est de type convenable.
- ajoute à po le programme objet associé à "-" .
- signale que le résultat de l'opération est de type entier.

Le chapitre consacré à t_{N+1}

- génère, le cas échant, sur le segment po l'implantation dans la pile de la valeur de la variable ou du paramètre.

- signale le type de la variable ou du paramètre.

-II- 2.3.2.2. Expression conditionnelle.

En relation avec le paragraphe précédent, nous montrons ici comment obtenir le programme objet correspondant à la définition de l'opérateur $\frac{\cdot}{-}$.

A cette fin, nous devons introduire des expressions conditionnelles. Nous supposons donc que la grammaire contient, par exemple, la règle :

< + 1 > : : = si < + 1 > alors < + 1 > sinon < + 1 >

Le chapitre associé à t1 s'écrira alors :

e t1 :

si nombre = 1 alors traiter * i * sinon

```
début
        traiter *ri*    ;
        si ¬ ( ‘type considéré’ [1] eq ‘booléen’ ) alors
        message "proposition si non conforme" ;
        générer "saut" sur po ;
        nouveau ps ; prévoir ps dans po ;
        nouveau po ; prévoir po dans po moins 1 ;
        traiter * r3 i * ;
        établir ‘tg’ = { ‘type considéré’ [1] , description ‘tg’ }
        générer compteur po + 1 sur ps ;
        inclure po dans po moins 1 ; inclure ps dans po moins 1 ;
        annuler po ; annuler ps ;
        générer "saut" sur po moins 1 ;
        prévoir po dans po moins 1 ; prévoir ps dans po moins 1
fin #

traiter * r5i * ;
si ¬ (‘type considéré’ [1] eq ‘tg’ [1] ) alors
   message "types non conformes dans expression conditionnelle" ;
supprimer 1 de ‘tg’ ;
inclure po dans po moins 1 ;
générer compteur po sur p  ; ancien po ;
inclure ps dans po ; ancien ps //#
```

Lorsqu'on considère l'expression

```
si  x > alors x-y sinon o
```

le rôle de ce chapitre est, successivement

- d'évaluer la première expression, c'est à dire

```
x > y
```

et de générer sur le segment po :

```
valeur paramètre o
valeur paramètre 1
sup
```

De plus, on vérifie que cette expression est de type booléen.

- de générer une instruction "saut" incomplète sur po. Il est prévu que le complèment sera contenu sur un nouveau segment ps : en effet, les expressions conditionnelles peuvent être imbriquées.

- de produire le résultat de l'examen de

x - y

c'est à dire :

```
valeur paramètre o
valeur paramètre
soustraction
```

sur un nouveau segment po. Ce segment est créé uniquement pour qu'on puisse disposer ultérieurement du contenu de son compteur.
L'instruction "saut" précédemment générée est complétée par la valeur de compteur po + 1. Les segments po et ps sont inclus convenablement dans l'ancien segment po.

- de retenir le type de l'expression traitée, c'est à dire entier

- de générer une instruction "saut" sur l'ancien segment po en faisant la même prédiction que précédemment.

- d'évaluer la troisième expression, ce qui a pour effet de générer

```
empiler 0
```

sur le segment po

- de vérifier que les types des deuxièmes et troisièmes expressions sont identiques.

- Enfin, d'effectuer des inclusions analogues aux précédentes et de reprendre en compte les anciens segments po et ps.

Remarque 1.

Dans le cas où un nouvel opérateur serait représenté par un nom indentique à celui d'un opérateur déjà existant, l'implantation présentée ici est en défaut à moins qu'ils ne possèdent la même priorité (la description attachée à l'opérateur permettrait de distinguer les deux cas). En effet, on ne pourrait alors les différencier que par le type de leurs opérandes ce dont il n'est pas possible de rendre compte dans notre grammaire des expressions : l'introduction d'une nouvelle règle provoquerait une ambiguïté. Il faudrait donc créer d'autres règles, ce qui est envisageable.

Remarque 2.

Nous avons traité ici le corps de la définition d'un nouvel opérateur comme le corps d'une procédure possèdant des paramètres appelés par valeur. Ceci est du au genre de langage objet que nous avons adopté. En présence d'un langage objet non interpréteur, on pourrait traiter cette définition comme une macro-instruction.

II.2.4. Environnement.

II.2.4.1. Relation avec une phase AMC.

Deux tâches principales sont dévolues à la première phase d'un compilateur, quand elle existe; ce sont :

- obtenir de la chaîne initiale, constituée par le programme source une transformée acceptable par la phase suivante, que celle-ci soit du type ASG ou de nouveau du type AMC.

- détecter, au choix de l'utilisateur, des erreurs que celui-ci juge importantes.

Une condition impérative "d'acceptabilité" de cette transformée par une phase ASG est évidemment que celle-ci soit susceptible d'être analysée syntaxiquement à l'aide de la grammaire context-free fournie à cette phase.
On remarque que la conception de la grammaire peut être grandement facilitée si l'on admet que l'analyse syntaxique opère sur une transformée de la chaîne initiale.

Considérant par exemple, la grammaire d'ALGOL 60 telle qu'elle est présentée dans [Nau] , on constate qu'elle est ambigue au niveau des variables : ceci empêche assurément son exploitation par une phase ASG. Si toutefois celle-ci n'admet comme chaînes initiales que des chaînes dans lesquelles la correspondance entre une variable et ses caractéristiques (indiquées dans la déclaration de la variable) a été établie, de légères modifications dans la grammaire permettront de l'utiliser.

Illustronsceci à l'aide d'un exemple simple :

Si nous trouvons dans un programme source ALGOL 60 l'instruction :

a : = b

la grammaire présentée dans [Nau] en fournit plusieurs interprétations possibles : instruction d'affectation entre deux entiers ou entre deux booléens ....etc.

Sachant que a et b sont des variables entières, une transformée livrée à l'analyse par une phase AMC serait :

variable-entière : = variable-entière

la règle

<primaire arithmétique> : : = <variable>

étant remplacée (en partie) par :

<primaire arithmétique> : : = variable-entière

dans la grammaire.

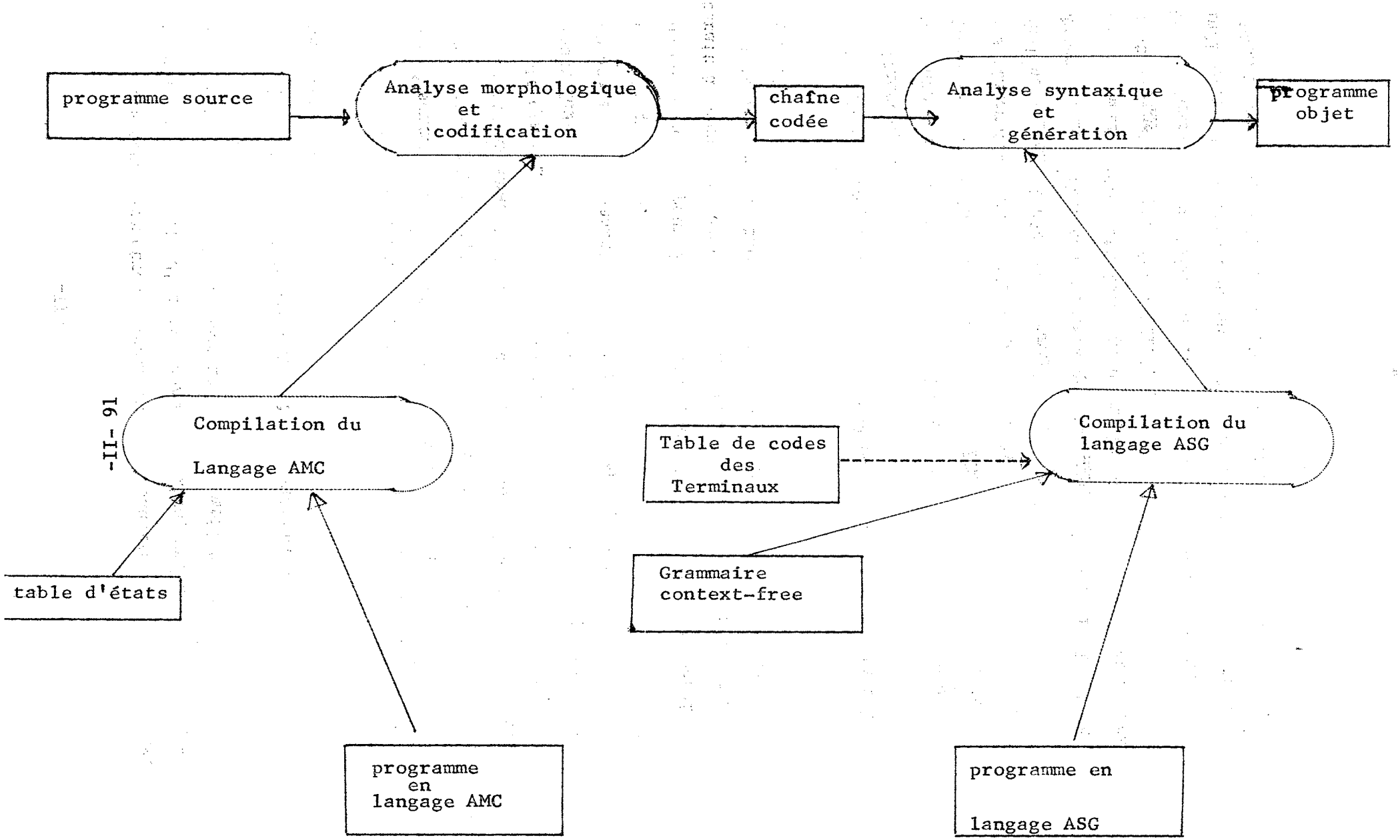
Intuitivement, les terminaux de la grammaire ne seront plus les caractères dont est composée la chaîne initiale mais des codes produits par la phase AMC, d'où le nom de chaîne codée que l'on donne habituellement à la transformée produite par cette phase.

Dans les compilateurs ALGOL 60 actuellement opérationnels et effectuant deux passages [Bou, le P] , ce travail de substitution est réalisé dans le premier. Il se double naturellement de la reconnaissance de la structure de bloc et de la détection des erreurs dues à sa violation.

D'autres transformations telles que codification des symboles de base, suppression de déclaration....etc facilitant la tâche de la phase suivante y sont en général accomplies.
On note d'ailleurs que certaines opérations peuvent être effectuées, au choix du programmeur-compiliste, dans l'une ou l'autre des phases. Ainsi, le calcul de la taille d'une variable de nouveau type décrit au §III.3.1. aurait-il pu être réalisé dans une phase AMC.

Un langage spécialisé dans l'écriture de la phase AMC a été conçu au Laboratoire de Calcul de l'Université de Grenoble [Jor] .
Brièvement, la partie reconnaissance, n'opérant que dans contexte restreint à l'intérieur de la chaîne donnée, est accomplie à l'aide d'un automate d'états finis : la description de cet automate, c'est à dire sa table d'états est introduite en paramètre. La partie production est réalisée par des séquences d'opérations élémentaires judicieusement distinguées. Le couplage reconnaissance-production s'effectue en associant une telle séquence à chaque état de l'automate.
Un problème important se pose lorsqu'on doit définir les communications à assurer entre les différentes phases d'un "système" de compilation.

Considérons par exemple, le diagramme suivant représentant un cas particulier fréquent, à savoir celui où une phase AMC unique est suivie d'une phase ASG : la ligne horizontale, au sommet du diagramme représente les transformations subies par le programme source au cours d'une compilation, alors que les lignes verticales gauches et droites concrétisent respectivement l'obtention en langage machine des deux phases utilisées pendant cette compilation.


programme source
Analyse morphologique et codification
chaîne codée
Analyse syntaxique et génération
programme objet
Compilation du Langage AMC
table d'états
programme en langage AMC
Table de codes des Terminaux
Grammaire context-free
Compilation du langage ASG
programme en langage ASG

Les deux paramètres principaux de reconnaissance de ces phases, c'est à dire une table d'états pour la première, une grammaire syntaxique pour la seconde sont fournis à leurs compilateurs.

La liaison à réaliser entre AMC et ASG comporte les trois aspects suivants

1) Comment les éléments terminaux de la grammaire doivent-ils être représentés de façon à correspondre à ceux de la chaîne codée ?
Dans une première approche, on peut envisager de laisser à l'utilisateur le soin de coder ces éléments terminaux dans la grammaire. Il sait en effet quelle en est la représentation donnée par la phase AMC.
Toutefois, dans le cas où ces codes seraient modifiés, il faudrait changer la représentation des terminaux pour toutes leurs occurences dans la grammaire.
Un moyen souple pour éviter de corriger ainsi la grammaire consisterait à fournir au compilateur ASG un nouveau paramètre : une table associant à chaque terminal écrit sous forme symbolique un code imposé.

2) De quelle façon l'analyseur de la phase ASG doit-il examiner la chaîne codée ?
On se rend compte que ceci est lié à la configuration de chaque élément de la chaîne codée. Il peut se faire que seule une partie des informations recueillies par la phase AMC sur un élément de la chaîne initiale intéresse l'analyseur.
Exemple : au III, nous verrons que le niveau d'une variable n'est pas une information utilisée par un analyseur employant la grammaire syntaxique du langage (cf § III.1.).
Une solution à ce problème serait de donner au compilateur ASG une indication sur le format unique adopté pour représenter dans tout élément de chaîne codée la partie que doit considérer l'analyseur.

3) Comment la partie génération peut-elle accéder aux informations contenues dans la chaîne codée et attachées à chaque terminal ?

Cette question pourrait être résolue en introduisant dans le langage de génération des opérations nécessaires au traitement d'unités binaires d'information. Une autre solution serait d'imposer un format standard à la chaîne codée : par exemple, tout élément de chaîne codée serait composé d'un "bit pattern" ou signe distinctif nécessaire pour l'analyse et d'un indicateur sur la liste des informations associées à cet élément. Cette liste formerait une description et les informations contenues seraient accessibles au moyen de composants (cf II.2.1.).
Ceci résoudrait d'ailleurs également la seconde question.
Un autre problème relatif à l'assemblage de phases AMC et ASG est celui posé par l'alternance de deux telles phases, généralisant le phénomène déjà constaté à l'intérieur de chacune d'elles entre les parties reconnaissance et production.
Une telle possibilité serait en effet intéressante dans le cas où l'on voudrait écrire à l'aide de ces langages un compilateur entièrement conversationnel.
Une solution à ce problème pourrait être d'introduire dans le langage AMC un ordre indiquant un transfert de contrôle à la phase ASG.
Dans ce cas, l'écriture de la grammaire fournie à l'analyseur peut se révéler délicate. Par exemple, en ce qui concerne ALGOL 60, il faudrait admettre un identificateur comme terminal, sans précision sur son type : en effet, il est possible par exemple que la déclaration d'une variable intervienne lexicographiquement après qu'elle ait été employée.
Nous présentons simplement ici deux écritures possibles des grammaires des expressions arithmétiques et booléennes admettant "identificateur" comme terminal.

La première est celle employée dans l'exemple du § II.2.3.2. Elle consiste shématiquement à ne pas distinguer les expressions arithmétiques des expressions booléennes, laissant ce travail à la partie productionde la phase ASG : l'écriture en est très simple, mais l'analyse demeure grossière.

Exemple : un texte source tel que :

.....si a+b alors x sinon y.....

sera accepté par l'analyse (et réfusé ensuite par la partie production).

La seconde est un peu plus élaborée : elle tient compte de la présence d'opérateurs pour distinguer les expressions en question.

Les notations employées sont les suivantes :

E, T, F sont des non-terminaux désignant respectivement des expressions, des termes et des facteurs arithmétiques contenant ou non des opérateurs.

- $\bar{E}$, $\bar{T}$, $\bar{F}$ sont des non-terminaux désignant respectivement des expressions, des termes et des facteurs arithmétiques contenant des opérateurs.

- enfin, E', T' F' représentent de même des expressions, termes et facteurs mais ne contenant pas d'opérateur.

Nous nous limitons ici aux seuls opérateurs "+" et "*" portant sur des opérandes identifiés par le terminal.i (identificateur).

D'après les définitions précédentes, on peut écrire :

$\bar{E} : : = E + T \mid \bar{T}$

$\bar{T} : : = T * F \mid \bar{F}$

$\bar{F} : : = (\bar{E})$

$E' : : = T'$

$T' : : = F'$ qui peut se résumer en :

$F' : : = i \mid (E')$

$E' : : = (E') \mid i$

$E ::= E' \mid \bar{E}$
$T ::= T' \mid \bar{T}$
$F ::= F' \mid \bar{F}$

qui se résume alors en :

$E ::= E' \mid \bar{E}$
$T ::= E' \mid \bar{T}$
$F ::= E' \mid \bar{F}$

Dans le cas où les identificateurs entre parenthèses ne seraient pas admis, la grammaire peut se simplifier en :

$\bar{E} ::= E + T \mid \bar{T}$
$\bar{T} ::= T * F \mid \bar{F}$
$\bar{F} ::= (\bar{E})$
$T ::= \bar{T} \mid i$
$F ::= \bar{F} \mid i$
$E ::= \bar{E} \mid i$

La grammaire des expressions booléennes (limitées ici aux unions et intersections) s'en déduit en remplaçant les opérateurs "+" et "*" respectivement par "V" et "$\wedge$" et les lettres majuscules par des minuscules.

Une instructions d'affectation pourra se décrire par :

<instruction d'affectation> : : = i : = <partie droite>
<partie droite> : : = $i / \bar{E} / \bar{e}$

Ceci n'entraine pas d'ambiguités car ni $\bar{E}$ ni $\bar{e}$ ne peuvent désigner des identificateurs.
Une proposition si pourra être représentée par :
<proposition si> : : = si e
ce qui permettra à l'analyseur d'interdire des écritures telles que :
....si a + b...
mais autorisera :
....si a' ...
quelque soit le type effectif de l'identificateur a.

II - 2. 4. 2 Relations avec l'analyse.

Nous examinons dans ce paragraphe quelles sont les caractéristiques souhaitables d'un analyseur opérant dans un système tel que nous l'envisageons.

Notre objet n'est pas ici de décrire dans le détail le fonctionnement des algorithmes d'analyse syntaxique les plus connus ; une très bonne étude comparative en a été faite par T.H. GRIFFITHS et S.R. PETRICK dans [G.P.]. Rappelons simplement quelques unes de leurs propriétés.

Leur rôle est de fournir, étant donné une grammaire context-free et une chaîne du texte source, une structure arborescente de cette chaîne : si plusieurs structures sont possibles, la grammaire est ambigue, si aucune n'est constructible la chaîne source n'appartient pas au langage défini par la grammaire : en d'autres termes, il existe une erreur dans cette chaîne.

Suivant que la structure produite est formée à partir de l'axiome ou de la chaîne, on dira que l'algorithme est descendant ou ascendant.

Un algorithme est appelé général s'il s'applique à toute grammaire context-free. Un résultat important concernant les algorithmes généraux nous dit qu'ils procèdent toujours par essais successifs : ou encore, qu'en un état donné de l'analyse, ils peuvent être amenés à choisir entre plusieurs voies de dérivation. Pour déterminer la seule qui soit acceptable (dans le cas où la chaîne est correcte et la grammaire non ambigue) ils doivent les examiner toutes, du moins jusqu'à l'obtention de cette dernière.

La conséquence de ce résultat est immédiate de notre point de vue : inclus dans un système de compilation, un tel analyseur pourrait fournir au programme de génération ou à un processus de transformations d'arbre, une structure qu'un examen plus large de la chaîne lui révèlerait ultérieurement erronée. Dans ces conditions, les étapes successives de la génération devraient être conservées afin qu'on soit en mesure le cas échéant d'en reprendre convenablement la poursuite.
Un tel système , on s'en rend compte serait d'une part difficile à mettre en place, d'autre part peu efficace.

La première des caractéristiques souhaitable pour l'analyseur est donc qu'il ne procède pas par essais successifs.

De nombreuses études ont été faites sur des algorithmes satisfaisants à cette condition : ils ne sont évidemment pas généraux et les grammaires qu'ils admettent sont particulières. Employés dans un système de compilation, ils nécessiteront le plus souvent une intervention humaine pour transformer la grammaire initiale en une grammaire équivalente acceptée par l'algorithme, si tant est qu'il en existe une.

Parmi ces analyseurs, rappelons les trois suivants, qui nous semblent particulièrement intéressants : il s'agit d'un analyseur descendant utilisant une grammaire sous une forme dérivée de la forme standard [Wpo] des analyseurs de précédence [Col] et d'un analyseur pour langages LR (K) [Knu]

Analyseur descendant.

L'idée de base peut être ainsi présentée : l'analyseur prédictif de Kuno-Oettinger [Ku O] opère sur des grammaires sous forme standard, c'est à dire composées de règles du type :

$$A \longrightarrow a\,\varphi$$

Les majuscules représentent des non terminaux, les minuscules des terminaux et les lettres grecques une suite de non terminaux (pouvant être vide).

La version "déterministe" de cet analyseur, ne procédant pas essais successifs doit opérer sur des grammaires sous forme standard telles que si

$$A \longrightarrow a\varphi$$

en est une règle, il n'existe pas d'autres règles de la forme :

$$A \longrightarrow a\psi$$

Le problème consiste donc à trouver une telle grammaire équivalente à celle donnée initialement. La question du passage à une grammaire sous forme standard est résolue [Gre]. L'autre transformation devra être effectuée semi-manuellement, si elle est possible.

En fait, les contraintes sur la forme des règles de la grammaire à obtenir peuvent être moins fortes : il suffit qu'elles soient du type :

$$A \longrightarrow B\varphi$$

en interdisant qu'il existe une règle

$$A \longrightarrow C\psi$$

pour laquelle

$$\text{si } S_M = \{x \mid M \Longrightarrow x\ y$$

($\Longrightarrow$ veut dire " se dérive en ")

on ait $S_B \cap Sc \neq \emptyset$

J.M. FOSTER a écrit un programme dénommé . SID [Wov] entreprenant la transformation de la grammaire initiale en une grammaire sous cette forme. Dans un premier stade, il élimine les récursivités à gauche (qui ne peuvent être traitées par un analyseur descendant), dans un second, où il peut échouer, essaye de "factoriser" les parties droites délictueuses.

Par exemple :

$$S \longrightarrow \underline{a}X \mid \underline{a}Y$$

se réécrit en :

$$S \longrightarrow aP$$
$$P \longrightarrow X \mid Y$$

En cas de succès, ce programme fournit un analyseur descendant ne procédant pas par essais successifs : de plus, si dans les règles étaient inclus des points de génération, ils réapparaissent aux endroits appropriés.

SID est employé pour la construction de compilateurs : il a été en particulier appliqué au traitement d'un langage du genre ALGOL 60 spécialisé dans les problèmes en temps réel [CGr].

Analyseurs de précédence.

- A..COLMERAUER décrit en [Col] des grammaires de précédences avec lesquelles on peut employer des analyseurs ascendants déterministes. D'une façon générale, elles se caractérisent par les restrictions suivantes :

  - toutes les règles ont des membres droits distincts

  - quelque soit le type des relations de précédence choisies (totale, de gauche à droite, terminale ou totale de gauche à droite), elles sont incompatibles deux à deux.

Ces relations binaires sont au nombre de trois : cette restriction implique qu'entre deux symboles appartenant à l'ensemble des terminaux et des non terminaux ou à une partie de cet ensemble suivant le type des relations, il ne peut y en avoir qu'une de satisfaite.

L'utilisateur dispose de programmes fournissant ces relations de précédence à partir de la grammaire initiale. Selon celle-ci, il peut vérifier si sa grammaire remplit la seconde condition et la modifier dans le cas contraire.

Cette méthode a été appliquée à ALGOL 60, FORTRAN, à un nouvel ALGOL proposé dans [WHo] : PL/I est en cours d'étude [Tas].

Analyseur pour langages LR (k) .

Cet analyseur est ascendant : à un instant donné de l'analyse, il utilise pour déterminer la règle de la grammaire qui s'applique des informations extraites de la partie de la chaîne déja traitée (celle-ci est parcourue de gauche à droite) et les k symboles (contexte) figurant à droite de cette chaîne, K étant fixé une fois pour toutes.

Un algorithme permet de déterminer si pour un k fixé, la grammaire est LR(k), c'est à dire acceptable par l'analyseur précédent. En cas d'insuccés, on fournit à l'utilisateur l'ensemble des règles qui seraient applicables pour un même contexte.

Sa mise en oeuvre est, à l'heure actuelle, étudiée par J. Courtin [Cou]. Il apparait qu'elle entraîne un gros encombrement de la mémoire.

La seconde caractéristique souhaitable concerne la liaison entre l'analyse et un programme de génération. En effet, celle-ci nécessite qu'on définisse l'ordre dans lequel les arbres construits seront délivrés pour leur exploitation.

Nous étendrons d'abord la notion de "frère puiné " (cf § II.2.0). Si un noeud y est frère puiné d'un noeud x nous dirons d'un descendant (cf § II.2.1.6) de x qu'il est situé à gauche d'un descendant de y.

Supposons qu'on dispose de l'arbre (syntaxique ou non) déduit d'une chaîne source - Soit Z l'ensemble de noeuds présents dans cet arbre ayant des noms correspondant à des chapitres d'entrée (cf § II.1.3) - Seuls les arbres ayant pour sommets ces noeuds sont délivrés au programme de génération.

Le premier arbre traité est celui ne possédant :

- aucun descendant appartenant à Z
- aucun noeud de Z à sa gauche -

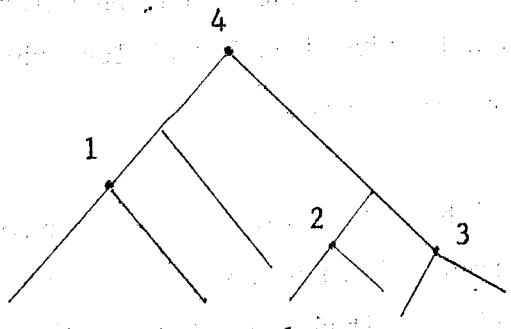
On détermine le prochain arbre livré de la façon suivante :

- S'il existe un sommet $t \in Z$ tel que tous les arbres ayant pour sommet des descendants de t appartenant à Z ont été traités, l'arbre de sommet t est le prochain délivré.

- Sinon l'arbre délivré est celui :

  - ne possédant aucun descendant appartenant à Z
  - tel que tous les noeuds de Z à sa gauche aient été traités -

Exemple :Les noeuds numérotés appartiennent à Z :



Les arbres dont ils sont les sommets sont livrés dans l'ordre indiqué.

Il est donc intéressant que l'analyseur procède à la construction de l'arbre syntaxique dans le même ordre (chaque noeud étant considéré comme appartenant à Z). En fait, la plupart d'entre eux procède de cette façon en examinant la chaîne de gauche à droite. Signalons cependant que certains types d'analyseur de précédence n'opèrent pas toujours ainsi.

II.2.5. Conclusion

On trouve en [ShF] une récapitulation des qualités que doit posséder un système de compilation. Dans ce qui suit, nous examinons dans quelle mesure le déroulement d'une phase ASG telle que nous l'envisageons répond à ces critères.

Possibilité de décrire le langage source et le langage objet.

En ce qui concerne le premier nous avons vu que sa véritable description se limitait à une grammaire context-free. Pour le second, le souhait d'en formuler une reste, à l'heure actuelle, un voeux pieux.

On pourrait toutefois envisager s'il s'agit d'un langage machine ou d'assemblage de préciser les différents champs d'une instruction et de faciliter ainsi l'écriture de l'instruction générer.

Compréhension aisée du langage source à la lecture du compilateur ainsi formulé.

Notre but n'est certainement pas de définir sémantiquement le langage source à l'aide d'un programme de génération. Cependant, une bonne connaissance du langage objet permet de comprendre sans trop de difficultés la signification du texte source qui en est donnée par le compilateur. En effet, les différentes parties en sont nettement séparées : d'une part, l'analyse syntaxique qui distingue certaines classes de sous-chaînes propres à être traitées indépendamment, d'autre part les actions effectuées à partir des arbres déduits de ces sous-chaînes, enfin la génération en langage objet.

Possibilités d'optimisations locales et globales du programme objet.

En ce qui concerne les premières, le fait de travailler sur un arbre et de pouvoir reconnaître diverses configurations dans cet arbre (cf. II.2.3.1) permet de faciliter leur recherche. Un exemple en est fourni ayant rapport aux générateurs d'enregistrements en III.3.4.

Les secondes sont en général très difficiles à obtenir : par exemple, celle relative à la non réévaluation d'un pas ou d'une borne dans une boucle pour d'ALGOL 60. Il n'y a rien de spécial dans le langage de génération prévu à cet effet. Signalons cependant que d'une façon analogue à celle que nous employons en III.4.1. pour calculer des "fermetures" dans les nouveaux types, on pourrait déterminer les procédures récursives d'un programme ALGOL 60 [ ].

Modification aisée du langage source.

On conçoit que tout retrait ou ajout au langage source et portant sur la syntaxe peut être facilement effectué. La fonction des chapîtres à créer, à détruire ou à modifier en conséquence, il est ensuite possible de déterminer l'ensemble des chapîtres susceptibles de remaniements. Ceux-ci seront donc les seuls à être examinés.

Facilité de détections d'erreurs.

En ce qui concerne les erreurs syntaxiques, elles sont automatiquement décelées par l'analyseur. Le fait que l'analyseur soit déterministe permet de les localiser d'autant mieux. Il serait aussi intéressant d'inclure dans l'analyseur un dispositif analogue à celui employé dans [Col] pour poursuivre l'analyse du texte source même lorsqu'on a reconnu qu'il est erroné : ceci afin de détecter d'éventuelles erreurs postérieures.

Il reste ensuite de nombreuses vérifications à effectuer dans le programme de génération : nous en verrons des exemples en III.3. En général, elles se situent en tête de chapîtres. L'instruction message pourrait facilement fourni en même temps qu'un libellé sur une incorrection une indication permettant de localiser une sous-chaîne qui la contient.

Possibilité pour le langage de génération de se décrire lui-même.

Nous entendrons par là, possibilité de traduire un programme écrit dans ce langage en un langage objet à l'aide d'un compilateur écrit en langage de génération.

Un tel travail n'a pas encore été effectué. Si le langage objet considéré est ALGOL 60, nous ne pensons pas qu'il se révèle difficile. D'une part, le langage de génération possède de nombreuses ressemblances avec ALGOL 60, d'autre part l'emploi des procédures décrites en II.2.2. ne nécessiterait dans la plupart des cas que des transformations locales.

Il reste à discuter de deux particularités de ce langage et qui peuvent apparaître comme contestables.

La première concerne le fait qu'à chaque chapître est uniquement attaché un noeud. Dans le cas où l'arbre utilisé est celui issu de l'analyse syntaxique, il serait plus intéressant de faire correspondre une règle à chaque chapître : en effet, il est possible qu'une métavariable soit sujet de plusieurs règles et ainsi, nous ne serions pas contraints de les distinguer dans un chapître. Cette information (à savoir qu'elle a été la règle appliquée pour obtenir telle métavariable) est ici sacrifiée à la facilité de lecture du compilateur. Signalons que dans la majeure partie des cas, les règles ayant même sujet sont peu nombreuses et aisément distinguables.

La seconde tient au choix que nous avons fait de construire des arbres et de les parcourir ensuite pour produire le programme objet. Nous avons vu que les systèmes dénommés TROL (cf. II.0) et SID (cf. II.2.4.2) ne procèdent pas ainsi: à certains stades de l'analyse, lorsqu'une règle intéressante a été appliquée, une action est déclenchée mais elle n'opère pas sur un arbre.

Les raisons de ce choix sont essentiellement les suivantes :

- être indépendant de l'analyseur choisi, pourvu qu'il délivre les arbres distingués dans l'ordre convenu (cf. II.2.4.2).
- pouvoir transformer ces arbres et ensuite les parcourir (dans un ordre qui peut ne pas être celui dans lequel ils ont été construits) afin d'effectuer la génération en langage objet de la façon la plus commode possible.

-III- COMPILATION D'UN LANGAGE DERIVE D'ALGOL X.

Parmi les nouveautés présentes dans les rapports ALGOL X préliminaire [v w 1, v w 2] figurent la possibilité d'utiliser des variables d'un type défini dans le programme et des variables de type "référence".

Notre but ici est d'examiner les problèmes posés par la compilation et l'implantation en machine de ces nouveaux éléments. Pour ceci, nous les avons introduits dans un contexte d'utilisation restreint : expressions arithmétiques et booléennes, instruction d'affectation et instructions conditionnelles, blocs, les deux types de base étant entier et logique. Le langage source ainsi formé et présenté en III.1.

En III.2, on décrit le langage objet qui a été considéré : sa zone de travail est divisée en deux parties, la pile et une zone dite de liste. Les questions relatives à la traduction d'un programme source dans ce langage sont examinées en III.3 ; les solutions proposées sont exprimées à l'aide du langage exposé en I.

En III.4, nous abordons le problème important de la gestion de la mémoire à l'exécution : diverses approches sont envisageables. Parmi celles-ci, nous avons choisi une organisation en "pages" de la mémoire. Le fonctionnement des processus de "récupération" et de "retassement" opérant sur la zone de liste sont décrits dans ce cadre.

Enfin, les réalisations pratiques sont présentées. Elles concernent essentiellement :

- un compilateur expérimental écrit en ALGOL 60 du langage source. Sa structure est déduite des programmes du § III.1.

- un interpréteur lui aussi écrit en ALGOL 60. Il est composé de procédures correspondant aux instructions du langage objet : en utilisant une grammaire secondaire (ici disques) il effectue la gestion de la mémoire précédemment vue. Le programme principal fourni à cet interpréteur est le produit du compilateur.

II.1 Description du langage source

II.1.0 Préliminaires.

Nous ne reprendrons pas complètement ici la partie de [Vw 1] consacrée à la sémantique d'un programme. On s'attachera essentiellement à la définition d'une valeur.

Une valeur est définie par un mode, (on dit qu'elle est "de ce mode") et une "valeur instantanée".

Un mode est défini par ses deux attributs :

- un type, soit de base (entier ou logique) soit nouveau.

- un niveau représenté par une valeur entrée. Dans le cas où ce niveau est supérieur à zéro, on dira qu'une valeur de ce mode est un nom.

Une valeur instantanée est "la propriété intrinsèque qui différencie une valeur des autres valeurs de même mode" [Vw1]. Dans un contexte où le mode est précisé, on emploiera souvent le terme valeur pour valeur instantanée.

On appelle valeurs primitives où de base les valeurs de type de base.

Une valeur "nouvelle" est d'un nouveau type, caractérisé par une série de modes dans un ordre donné : à chacun de ces modes est associé un nom dit "composant". Une valeur nouvelle est alors associée à une série de valeurs, chacune de celles-ci étant du mode du composant correspondant.

A un nom peut être affectée (ou attribuée) une valeur (Cf. 4) : celle-ci est alors de même type que le nom et de niveau inférieur de 1 à celui du nom. Une expression (Cf 5,6,7,8,9) "possède" une valeur dès qu'elle a été évaluée.

Nous emploierons dans la suite le terme de valeur p-terminale (p entier) ; étant donné une valeur de niveau n, sa valeur p-terminale est ($p \leq n$) obtenue de la façon suivante :

- considérant la valeur donnée, si celle-ci est de niveau supérieur à p, la valeur qui lui est attribuée devient la valeur considérée et le processus est recommencé.

- dans le cas contraire, la valeur p-terminale est la valeur considérée, de niveau p.

Dans le cas où une valeur p-terminale est cherchée avec $p > n$, il y a erreur.

1 Programme - Blocs - Instructions.

Syntaxe :

<programme> ::= <bloc> | <instruction-composée>
<bloc> ::= début <partie-déclaration> ; <corps> fin
<partie-déclaration> ::= <déclaration>
<corps> ::= <instruction> | <instruction> ; <corps>
<instruction-composée> ::= début <corps> fin

Sémantique :

Le nom associé (Cf 2) à une variable déclarée dans un bloc est rendu local à ce bloc. C'est-à-dire que ce nom n'a pas d'existence en dehors de ce bloc.

Si un identificateur "dénote" (Cf. 2) plusieurs noms, le nom considéré à l'occurence de cet indentificateur dans un corps ou dans une déclaration est celui dénoté dans le plus petit bloc qui contient l'occurence : la détermination de ce bloc s'effectue de la même façon qu'en ALGOL 60

A l'extérieur d'un bloc dans lequel un nouveau type est déclaré :

- les valeurs de ce nouveau type sont inexistantes
- les valeurs qui pouraient être possédées par des expressions de composants (Cf 9) ce rapportant à ce nouveau type dans ce bloc sont aussi inexistante.

Exemples :

```
1) début
       référence entier i ;
                ...
       début
           entier j ;  }
           i := j      }   (II)
       fin             }
       ...             }  (I)
   fin                 }
```

La valeur de i qui est le nom j (Cf 4) dans le bloc (II) est inexistante en dehors de ce bloc, par exemple en (I). On peut dire aussi que valeur i est indéfini (Cf 5).

```
2) début
       référence libre i ;
            . . .
           début
               (entier a, logique b) nt ;   }
               nt x ; x := nt (3, vrai) ;   }   (III)
               i := x ;                     }
           fin                              }
       . . .                                }
   fin                                      }  (IV)
```

La valeur de i dans le bloc III est de type nt. De même que précédemment elle est inexistante à l'extérieur du bloc (III).

par exemple en IV .

3) début

référence entier i ;

. . .

début

(entier a, logique b) nt ;
nt x ; x := nt (3,vrai) ;
i := a de x ;

(V)

fin

. . .

fin

Dans le bloc (V), l'expression de composant a de x possède comme valeur un nom dont la valeur attribuée est 3 (Cf 9) : ce nom est attribué à i. A l'extérieur de ce bloc val i est indéfini.

2 Déclaration

Syntaxe :

⟨déclaration⟩ ::= ⟨déclaration-unique⟩ |
⟨déclaration-unique⟩ ; ⟨déclaration⟩

⟨déclaration unique⟩ ::= ⟨déclaration-simple⟩ |
⟨déclaration-de-nouveau-type⟩

⟨déclaration-simple⟩ ::= ⟨déclaration-de-variable⟩ |
⟨déclaration-de-référence⟩

⟨déclaration-de-variable⟩ ::= ⟨type-entier⟩ ⟨variable-entière 2⟩ |
⟨type-logique⟩ ⟨variable-logique 2⟩ |
⟨type-nouveau⟩ ⟨variable-nouvelle 2⟩
⟨type-libre⟩ ⟨variable-libre 2⟩

⟨déclaration-de-référence⟩ ::= référence ⟨déclaration-simple⟩

⟨déclaration-de-nouveau-type⟩ ::= (⟨déclaration-de-composants⟩)
⟨nouveau-type⟩

⟨déclaration-de-composants⟩ ::= ⟨déclaration-de-composant-unique⟩ |
⟨déclaration-de-composant-unique⟩ ;
⟨déclaration-de-composants⟩

⟨déclaration-de-composant-unique⟩ ::= ⟨déclaration-simple⟩

⟨type-entier⟩ ::= entier

⟨type-logique⟩ ::= logique

⟨type-libre⟩ ::= libre

⟨variable-entière 2⟩ ::= ⟨variable-entier⟩ |
⟨composant-entier⟩

⟨variable-logique 2⟩ ::= ⟨variable-logique⟩ |
⟨composant-logique⟩

⟨variable-nouvelle 2⟩ ::= ⟨variable-nouvelle⟩
⟨composant-nouveau⟩

⟨variable-libre 2⟩ ::= ⟨variable-libre⟩ |
⟨composant-libre⟩

Sémantique

Une déclaration-unique a pour effet soit de déclarer une variable soit de déclarer un nouveau type.

La déclaration d'une variable crée un nom, dit "appellation" de la variable. Cette appellation est "dénotée" par l'identificateur représenté par ⟨variable-entiere2⟩, ⟨variable-logique 2⟩ ... etc... Le mode de ce nom est spécifié par :

- le type associé à cette déclaration
- un niveau égal au nombre de symboles référence présents dans cette déclaration plus 1.

Ceci implique que l'on ne peut déclarer aucune variable de niveau 0. Cependant, certains éléments du langage(valeur de base, expressions arithmétiques ou logiques, valeur x,etc...) sont, comme nous le verrons de niveau nul.

Enfin, un nom de type libre est tel que la valeur qui lui est affectée peut être de type quelconque.

Une déclaration de noveau type entraine :

- la désignation par l'identificateur que représente ⟨nouveau type⟩ de ce nouveau type (Cf 6).

- la spécification des modes des composants de ce nouveau type. Ce mode est spécifié pour chaque composant comme précédemment pour une variable par une déclaration de composant unique. De plus, cette dernière fournit l'identificateur (représenté par ⟨variable-entiere 2⟩,⟨variable-logique 2⟩,... etc....) désignant le composant correspondant d'une valeur de ce nouveau type (Cf 0).

L'ordre des modes caractérisant ce nouveau type est donné par l'ordre d'apparition de gauche à droite des déclarations de composant qui lui correspondent.

D'une manière analogue à celle de [V w 2], les modes des composants de nouveaux-types sont restreints de la façon suivante :

Soit un nouveau type a

- si le mode d'un de ses composants est de type b et de niveau 1.

- si ce type b est a libre, la déclaration d'un tel nouveau type est interdite.

- sinon on considère le mode des composants du nouveau type b et on recommence le processus.

Exemples :

1) déclaration de variable :
référence référence entier i ;

Un nom dénoté par i est crée. Son mode peut être représenté par le couple (entier, 3).

2) déclaration de nouveau type :
(entier a, logique b) nt ;

- nt désigne le nouveau type déclaré
- le nouveau type est formé des modes :
  (entier, 1) (logique, 1)
- a et b sont les identificateurs se rapportant respectivement au premier et au second composant de nt.

La déclaration du nouveau type

(entier a, nti b) nti est interdite

De même celle de ntii tel que :

(ntii aa, logique bb) nt ;

(entier a, nt b)ntii ;

3) Instruction.

Syntaxe :

⟨instruction⟩ ::= ⟨instruction-de-base⟩ |
⟨instruction-étiquetée⟩ |
⟨rien⟩

⟨instruction-de-base⟩ ::= ⟨instruction-d'affectation⟩ |
⟨instructinn-conditionnelle⟩ |
⟨instruction-allera⟩ |
⟨bloc⟩ |
⟨instruction composée⟩

⟨instruction étiquetée⟩ ::= ⟨étiquetage⟩ ⟨instruction⟩

⟨étiquetage⟩ ::= ⟨étiquette⟩ :

⟨rien⟩ ::= ( )

⟨instruction conditionnelle⟩ ::= ⟨proposition si⟩ ⟨il⟩ sinon ⟨instruction⟩

⟨proposition si⟩ := si ⟨expression logique⟩

⟨il⟩ ::= ⟨instruction⟩

⟨instruction allera⟩ ::= allera ⟨étiquette⟩

Sémantique :

Les règles régissant l'utilisation des étiquettes sont les mêmes qu'en ALGOL 60. Une instruction "allera" interrompt le déroulement séquentiel des instructions : l'instruction suivante à exécuter est l'instruction ayant pour étiquette celle précisée dans l'instruction "allera".

L'instruction "rien" est une instruction ineffective. Les instructions conditionnelles provoquent l'exécution de l'instruction ⟨i1⟩ si l'expression logique possède une valeur 0-terminale, égale à vrai sinon l'exécution de l'instruction suivant sinon.

Exemple :

Soit l'instruction conditionnelle
Si a alors X := b+c sinon ( )
où a est déclaré :
référence référence logique a

Cette instruction peut être interpretée comme [Cf 5]
Si valeur valeur valeur a alors X := B+C sinon ( )

4 Instruction d'affectation

Syntaxe :

⟨instruction d'affectation⟩ ::= ⟨partie-gauche⟩ := ⟨expression⟩

⟨partie gauche⟩ ::= ⟨expression-non-libre⟩ |
⟨expression-libre⟩

⟨expression⟩ ::= ⟨expression-arithmétique⟩ |
⟨expression-logique⟩ |
⟨expression-nouvelle⟩ |
⟨expression-libre⟩ |
⟨toin⟩

⟨expression-non-libre⟩ ::= ⟨expression-nouvelle 1⟩ |
⟨variable-entière 1⟩|
⟨variable-logique 1⟩

⟨toin⟩ ::= nil

Sémantique :

Les instructions d'affectation servent à attribuer une valeur (déduite de la valeur possédée par l'expression en partie droite à un nom désigné par la partie-gauche : ceci entraîne que la valeur possèdée par l'expression en partie gauche soit au moins de niveau 1.

Les types de ce nom et de la valeur possédée par l'expression en partie droite doivent être identiques sauf si le nom est de type libre.

Soit n le niveau de la partie-gauche, m celui de la partie droite. Si m est inférieur à n-1 l'affectation est impossible. Dans le cas contraire, la valeur attribuée au nom est la valeur n-1-terminale de la valeur possédée par l'expression.

La valeur de l'expression toin est un nom tel que :

- son mode est quelconque
- la valeur qui lui est attribuée est une valeur non existante pour tous les modes.
- On ne peut attribuer à ce nom une autre valeur.

Il ressort de ceci qu'on ne peut attribuer la valeur attribuée à nil à un nom de niveau 1.

Exemple :

Soit x et y déclarés comme suit :

entier x ; référence entier y ;

la suite d'instructions :

```
x := 3 ;
y := x ;
```

a pour effet :

- d'affecter la valeur de base 3 et de niveau zéro (Cf 5) au nom x.
- d'affecter au nom y de niveau 2 une valeur de niveau 1, c'est-à-dire le nom x.

l'instruction :

```
y := nil ;
```

affecte au nom y le nom nil. L'écriture

```
x := nil
```

est interdite.

5 Expressions arithmétiques et logiques

Syntaxe :

⟨expression arithmétique⟩ ::= ⟨terme⟩ |
⟨expression arithmétique⟩ ⟨S+⟩ ⟨terme⟩

⟨S+⟩ ::= +

⟨terme⟩ ::= ⟨facteur⟩ | ⟨terme⟩ ⟨smul⟩ ⟨facteur⟩

⟨smul⟩ ::= mul

⟨facteur⟩ ::= (⟨expression-arithmétique⟩) |
⟨variable-entiere 1⟩ |
⟨nombre-entier⟩

⟨variable-entière 1⟩ ::= ⟨variable-entiere⟩ |
⟨expression-entiere de composant⟩ |
valeur ⟨variable-entier 1⟩

⟨nombre-entier⟩ ::= ⟨nombre-entier⟩ ⟨chiffre⟩ |
⟨chiffre⟩

⟨chiffre⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

⟨expression-logique⟩ ::= ⟨e⟩ | ⟨facteur logique⟩ | ⟨relation⟩

⟨e⟩ ::= ⟨expression-logique⟩ ou bien ⟨facteur-logique⟩

⟨facteur-logique⟩ ::= ⟨0⟩ | ⟨secondaire-logique⟩

⟨0⟩ ::= ⟨facteur-logique⟩ et ⟨secondaire-logique⟩

⟨secondaire-logique⟩ ::= non ⟨primaire-logique⟩ |
⟨primaire logique⟩

⟨primaire-logique⟩ ::= (⟨expression-logique⟩) |
⟨variable-logique 1⟩ |
⟨valeur-logique⟩

⟨variable-logique 1⟩ ::= ⟨variable-logique⟩ |
⟨expression-logique-de-composant⟩
valeur ⟨variable-logique 1⟩
⟨valeur-logique⟩ ::= vrai | faux

Sémantique :

Une variable-entière
un nom de type entier, une variable logique un nom de type logique. Un nombre entier et une valeur logique sont respectivement des valeurs primitives de type entier et logique et de niveau zéro.

Soit n ($n \geq 0$) le nombre de symboles valeur présents dans une variable-entière 1 et m le niveau du nom de la variable-entière ou de l'expression entière de composant associée (Cf 8). La valeur m-n terminale de ce nom devient la valeur possédée par la variable entière 1 dès l'évaluation de celle-ci. Le schéma est analogue pour une expression logique 1.

Un terme ou une expression arithmétique contenant un opérateur (+, mul) possède une valeur de type entier et de niveau zéro, résultat de l'opération addition ou multiplication sur les valeurs 0 terminales des valeurs possédées par les opérandes. De même, les expressions logiques dans lesquelles figurent un ou plusieurs opérateurs logique (non, ou, et) possèdent une valeur 0-terminal de type logique.

Exemple :

Soit x et y déclarés comme suit :

entier x
ref entier y

et initialisés par :

x := 3 ;
y := x

l'expression

x + y

a pour valeur la somme des valeurs 0-terminales de x et de y c'est-à-dire 6.

Cette expression pourrait aussi s'écrire :

x + valeur y

ou encore

valeur x + valeur valeur y

6 Expressions nouvelles

⟨expression-nouvelle⟩ ::= ⟨expression-nouvelle 1⟩ |
⟨générateur-d'enregistrement⟩

⟨expression-nouvelle 1⟩ ::= ⟨variable-nouvelle⟩ |
⟨expression-nouvelle-de-composant⟩ |
⟨valeur expression-nouvelle 1⟩

⟨generateur-d'enregistrement⟩ ::= ⟨nouveau-type⟩
(⟨liste-de-composants⟩)

⟨liste de composants⟩ ::= ⟨composant⟩ |
⟨composant⟩ , ⟨liste-de-composants⟩

⟨composant⟩ ::= ⟨expression⟩

Sémantique :

L'évaluation d'un générateur d'enregistrement crée un nom de niveau 1 dont la valeur attribuée est une valeur nouvelle (du type désigné par un identificateur de⟨nouveau type⟩ Cf(2) . Aux composants de cette valeur nouvelle considérés de gauche à droite sont affectées (Cf4) en "parallèle" les valeurs des expressions représentées par⟨liste-de-composants⟩ elles-mêmes considérées dans le même ordre.

La présence de l'opérateur valeur dans une expression nouvelle produit le même effet que dans le paragraphe précédent : une variable nouvelle représente d'une façon analogue un identificateur dénotant un nom de type nouveau.

Exemple :

Soient les déclarations :
(entier a, logique b) nt ;
nt x ;
l'instruction :

x :=nt ($, vrai)^5

affecte au nom x une valeur nouvelle telle que les relations suivantes (Cf 8,9) sont vraies :

a de x = 5
b de x = vrai

7 Expressions libres.

Syntaxe :

⟨expression-libre⟩ ::= ⟨variable-libre⟩ |
⟨expression-libre-de-composant⟩

Sémantique :

La valeur possédée par une expression-libre est :

- soit le nom dénoté par l'identificateur représenté par ⟨variable-libre⟩
- Soit le nom possédé par une expression libre de composant.

8 Relations

Syntaxe :

⟨relation⟩ ::= ⟨égalité⟩ |
⟨supinf⟩ |
⟨identité⟩ |
⟨conformité⟩ |

⟨égalité⟩ ::= ⟨expression-arithmétique⟩ ⟨op=⟩ ⟨expression-arithmétique⟩ |
⟨primaire-logique⟩ ⟨op=⟩ ⟨primaire-logique⟩

⟨op=⟩ ::= =

⟨supinf⟩ ::= ⟨expression arithmétique⟩ ⟨opsupinf⟩ ⟨expression arithmétique⟩

⟨opsupinf⟩ ::= > | <

⟨identité⟩ ::= ⟨expression 1⟩⟨opidentité⟩ ⟨expression 1⟩

⟨expression 1⟩ ::= ⟨expression-non-libre⟩ | ⟨expression libre⟩ | ⟨toin⟩

⟨opidentité⟩ ::= est

⟨conformité⟩ ::= ⟨expression-non-libre⟩ ⟨opconformite⟩ ⟨expression libre⟩

⟨opconformité⟩ ::= .=

Sémantique :

Les relations possèdent une valeur logique de niveau zéro. Les relations d'égalité et de supériorité-infériorité portent sur des opérandes de type de base : ces opérandes sont les valeurs 0-terminales des expressions considérées.

Une relation d'identité possède la valeur vraie si les deux noms de même mode possédés respectivement par les expressions 1 à gauche et à droite du symbole est sont identiques.

Soit m (m > 1) le niveau de la valeur possédée par une expression non libre dans une conformité.

La relation de conformité est vraie si le mode de la valeur m-2 terminale de l'expression non libre et celui de la valeur m-2 terminale de l'expression libre sont identiques. De plus cette dernière valeur est affectée (Cf 4) au nom, valeur m -1 terminale de l'expression libre.

Exemple :

Soient les déclarations :

logique u ; réf logique v ; ref logique w ;

libre z ;

et les initialisations :

```
u := vrai ;
v := u ;
```

1) la relation d'égalité

(u=v) = u

est vraie

2) si on fait

w := u

la relation d'identité

v est w

est fausse car les noms v et w, étant de mêmes modes, ne sont pas identiques.

Par contre :

valeur v est valeur w

est vraie. En effet, les valeurs attribuées à v et w sont identiques au nom u.

L'écriture :

valeur valeur v est valeur valeur w

est interdite, les opérandes d'une identité devant être des noms.

3) Si z est initialisé comme suit :

z := vrai ;

la relation de conformité

v .= z

est vraie. La valeur attribuée à z est du même mode que la valeur 0-terminal du nom v.

De plus, la valeur vrai est affectée au nom u.

9) Expression de composant

Syntaxe :

⟨expression-entrée-de-composant⟩ ::= ⟨composant entier⟩ de ⟨expression-nouvelle 1⟩

⟨expression-logique-de-composant⟩ ::= ⟨composant-logique⟩ de ⟨expression-nouvelle 1⟩

⟨expression-nouvelle-de-composant⟩ ::= ⟨composant-nouveau⟩ de ⟨expression-nouvelle 1⟩

⟨expression-libre-de-composant⟩ ::= ⟨composant-libre⟩ de ⟨expression-nouvelle 1⟩

Sémantique :

La valeur terminal de l'expression nouvelle 1 est d'abord déterminée.C'est un nom de nouveau type.

La valeur possédée par l'expression composant est la valeur correspondant au composant indiquée par ⟨composant-entier⟩ , ⟨composant logique⟩ ... etc. (Cf2) dans la valeur attribuée au nom cité ci-dessus.

Exemple :

1) Soient les déclarations :

(entier a, référence nt b) nt ;
référence nt x ; entier y ;
et l'initialisation :

x := nt (3, nt (5, nil))

l'instruction :

y := a de b de x

a pour effet :

- de considérer la valeur -1-terminale du nom x

- de déterminer son deuxième composant qui est le nom

nt (5,nil)

créé dans l'intialisation

- de considérer le premier composant de la valeur nouvelle attribuée à ce nom, c'est-à-dire un nom dont la valeur attribuée est 5.

- enfin d'affecter au nom y la valeur entière 5.

2) Dans le même contexte l'écriture :

a de valeur valeur x

ext interdite : valeur valeur x n'est pas un nom

REMARQUE :

On trouve dans la grammaire des mots variables qui ne sont pas indispensables à la description syntaxique de ce langage, par exemple ⟨partie-déclaration⟩ et ⟨composant⟩. Leur utilité comme "points de génération" est précisée au § III.III.3..

III - 2. Description du langage objet .

Syntaxe :

programme<objet> :: = début <interpréteur> <procédures marque>
<suite d'instructions objet> fin

<procédures marque > : : = <procédure marque> ;
<procédure marque> ; <procédures marque>

<procédure marque > : : = <procédure marque> <nombre entier> ;
début <suite d'instructions objet> fin

<suite d'instructions:objet> :: = <instruction objet>
<instruction objet>;<suite d'instructions objet>

<instruction-objet> : : = <instruction de base>
si comparer alors
début <suite d'instructions> fin sinon
début <suite d'instructions> fin
<étiquette> : <instruction-objet >

Sémantique :

Un programme écrit dans ce langage est un programme ALGOL 60.

L'interpréteur désigne un ensemble de déclarations et de déclarations de procédure qui décrivent l'action des instructions de base.

Nous verrons au § III 4 quelle est la signification des procédures marque.

L'espace de travail des instructions de base est divisé en plusieurs zones, une pile, une pile dite "variables", une pile dite "types" et une zone dite zone de liste : nous en examinerons l'utilité et le fonctionnement aux § III-3 et III.4-

Les instructions de base sont :

- empiler (x)

x est une valeur entière. Elle est placée au sommet de la pile.

- désempiler (n)

Le sommet de la pile est reculé de n-positions

- empiler ad (x)

x est une adresse dans la pile. Elle est placée au sommet de la pile.

- valeur pile (t)

t est une valeur entière. Le contenu du sommet de la pile devient les t éléments situés à l'adresse de pile indiquée par le précédent contenu du sommet de la pile .

- affectation (t)

les t éléments supérieurs de la pile sont placés à l'adresse indiquée par le contenu de l'élément immédiatement en arrière des précédents - Ce dernier devient le premier élément disponible de la pile.

- transfert à la (t, type)

Les t éléments supérieurs de la pile sont transférés dans la zone de liste - Ils sont effacés du sommet de la pile : le contenu de celui-ci devient l'adresse allouée en zone de liste aux précédents éléments - type est une adresse de pile où sont rangées les informations nécessaires à ce transfert (cf § III.3.I et § III.4.I).

- composant (x)

La valeur entière x est ajoutée au contenu du sommet de la pile.

- alléra (x)

Cette instruction est équivalente à l'instruction ALGOL 60 :

alléra x

- addition
- multiplication

Elles effectuent respectivement l'addition et le produit des premiers et seconds éléments de la pile - les opérandes sont détruits et le résultat figure en sommet de pile.

- union
- intersection

Elles effectuent respectivement l'union et l'intersection des valeurs logiques (vrai est représenté par la valeur 1 et faux par 0) qui sont contenues dans les deux premières positions de la pile - Le résultat est fourni comme précédemment.

- infériorité
- supériorité
- égal

Elles produisent le résultat vrai comme précédemment si la valeur du contenu du second élément de la pile est respectivement inférieure, supérieure ou égale à la valeur du premier.

Les autres instructions de base ont été conçues spécialement pour le traitement des variables libres et pour la gestion de la zone de liste : elles sont citées ici, mais on devra se reporter aux paragraphes indiqués pour en comprendre la signification.

- libérer type (< adresse pile d'un type >)
- empiler type (type, fermeture, code composant, code marque, t)
- empiler variable (adresse, type pile, niveau)

Elles ont pour but d'emmagasiner des informations dans la pile types et dans la pile variables (cf III.3.1, III.3.4 , III.4.1).

- valeur libre (x)
- valeur libre 1
- égalité (x)

Ces instructions se rapportent au traitement des variables libres (cf § III.3.2, § III.3.4, § III.3.5).

- composant 1 (x)
- noter (t, et)
- marquer ref (numéro, niveau)
- marquer libre (niveau)

Ces instructions sont nécessaires à la phase de "marquage" (cf § III.4.2 et § III.4.3).

- comparer est une procédure booléenne de l'interprêteur prenant la valeur vraie si un 1 est présent au sommet de la pile ; celui-ci est reculé d'une position.

De plus, trois variables entières déclarées dans l'interprêteur sont utilisées :

- nil (cf § III III.1)
- nihil qui est une valeur spéciale pour les adressess (cf § III.2.1 et § III.III.4.1).
- libre qui repère l'adresse dans la pile type des informations relatives au type libre (cf § III.3.2).

III - 3 - PRINCIPAUX PROBLEMES

Notre propos est ici de décrire au moyen du langage de génération présenté en II.2 le processus de traduction d'un programme écrit dans le langage source (III.1) en un programme objet (III.2).

On a supposé que préalablement à l'analyse syntaxique un examen de la chaîne source (phase AMC, par exemple) a reconnu la structure de bloc. Ceci veut dire que la chaîne source que nous examinons est en fait telle que tous les nouveaux types, composants, variables et étiquettes différents sont désignés par des identificateurs différents.

Un moyen simple de simuler l'existence d'un tel prétraitement a été de rajouter à la grammaire du langage source les règles lexicographiques supplémentaires suivantes :

<nouveau-type> ::= nta | ntb | ntc | ntd | nte | ntf
<variable-entière> ::= eva | evb | evc | evd | eve | evf
<variable-logique> ::= lva | lvb | lvc | lvd | lve | lvf
<variable-nouvelle> ::= nva | nvb | nvc | nvd | nve | nvf
<variable-libre> ::= la | lb | lc | ld | le | lf
<composant-entier> ::= cea | ceb | cec | ced | cee | cef
<composant-logique> ::= cla | clb | clc | cld | cle | clf
<composant-nouveau> ::= cna | cnb | cnc | cnd | cne | cnf
<composant-libre> ::= ca | cb | cc | cd | ce | cf
<étiquette> ::= ea | eb | ec | ed | ee | ef |

Bien entendu ce procédé qui n'intéresse qu'un aspect morphologique du langage source n'enlève rien à la généralité de notre expérience du point de vue qui nous concerne.

Avant de passer successivement à la description des chapitres relatifs aux déclarations, aux expressions, aux générateurs d'enregistrement, aux instructions conditionnelles et d'affectation et aux relations, nous donnons les raisons pour lesquelles la mémoire est divisée en quatre zones à l'exécution d'un programme objet.

La pile est principalement destinée à contenir la représentation des variables déclarées et à évaluer les expressions. On sait qu'une pile est particulièrement adaptée à la gestion des variables en fonction d'une structure de bloc.

Les piles types et variables contiennent des informations se rapportant aux nouveaux types et aux variables déclarées : ces informations sont principalement utilisées au cours de l'exécution pour le déclenchement d'une récupération (cf. § III.4.2.) ou d'un retassement (cf. § III.4.3.).

L'existence de la zone de liste est due au concept de référence (c'est-à-dire considération de valeurs de niveau quelconque) et à la possibilité de créer dynamiquement (c'est-à-dire en cours d'exécution d'un programme), des enregistrements ou valeurs nouvelles. En effet, un enregistrement créé dans un bloc peut-être accessible à l'extérieur de ce bloc; si au nom d'une variable est attribuée une valeur nouvelle, celle-ci ne cesse pas d'exister en fin de validité de la variable. Seul le nom disparaît, c'est-à-dire qu'on ne pourra plus accéder à cette valeur à l'aide de ce nom.

Exemple :

```
début
     (entier a, logique b) nt;
   référence nt xx;
        . . .
        début
           référence nt yy;
                    } (I)
        fin
fin
```

Supposons qu'en (I) figure la suite d'instructions :

```
yy := a(5, vrai)
xx := yy
```

Ceci attribue aux noms xx et yy la même valeur (le nom de la nouvelle valeur créée). En fin de bloc le nom yy disparaît, mais la valeur de xx est toujours valide.

Remarquons que dans le cas ou en (I) figure :

```
y := a(5, vrai)
xx := y
```

Ceci attribue au nom y une valeur nouvelle et ensuite au nom xx le nom y. Ce dernier disparaissant en fin de bloc, la valeur de xx n'est pas définie à l'extérieur du bloc.

Il en résulte que les valeurs nouvelles, non soumises en ce qui concerne leur création et leur disparition à la structure de bloc ne peuvent être contenues dans la pile. Elles seront rangées dans la zone de liste dont nous verrons la gestion au § III.4.

III.3.1. Déclarations

Représentation des valeurs de type de base et de type nouveau.

- Une valeur de type de base et de niveau zéro est contenue dans un seul élément de mémoire. Nous dirons que la "taille" des types entiers et logique est 1.
- Un nom est une adresse d'élément de mémoire : cette adresse peut être contenue dans un élément de mémoire. Le nom spécial nil est l'adresse -1 (qui n'existe pas).

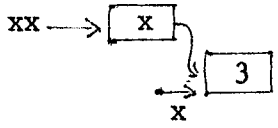
On utilisera quelquefois la notation dite des "petites boites" pour représenter un nom et sa valeur attribuée : une petite boite est un élément de mémoire, une flèche associée indique son nom.

Exemple :

Soit le programme :

```
début entier x;
      référence entier xx ;
      x  := 3;
      xx := x
fin
```

Le nom xx après exécution, peut être figuré par :



Nous avons choisi, pour des raisons d'économie de place en mémoire, de représenter une valeur nouvelle par un ensemble ordonné d'éléments contigus de mémoire. Un nom de niveau 1 et de nouveau type est l'adresse de l'élément inférieur dans un tel ensemble.

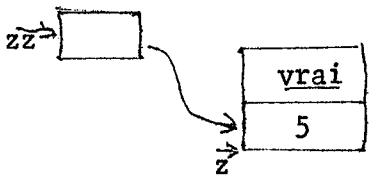
Le nombre de ces éléments, ou taille du nouveau type est la somme des tailles des valeurs de ses composants.

Exemple :

Soit le programme :

```
début (entier a, logique b) nt;
      référence nt zz; nt z;
   z  := nt(5, vrai);
   zz := z
fin
```

Le nom zz, après exécution, peut être représenté par :



Déclaration d'une variable

La tache à remplir est la suivante :

- associer à l'identificateur représentant cette variable une description formée des trois composants :
  {<type>, <niveau>, <adresse pile>}
  qui se rapportent à cette variable.
- générer une réservation de n éléments dans la pile, n étant la taille du type de la variable si le niveau de celle-ci est 1 ou 1 si son niveau est plus élevé. L'adresse pile de cette variable est l'adresse de l'élément ayant la plus petite adresse dans la pile.

- générer la réservation de trois éléments dans la pile variables contenant :
  - l'adresse pile de la variable
  - l'adresse pile type du type de la variable (cf. déclaration de nouveau type).
  - le niveau de la variable.

Notons que cette génération n'est pas toujours possible si la taille du type est inconnue : si celui-ci est déclaré dans le même bloc que la variable, il faut donc attendre que cette taille soit calculée.

Déclaration d'un composant

La tâche à remplir est d'associer à l'identificateur représentant chaque composant une description de la forme :

{<type>, <niveau>, <adresse enregistrement>}

L'adresse enregistrement est la somme des tailles des valeurs des composants précédant, dans l'ordre de gauche de droite, à la déclaration, le composant considéré.

Déclaration d'un nouveau type

Les tâches à remplir sont les suivantes :

- associer à l'identificateur représentant un nouveau type une description de la forme :

{<taille>, <code type>, <nombre de composants>, <adresse pile>, <adresse pile types>, <description des composants>}

* Code type est un entier distinct pour tous les types
* le sixième élément de cette description est une description ayant pour composants les identificateurs représentant les composants de ce nouveau type.

- générer la réservation de deux éléments dans la pile. L'adresse du premier de ces éléments est l'adresse pile. Son contenu fournit l'adresse à laquelle sera éventuellement créée une nouvelle valeur.
- générer la réservation de 5 éléments dans la pile types. L'adresse du premier est l'adresse pile types.

  Les éléments se rapportant à ce type contiendront :
  - l'adresse pile
  - le numéro de "fermeture" (cf. III.4.2.)
  - le code composants (cf. § III.4.2.)
  - le code type
  - la taille

Chapitres du programme de génération se rapportant aux déclarations :

Dans ces conditions, la partie génération d'un compilateur s'écrit, pour les déclarations, de la façon suivante :

e déclaration de variable :

```
compteur références := 1;
établir *i i r i* = {* i i *}
établir 'd-courant' = {* i i r i *} #
```

e déclaration de référence :

```
traiter * i r i * ;
compteur références := compteur références +1 #
```

```
* e déclaration-unique :
        traiter * i i * ;
        si * i * eq 'déclaration-simple' alors
            début
                établir 'd-courant' [1] = { description 'd-courant' [1],
                                            compteur références};
                établir '1 n variables' =
                        {'d-courant' [1] , description '1 n variables'}
            fin #
```

commentaire la description dénommée " 1 n variables" contient le nom des variables déclarées dans la partie déclaration considérée.

e déclaration de composant unique :

```
        traiter * i i * ;
        établir 'd-courant' [1] = {description 'd-courant' [1], compteur références};
        si présence de 'd-courant' [1,1] dans 'types inclus'
        alors
            établir 'types inclus' = {'d-courant' [1,1], description 'types-inclus'};
        établir 'liste de composants' = {'d-courant' [1], description
                                         'liste de composants'};
#
```

Commentaire

La description "types inclus" contient les types des composants du nouveau type.

La description "liste de composants" contient les noms des composants du nouveau type.

e déclaration de composant :

```
traiter * i *;
si * r i * eq ',' alors
    début
        traiter * r r i *
        compteur composants := compteur composants +1
    fin
#
```

e déclaration de nouveau type :

```
traiter * r i * ;
code type := code type +1 ;
établir * i r 3 i * = {0, code type, compteur composant,
                       ip+1, 0, {description 'liste de composants'}};
établir 'relation' = { {* i r 3 i *, description 'types inclus' },
                       description 'relation'};
établir 'nouveaux types' = {* i r 3 i *, description 'nouveaux types'};
initialiser 'liste de composants' ;
initialiser 'types inclus' ;
générer "empiler ( nihil) ;
         empiler ( nihil) ;" sur po ;
ip := ip+2
#
```

Commentaire

La description "relation" a pour éléments des descriptions, chacune contenant en premier élément le nom d'un type, les autres éléments étant les types de ses composants.

La description "nouveaux types" contient les noms des nouveaux types déclarés dans la partie-déclaration considérée.
ip est l'indice du sommet de pile.

* e partie-déclaration :

```
traiter * i * ;
nouveau ip; nouveau ipo; nouveau ipt;
ip := ip moins 1; ipo := ipo moins 1; ipt := ipt moins 1;
```

Commentaire

Lorsque toutes les déclarations ont été examinées on procède successivement :

- à la création de nouveaux indices de piles :
  ip pour la pile, ipt pour la pile types,
  ipv pour la pile variable.
- au calcul des tailles des types.

Pour ceci on examine successivement tous les types et on calcule la somme des tailles valeurs de ses composants : si cela est possible, c'est-à-dire si les composants sont de niveaux supérieurs à 1 ou de types dont la taille est connue, on supprime ce nouveau type de la liste des nouveaux types examinés. Si après un examen, il reste des tailles à déterminer, on recommence le processus.

Du fait de la restriction sur les modes des composants (cf. § III.1.2) et du nombre fini de nouveaux types déclarés, le processus épuise la liste des nouveaux types examinés.

- au calcul du code composant de chaque type (cf.III.4.2.).
- à la génération des procédures marque correspondant aux nouveaux types déclarés (cf. III.4.2.).
- au calcul de l'adresse enregistrement de chaque composant.

Celle-ci est déterminée en examinant pour chaque type tous ses composants : connaissant la taille des valeurs de chacun d'entre eux, on calcule aisément leur adresse enregistrement.

- à l'établissement des variables. Les tailles étant connues, on procède à la réservation des éléments de pile nécessaires pour une variable de niveau 1.
- à la retenue dans la description types valides des nouveaux types déclarés dans ce bloc. Ceci est utile pour la "libération" en fin de bloc des valeurs de nouveaux types créés dans ce bloc.

Remarquons que la présence d'un point de génération pour ce chapitre rend nécessaire la règle syntaxique a priori inutile qui s'écrit (cf. III.1) :

<partie déclaration> ::= <déclaration>

Commentaire calcul des tailles ;

```
examen:
    parcourant 'z' avec i faire
        début
            établir 'genre'= {'z'[i,6] };
            c := o;
            parcourant 'genre' avec j faire
                début
                    si 'genre'[j,2] > 1 alors c := c+1 sinon
                        début
                            t := 'genre'[j,1,1] ;
                            si t = o alors allera sortie c sinon
                            c := c+t
                        fin
                fin;
            'z'[i,1] := c ;
            suprrimer i de 'z' ; i := i-1;
    sortie c :
        fin;
        si ¬ nul 'z' alors allera examen;
```

```
Commentaire   calcul de l'adresse enregistrement
              Génération des procédures marque;

parcourant ‘nouveaux types’ avec i faire
    début
      c :=‘nouveaux types’ [1,1] + 1;
      générer " 'procédure' marque ‘nouveaux types’ [i,2] ";
                'debut'
                   noter (" ‘nouveaux types’ [i,1] " ,
                         et " ‘nouveaux types’ [i,2] ");"
              sur p marquage;

parcourant ‘nouveaux types’ [i,6] avec  j  faire
    début
        établir ‘composant’ = {description ‘nouveaux types’ [i,6,j]};
        c := c- si ‘composant’ [2] > 1  alors
                   1 sinon ‘composant’ [1,1] ;
        établir ‘composant’ = {description ‘composant’, c};
        générer "composant 1 ("c"); "sur p marquage;
        si ‘composant’ [1] eq ‘libre’ alors
              générer "marque libre ("‘composant’ [2]");"
              sur p marquage
        sinon si ‘composant’ [2] > 1 alors
              générer "marquer réf ("
                      ‘composant’ [1,2] ","
                      ‘composant’ [2] "); "sur p marquage.
        sinon
              générer "marque ‘composant’ [1,2] ";"sur p marquage;
    fin
         désempiler(1) 'fin';" sur p marquage
    fin;
```

commentaire - Calcul du code composant. Ce code permettra de retrouver (par divisions entières successives par l'entrée tc) les adresses enregistrements de tous les composants de niveau supérieur à 1 contenus dans un enregistrement de type donné.

- Etablissement dans pile types.

```
Commentaire calcul du code composant;
    établir 'z1' = {copie 'nouveaux types'};
    parcours : parcourant 'z1' avec i faire
        début
            établir 'genre' = {'z1'[i,6]};
            cc := o; k := o;
            parcourant 'genre' avec j faire
                début
                    nc := 'genre'[j,2];
                    si nc = 1 et présence de 'genre'[j,1] dans 'z1'
                                alors allera sortie cc
                    sinon si nc > 1 alors
                       début k := k+1;
                           cc := cc * tc + 'genre'[j,3]
                       fin
                    sinon
                       début k := k + 'genre'[j,1,7];
                             m := 0; 1 := 0;
                          e : si m ≠ k alors
                              début 1 := 'genre'[j,3]* tc + 1;
                                    m := m+1; allera e fin;
                              cc := cc * 'genre'[j,1,7]*tc + 1 + 'genre'[j,
                       fin
                fin;
```

```
        établir ‘z1’[i] = {description ‘z1’[i], k,cc};
        générer "empiler type (" z1[i,4] ","
                                 0  ","
                                 cc ","
                              ‘z1’[i,2] ","
                              ‘z1’[i,1] ");" sur po;
        ‘z1’[i,5] := ipt + 1;
        ipt := ipt+5;
        supprimer i de ‘z1’ ; i := i-1;
    sortie cc :
fin;
si ¬ nul ‘z1’ alors allera parcours ;

Commentaire établissement des variables ;
    parcourant ‘liste n variables’ avec i faire
        début  ip := ip+1; ipv := ipv + 3;
            établir ‘liste n variables’[i] =
                {description ‘liste n variables’[i], ip};
            générer "empiler (hi hil); " sur po;
            si ‘liste n variables’[i,2] = 1 alors
                début
                    i := 2;
                cr : si i ≤ ‘liste n variables’[i,1,1] alors
                    début
                        générer "empiler (hi hil); "sur po;
                        i := i+1; allera  cr
                    fin
                ip := ip + ‘liste n variables’[i,1,1]-1
                fin;
            générer "empiler variable" ‘liste n variables’[i,3] ","
                                       ‘liste n variables’[i,1,5] ","
                                       ‘liste n variables’[i,2] ");" sur po;
        fin;
```

établir 'types valides' = {'nouveaux types', description 'types valides'};
initialiser 'nouveaux types'

* e bloc :
    traiter * r r i * ;
    traiter * r r r r i *
    ancien ip; ancien ipv; ancien ipt;

    parcourant 'types valides'[1] ou i faire

        générer "libérer type ("'types valides'[1,i,4]");" sur po;

    supprimer 1 de 'types valides'
    #

commentaire En fin de bloc, les anciennes valeurs des indices de piles sont considérées, les valeurs de nouveaux types sont "libérées" (cf. III.III.4.1.)

III.3.2. Variables libres.

Représentation

Du fait que ces variables peuvent prendre des valeurs de tous les types et qu'une telle valeur peut être affectée à un nom de type déterminé par l'intermédiaire d'une opération de conformité, il faut qu'à chaque variable libre soit attachée à l'exécution une information relative au type de sa valeur attribuée.
La valeur d'une variable libre de niveau supérieur à 1 sera donc représentée par deux éléments de mémoire, l'un contenant une valeur d'un type donné (libre ou non) l'autre l'adresse pile types du type de cette valeur : cette adresse a été choisie pour faciliter la phase de récupération (cf III.4.2.).

En ce qui concerne la taille de la valeur d'une variable libre de niveau 1, on peut envisager de lui attribuer (en plus de l'élément se référant au type de la valeur) :

- soit la taille du plus grand enregistrement pouvant exister dans le programme source.

- soit la taille 1 étant entendu que le contenu d'un tel élément serait une adresse où figure une valeur de niveau zéro.

La première solution nécessite qu'on connaisse la taille du plus grand enregistrement, donc qu'on ait procédé à un examen complet du programme. De fait que des composants peuvent être de niveau 1 et de type libre, la place prise en mémoire par des enregistrements les contenant peut devenir prohibitive si la taille de l'enregistrement maximum est importante : en effet

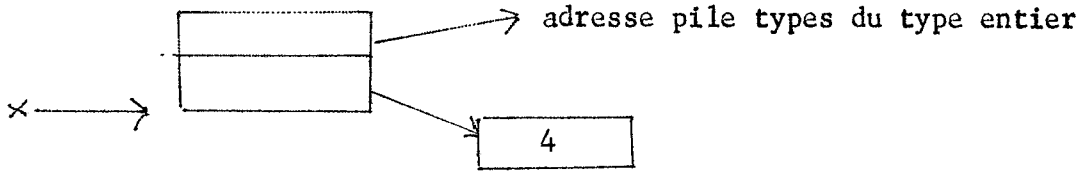
- les enregistrements sont créés dynamiquement.
- les valeurs de ces composants n'étant pas toutes du type de l'enregistrement de taille maximale certains éléments de mémoire seront inutiles et non disponibles pour d'autres utilisations.

La seconde solution aboutit (cf III.3.4.) à une création de valeurs non précisée par le programmeur et éventuellement donc à un grand encombrement de la mémoire :
celui-ci ne pourra être réduit que par l'action d'une procédure de récupération ou de retassement (cf III.4.). Nous avons choisi ici cette dernière solution.

Exemple :

```
libre x ;
  x : = 4  ;
```

la variable libre x est représentée par :



Instructions du langage objet se rapportant aux variables libres.

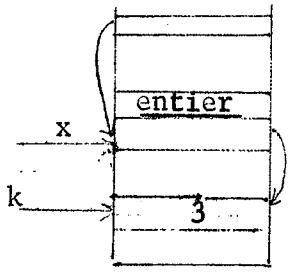
1) Valeur libre (n)

- soit une variable libre de niveau m ; supposons l'adresse de cette variable placée au sommet de la pile : Cette instruction a pour effet d'amener au sommet de la pile la valeur m-n terminale de cette variable et une indication relative au type (libre ou non) de cette valeur.

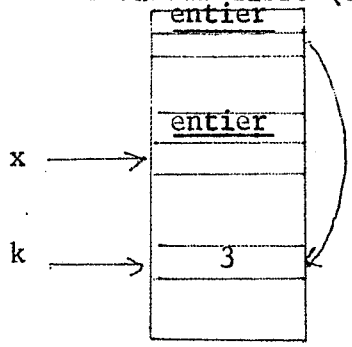
Exemple :

```
référence libre x ;
        entier k ;
        k : = 3 ;
        x : = k ;
```

Soit l'adresse de x au sommet de la pile :



L'action de valeur libre (1) transforme cette configuration en :



2) Egalité (x)

x est l'adresse pile types d'un type : si cette adresse est égale à celle présente au sommet de la pile, ce sommet est remplacé par un 1 sinon par 0.

3) Valeur libre 1

Cette instruction est activée dans le cas d'une affectation où l'on trouve

- en partie gauche : une variable libre a de niveau 1

- en partie droite : une variable libre b de niveau supérieur à 1.

Si la valeur 0-terminale de b est d'un type non libre, il importe de recopier cette valeur. Le rôle de valeur libre 1 est de tester ce type et de déclencher éventuellement cette recopie (cf.III.3.4.1.).

III.3.3. Expressions.

Nous examinons ici le traitement réservé aux métavariables représentant :

- des variables
- des expressions de composants
- des expressions arithmétiques et logiques.

Le chapître consacré aux générateurs d'enregistrements est décrit en III.3.4.

1 Variables

A l'occurence d'une variable un mode "courant" est établi : il est spécifié par :

- la description type considéré
- l'entrée niveau considéré

De plus, on génère le placement de l'adresse correspondant à cette variable au sommet de la pile.

L'effet de l'opérateur valeur est de décrémenter de 1 le niveau considéré et d'entraîner le placement au sommet de la pile de la valeur attribuée à la variable.

Enfin, le mode associé à nil est un mode spécial spécifié par :

- 1 descript (nil)
- le niveau nnil

* e variable-entrée :
* e variable-logique :
* e variable-nouvelle :
* e variable-libre :

```
établir (type considéré) = {*i*[1]};
niveau considéré :=* i*[2];
générer "empilerad ("*i*[3]");"sur po #
```

* e variable-entière 1 :

* e variable-logique 1 :

```
si *i* eq (valeur) alors
   debut
        traiter*ri*;
        si niveau considère ≤ 0 alors
                  message "niveau négatif";
        générer "valeur pile(1);"sur po;
        niveau considéré := niveau considéré-1
   fin
sinon traiter*i* #
```

```
* e variable-nouvelle 1;
    si * i * eq ⌜valeur⌝ alors
        début
            traiter * r i * ;
            si niveau considéré ⩽ 0  alors
                    message "niveau négatif";
            si niveau considéré = 1 alors
                générer "valeur pile" ⌜type considéré⌝[1,1]");" sur po
            sinon
                générer "valeur pile (1);" sur po;
            niveau considéré := niveau considéré -1
        fin
    sinon traiter * i *                          #


* e Foin
        établir ⌜type considéré⌝ = {⌜tnil⌝} ;
        niveau considéré := nnil;
        générer "empiler (nil); "sur po      #
```

2 Expressions de composants

Il faut d'abord vérifier :

- que le composant figure dans la liste de composants associée au type considéré (celui de l'expression nouvelle 1)
- que le niveau considéré n'est pas nul.

Le placement de la valeur 1-terminale de la valeur possédée par l'expression nouvelle 1 est ensuite généré si besoin est.

Il reste :

- à générer le placement de l'adresse correspondant au composant
- à modifier le mode courant.

```
* e expression-entière-de-composant :
* e expression_logique-de-composant :
* e expression_nouvelle-de-composant :
* e expression_libre-de-composant :

        traiter * r r i * ;
        si ¬ présence de * i i * dans ‘type considéré’[1,6] alors
             message "composant non conforme" ;
        si niveau condidéré = 0 alors message "niveau composant nul" sinon
         desniv    : si niveau considéré > 1 alors
             début
                 générer "valeur pile (1); "sur po;
                 niveau considéré := niveau considéré -1;
                 allera desniv
             fin ;
        établir ‘type considéré’ = {* i i *[1] };
        niveau considéré := * i i *[2];
        générer "composant (" * i i *[3]");" #
```

3 Expressions arithmétiques et logiques

Le processus de traduction de telles expressions dans un langage objet analogue a été vu en II.2.3.1.

Nous présentons simplement ici :

- le chapitre relatif aux opérateurs +, nul, ou,et, = et à la métavariable ⟨e a⟩ : il provoque le placement de la valeur 0 terminale de l'opérande gauche au sommet de la pile.
- le chapitre relatif à la métavariable ⟨0⟩ : celle-ci a été introduite afin de distinguer une règle dans laquelle figure un opérateur. Il provoque le placement de la valeur o-terminal de l'opérande droit au sommet de la pile.
- le chapitre relatif à valeur-logique.

* e st :
* e smul :
* e sou :
* e set :
* e op= :
* e ea :

```
desniv : si niveau considéré >0  alors
            début
                générer "valeur pile (1);" sur po ;
                niveau considéré := niveau considéré -1;
                allera desniv
            fin  #
```

```
* e 0 :
        traiter * i * ;
        traiter * r i * ;
        traiter * r r i * ;

  desniv : si niveau considéré > 0 alors
                début
                    générer "valeur pile(1);" sur po;
                    niveau considéré := niveau considéré -1;
                    allera desniv
                fin ;
        générer "intersection;" sur po        #

* e valeur -logique :
        si * i * eq 'vrai' alors
              générer "empiler (1);" sur po sinon
              générer "empiler (0);" sur po ;
        établir 'type considéré' = {'logique'};
        niveau considéré := 0        #
```

III . 3 . 4 - Génération d'enregistrements - Instructions

1. Instruction d'affectation

Les vérifications à effectuer sont les suivantes :

- si la partie droite est nil le mode de cette partie droite devient celui d'une valeur qui peut être attribuée à la partie gauche.
- les types gauche et droit doivent être compatibles, c'est-à-dire identiques à moins que le type gauche soit le type libre.
- si la partie droite est un générateur d'enregistrement (cf. 2 ci-après) et si la partie gauche est de niveau 2, un transfert en zone de liste des n éléments supérieurs de la pile (n étant la taille du type droit) permet d'obtenir au sommet de celle-ci une valeur de niveau 1 relative a l'enregistrement créé.
- le niveau droit ne doit pas être inférieur au niveau gauche moins 1.

Nous distinguerons trois cas d'affectation suivant les types gauche et droit.

Type gauche non libre.

Le niveau droit est d'abord ramené à égalité avec le niveau gauche : c'est à dire qu'il y a m (m étant la différence de ces niveaux) génération de l'instruction.

valeur pile (1)

Ensuite, suivant que la valeur à affecter est de niveau zéro ou non, on génère une affectation portant sur la taille du type considéré ou sur la taille 1.

Exemple :

(entier a, logique b) nt ;

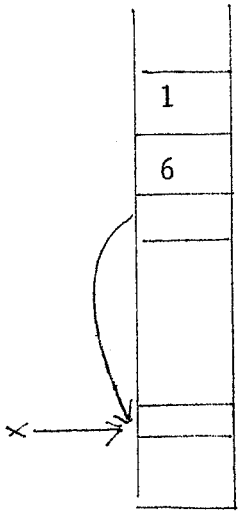
référence nt x ;

x : = a (6,) vrai) ;

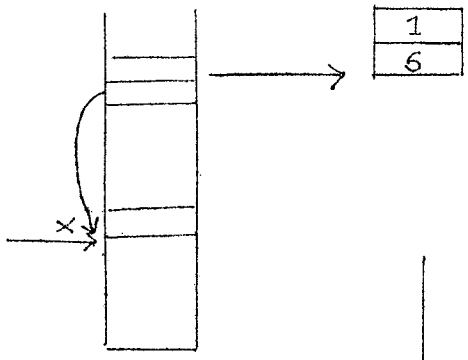
Pour la dernière affectation le programme généré est :

empilerad (<adresse pile de x>) ;
empiler (6) ;
empiler (1) ;
(I)
transfert z la (2,< adresse pile du type nt>)
(II)
affectation (2) ;

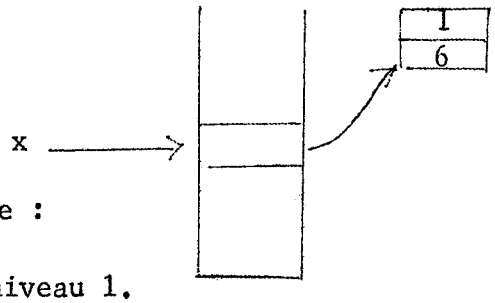
La configuration de la pile en (I) est



en (II) :



et une fois l'affectation achevée :



Soit **y** une variable de type nt et de niveau 1.
L'affectation

```
y := x;
```

se traduit par :

```
empiler ad (⟨adresse de y⟩ );
empiler ad (⟨adresse de x⟩ );
valeur pile (1);
valeur pile (2);
affectation (2);
```

Type gauche libre et type droit non libre

Le niveau droit est tout d'abord ramené au niveau gauche moins 1; si ce niveau droit devient nul, il y a recopie de la valeur du type droit à affecter en zone de liste.

Pour former une valeur libre il reste à empiler l'adresse pile types du type droit.

Exemple : (avec le même type nt que précédemment).

```
    libre y ;
référence nt k;
    k := a(5,vrai);
    y := k;
```

Le programme généré pour la dernière affectation est :

empiler ad (<adresse de y >)

empiler ad (<adresse de k >)

valeur pile (1);

(I)

valeur pile (2);

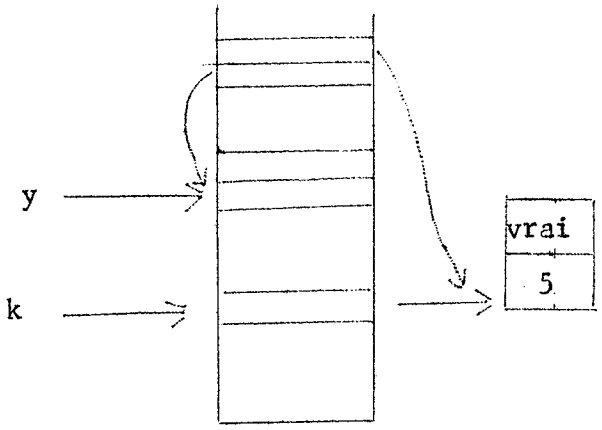
transfert z la (2, <adresse pile de nt>)

empiler (<adresse pile types de nt >)

(II)

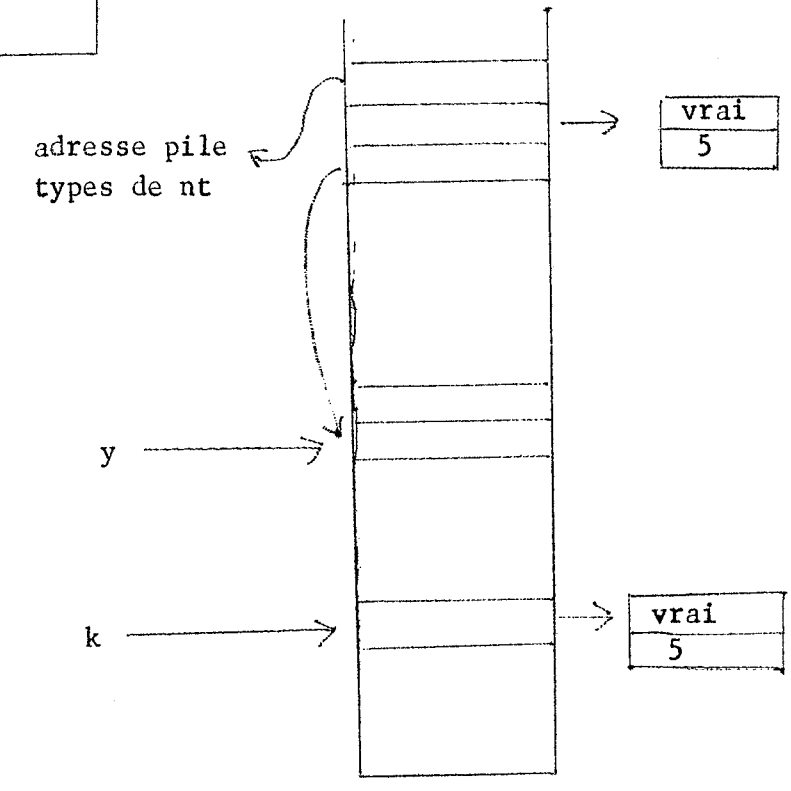
affectation (2)

La configuration de la pile en (I) est :



en (II)

adresse pile types de nt



Types gauche et droit libres

Plusieurs cas sont à envisager :

a) - les niveaux sont ajustés (niveau droit égal au niveau gauche moins 1). Il suffit alors de compléter la valeur à affecter à l'aide de l'adresse pile types du type libre (c'est-à-dire le contenu de la variable dénommée "libre" de l'interpréteur).

Exemple :

```
référence libre x;
          libre y;
            y := 3;
            x := y;
```

Le programme généré pour la dernière affectation est :

```
empiler ad (<adresse de x>);
empiler ad (<adresse de y>);
empiler (libre);
affectation (2);
```

b) - Le niveau gauche est égal à 1.

Si le niveau droit est supérieur à 1, il est possible que la valeur o-terminale de la partie droite soit d'un type non libre : dans ce cas, cette valeur doit être recopiée :

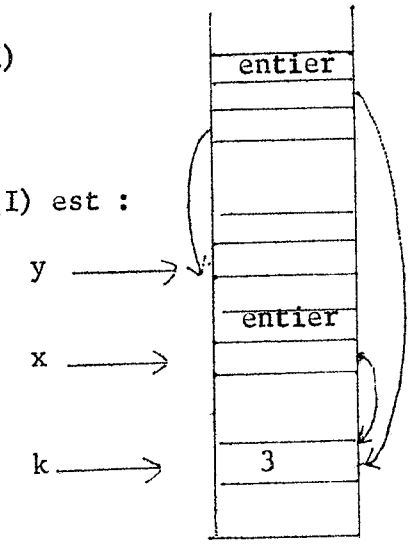
Exemple :

```
      libre y ;
référence libre x ;
      entier k ;
        k : = 3 ;
        x : = k ;
        y : = x ;
```

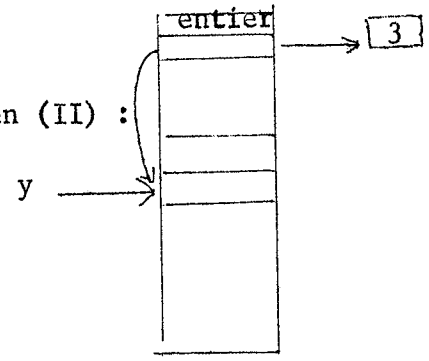
Le programme génèré pour la dernière affectation est :

```
empiler ad (<adresse de y >) ;
empiler ad (<adresse de x>) ;
valeur libre (1) ;          (I)
valeur libre 1 ;            (II)
affectation (2)
```

La configuration de la pile en (I) est :



et en (II) :



Si x avait été une variable de niveau 1, l'instruction valeur libre 1 n'aurait pas été utilisée.

c) Dans les autres cas, l'instruction valeur libre permet d'obtenir au sommet de la pile la valeur à affecter et son type.

Exemple : (cf. III.3.2.)

```
référence libre y;
référence libre x;
        entrer  k;
    x := k;
    y := x;
```

La dernière affectation se traduit par le placement des deux adresses au sommet de la pile et par :

```
valeur libre (1);
affectation (2);
```

```
* e partie-gauche :
    traiter * i * ;
    si niveau considéré < 1  alors
        message "erreur niveau gauche affectation";
    niveau g := niveau considéré ;
    établir ‘type g’ = {‘type considéré’[1]}   #
```

```
* e instruction-d'affectation :
    traiter * i * ;
    traiter * r r i *;
    commentaire cas du nil;
        si niveau considéré := nnil alors
            début
                si niveau g< 2 alors message "erreur affectation nil";
                niveau considéré := niveau g-1;
                établir ‘type considéré’ = {‘type g’[1]}
            fin ;
```

```
commentaire vérification des types ;

    si ¬ (‘type g’ [1] eq ‘type considéré’ [1]  V
          ‘type g’ [1] eq ‘libre’ ) alors
    message "erreur type affectation" ;

commentaire cas d'un générateur d'enregistrement ;
    si niveau considéré = naissance alors

        début

          si  niveau g = 2  alors

            début
                  niveau considéré : =  1 ;
                  générer "transfert z la ("
                          ‘type considéré’ [1,1]" , "
                          ‘type considéré’ [1,4]") ;"sur po

              fin
    sinon niveau considéré : = 0

        fin ;

commentaire vérification des niveaux ;
    si niveau considéré <niveau g-1 alors
            message "erreur niveau affectation" ;
```

```
commentaire affectation libre. libre ;
    si ‘type considéré’ [1] eq ‘libre’ et
       ‘type g’ [1] eq ‘libre’ alors

    début

       si  niveau considéré = niveau g-1 alors
            générer "empiler (libre) ; " sur po
       sinon si niveau g=1 alors

    début
       si niveau considéré ≠ 1 alors
           générer "valeur libre (1); " sur po  sinon
           générer "valeur libre ("niveau considéré-niveau g") ;
                    valeur libre 1 ; " sur po
    fin

sinon
    générer "valeur libre ("
                     niveau considéré-niveau g+1");"sur po ;
générer "affectation (2) ; " sur po
    fin  sinon
```

```
commentaire affectation libre-type connu ;

    si ⌜type g⌝ eq ⌜libre⌝ alors

        début

    desnivl : si  niveau considéré > niveau g-1 alors

        début

            niveau considéré : =  niveau considéré-1 ;
            si niveau considéré = 0 alors
               générer "valeur pile ("
                           ⌜type considéré⌝ [1,1] ") ;
                       transfert z la ("
                         ⌜type considéré⌝ [1,4] ");" sur po

            sinon
                générer "valeur pile (1); " sur po ;
            allera desnivl


        fin ;

    générer "empiler (" ⌜type considéré⌝ [1,5]") ;
            affectation (2) ; " sur po

        fin sinon

début
    commentaire cas général ;
    desniv : si niveau considéré> niveau g alors

        début
           niveau considéré : = niveau considéré-1 ;
           générer "valeur pile (1);" sur po ;
           allera desniv

        fin ;
```

```
si niveau considéré = 0 alors
    générer "valeur pile (" ⟨type considéré⟩ [1,1] ") ;
            affectation  (" ⟨type considéré⟩ [1,1] "); " sur po
sinon
    générer "valeur pile (1) ;
            affectation (1); " sur po

fin
      #
```

2. Générateur d'enregistrement.

On sait que la valeur fournie par l'"évaluation d'un générateur d'enregistrement doit être un nom dont la valeur attribuée est une valeur de nouveau type nouvellement créé. Or, on remarque qu'il est possible que ce nom ne soit pas utilisé : ainsi, dans le cas d'une affectation à une variable de niveau 1.

Exemple :

```
n t x ; x : = n t (5, vrai) ;
```

On construira donc une valeur de nouveau type au sommet de la pile et si, dans le contexte, son nom est exigé on opérera un transfert de cette valeur en zone de liste (cf III.3.4.1.). Ceci évite

- une perte de temps à l'exécution
- un encombrement injustifié de la zone de liste.

Nous avons supposé ici que le chapitre relatif à <composant> est un chapitre d'entrée et qu'à l'intérieur de ce chapitre on ne connait pas le type de la valeur à créer. Dans ce cas, il est possible que les programmes générés pour chaque composant soient justifiables de compléments à l'intérieur du chapitre relatif à <générateur-d'enregistrement>. C'est pourquoi à chaque composant est associé un nouveau segment local $_{pc}$ destiné à s'intercaler dans le segment po après le programme objet traduction du composant. Ce segment est éventuellement rempli dans le chapitre relatif à <générateur-d'enregistrement>.

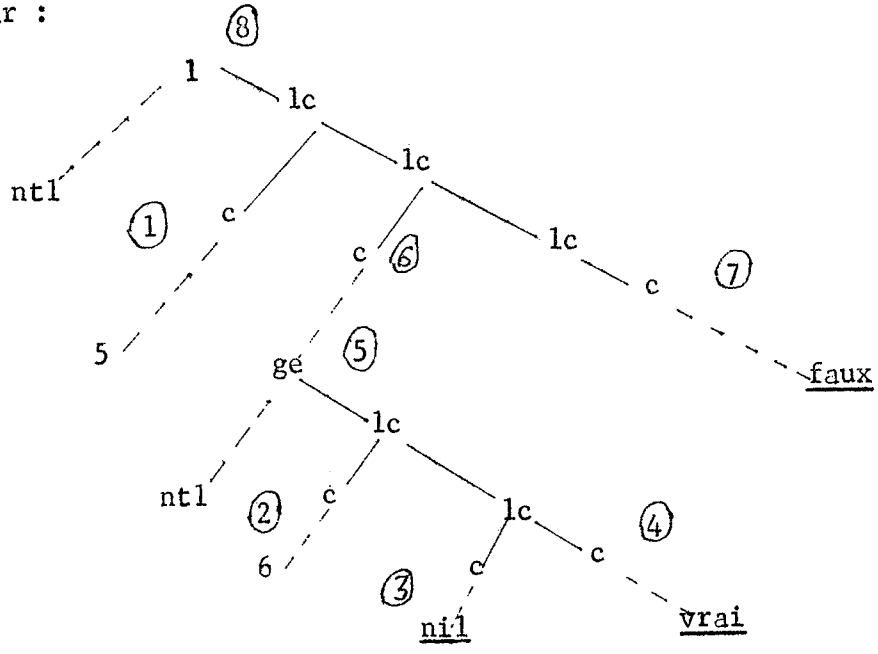
Exemple :

(entier a, référence n t 1 b, logique c) ntl ;

ntl x ;

x : = ntl (5, ntl (6 nil,vrai), faux)

L'arbre syntaxique du générateur d'enregistrement peut être représenté schématiquement par :



où ge, lc et c sont des abréviations de générateur d'enregistrement, liste de composants et composant.

L'ordre d'examen des arbres est indiqué par un numéro attribué aux métavariables qui correspondent à un chapitre d'entrée. Au traitement de l'arbre numéroté 8 le programme généré sur le segment po est :

```
empiler (5) ;
empiler (6) ;
empiler (nil) ;
empiler (1) ;
                (I)
empiler (0)
```

L'effet du chapitre correspondant à cet arbre est de générer sur le segment pc correspondant au deuxième composant l'instruction

transfert z la (3, <adresse pile de nt1>)

et de l'intercaler dans po en (I).

* e composant :

```
traiter * i * ;
établir 'composants créés' = { { 'type considéré' [1],
                                 niveau considéré },
                               description 'composants créés' } ;

nouveau pc ; prévoir pc dans po        #
```

e liste de composants :

```
traiter * i * ;
si * r i * eq ',' alors
```

```
            début
               traiter * r r i * ;
               nb composants : = nb composants + 1
            fin
         sinon nb composants : = 1        #
```

* e générateur-d'enregistrement :

```
         traiter * r r i * ;
         Si nb composants # * i i * [3]  alors
               message "erreur nombre de composants" ;

         k : = o ;
      création : k : = k+1 ;
         établir ⌜tg⌝ = { * i i * [6, k, 1] } ;
         ng : = * i i [6, k,2 ] ;
         établir ⌜td⌝ = { ⌜composants créés⌝ [1,1] };
         nd : = ⌜composants créés⌝ [1,2] ;
```

<suite d'instructions identiques au corps du chapitre relatif à l'instruction d'affectation, mais dans lequel :

- ⌜type g⌝ ,niveau g, ⌜type considéré⌝ , niveau considéré et po sont remplacés respectivement par : ⌜tg⌝, ng, ⌜td⌝, nd et pc.

- aucune instruction "affectation" n'est générée >

```
inclure pc dans po ; ancien pc ;
supprimer 1 de (composants créés) ;
si k ⩽ nb composants alors allera création ;
établir (type considéré) = { * i i * } ;
niveau considéré : = naissance                    #
```

3. Instructions conditionnelles et allera

Les instructions allera provoquent simplement la génération d'une instruction allera (e) où e est une étiquette du programme objet. Notons ici que nous n'avons pas traité les sorties de bloc par de telles instructions.

Les instructions conditionnelles sont traduites en ALGOL 60 en utilisant la procédure booléenne comparer.

```
* e proposition ri :
        traiter * ri * ;
        <réduction du niveau à zéro> ;
        générer "'si' comparer 'alors' 'début'" sur po   #

* e i l :
        traiter * i * ;
        générer " 'fin' 'sinon' 'début' " sur po  #

* e instruction conditionnelle :
        traiter * i * ;
        traiter * r i * ;
        traiter * r r i * ;
        générer "'fin' ; " sur po #
```

III.3.5. Relations

1 Egalité. Infériorité. Supériorité.

Les opérandes gauches sont de types de base, leur niveau est ramené à zéro dans le chapitre correspondant à l'opérateur associé (cf III.3.3.3.). Il reste à faire de même pour l'opérande droit et à générer l'instruction correspondant à l'opérateur.

*e égalité :

```
traiter * i * ;
traiter * ri * ;
traiter * r r i * ;
    <réduction du niveau à zéro>
générer "égal; " sur po   #
```

* e supinf :

```
traiter * i * ;
traiter * r r i * ;
    <réduction du niveau à zéro> ;
si  * i r i * eq 'inf' alors
     générer "infériorité;" sur po sinon
     générer "supériorité;" sur po      #
```

2 Identité

La condition d'identité des modes est d'abord vérifiée : si elle est satisfaite, on procède à la génération de l'instruction égal.

```
* e opid :
    établir ⌜type i⌝ = { ⌜type considéré [1]⌝ };
    niveau i : = niveau considéré #

* e identité :

    traiter * i * ;
    traiter * r i * ;
    traiter * r r i * ;

    Si niveau considéré ≠ nnil ∧ niveau i ≠ nnil alors

        début
                si ¬ ( ⌜type i⌝ [1]  eq  ⌜type considéré⌝ [1]
                                                    alors
                   message "erreur type identité" ;
                si niveau i ≠ niveau considéré alors
                   message "erreur niveau identité"
        fin ;

générer "égal; " sur po      #
```

3 Conformité.

Les vérifications à effectuer concernent :

- le niveau de la partie gauche. Il doit en effet être supérieur à 1.

- la différence des niveaux gauche et droit, qui sont soumis à la même contrainte que dans une affectation.

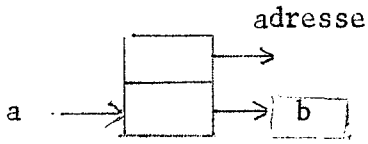
Soient m et n les niveaux gauche et droit. On peut distinguer trois phases dans une conformité :

- recherche du type de la valeur m-2 terminale de la partie droite. L'instruction valeur libre permet d'obtenir l'adresse pile de ce type au sommet de la pile.

Toutefois, du fait de la représentation des variables libres, le cas où une valeur o-terminale est demandée, avec n>1 est spécial : en effet, il faudra déterminer à l'exécution si la valeur 1-terminale est d'un type libre ou d'un type non-libre. Si cette valeur est le nom a et sa valeur attribuée b, dans la première hypothèse sa représentation est



dans la seconde :



- affectation de la valeur m-2 terminale de la partie droite à la valeur m-1 terminale de la partie gauche si leurs types sont identiques.

Enfin, placement d'un 1 au sommet de la pile dans le cas où l'affectation s'est produite : dans le cas contraire, effacement des opérandes et placement d'un 0.

```
* e op conf :
    si niveau considéré < 2 alors
            message "erreur niveau gauche conformité"
    générer "valeur pile (1);" sur po;
    établir ‘type C’ = { type considéré [1] } ;
    niveau c : = niveau considéré                    #
```

```
* e conformité :

    traiter * i * ;
    traiter * r r i *;

    si niveau considéré <niveau c-1 alors
            message "erreur niveau conformité" ;

commentaire ajustement des niveaux ;

    si niveau c = 2     alors

       début
            si niveau considéré = 1 alors
                générer "valeur libre (1); " sur po
            sinon
                générer "valeur libre ("niveau considéré-niveau c");
                        égalité (libre) ;
                        'Si' comparer 'alors'
                        valeur libre (2) 'sinon' valeur libre (1);"
                                                    sur po

       fin

  sinon générer "valeur libre" ("niveau considéré-niveau c+1")";
                                                    sur po ;

commentaire vérification du type et affectation ;
générer "égalité  (" 'type c' [1,5]")  ;
      'si' comparer 'alors' 'début' " sur po ;
si niveau c = 2 alors
   générer "valeur pile (" 'type c'  [1,1] ") ;
            affectation (" 'type c' [1,1] "); " sur po
```

```
sinon

    générer "valeur pile (1)  ;
            affectation (1) ; "sur po ;

générer "empiler (1) 'fin' 'sinon'
    'début' désempiler (2) ; empiler (0) 'fin' ; "sur po
#
```

Exemple :

```
référence entier x  ;
entier j; entier k  ;
référence libre  y  ;
          j : = 2  ;
          x : = j  ;
          k : = 3  ;
          y : = k  ;
    .....x. = y ....
```

Le programme généré pour cette conformité (on considére ici que les symboles entre apostrophes sont des symboles de base ALGOL)est :

```
empiler ad (<adresse de x>) ;
valeur pile (1) ;
empiler ad (<adresse de y>) ;

valeur libre (0) ;
                        (I)
égalité (libre) ;

si comparer alors valeur libre (2) sinon valeur libre (1) ;
                        (II)
Egalité (<adresse pile types du type entier>) ;
```

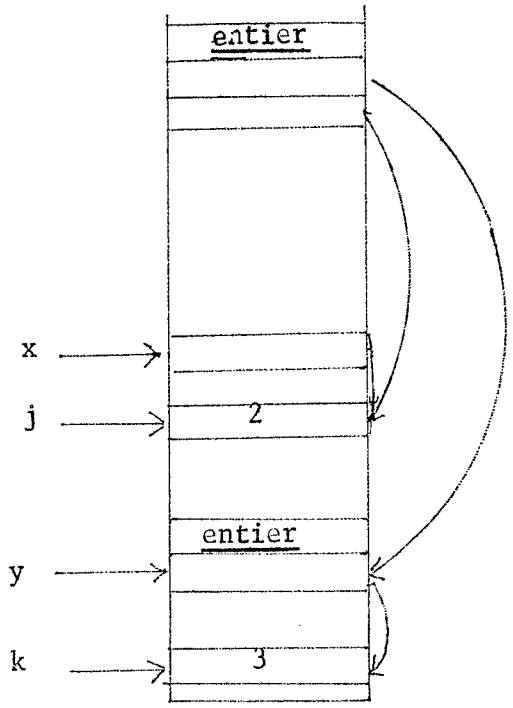
si comparer alors début

valeur pile (1) ;
affectation (1) ;
empiler (1)

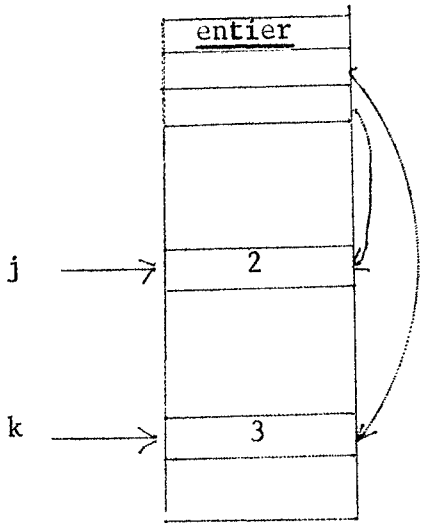
fin sinon

début désempiler (2) ; empiler (0) fin ;

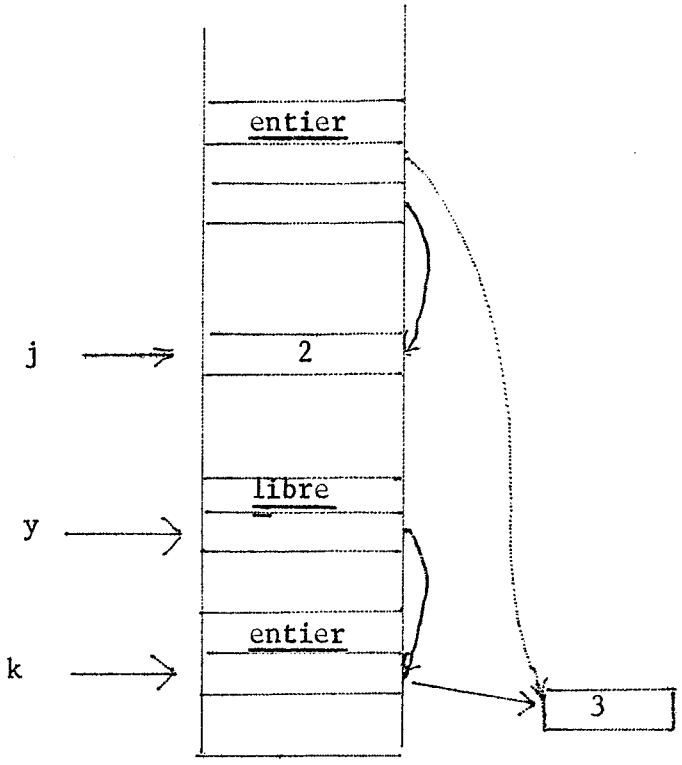
La configuration de la pile en (I) est :



Après l'exécution de valeur libre (1), la configuration en (II) est :



Remarquons que si k avait été une variable libre de niveau 1, nous aurions eu en (II) :



III.4. Gestion de la mémoire.

Les critères à respectuer sont les suivants :

-encombrement minimum.

-rapidité d'exécution acceptable

-élimination aisée des éléments non intéressants, c'est à dire non accessibles à un instant donné de l'exécution :

- soit en fin de validité d'un type
- soit par "récupération" (garbage collection).

Parmi les méthodes d'allocation dynamique de la mémoire (dynamic storage allocation) applicables à notre problème, on peut dégager deux voies principales :

- réservation à la zone de liste de la zone de mémoire située à l'extrémité de celle contenant la pile.

- division de la mémoire en pages, chacune d'entre elles contenant soit des éléments de pile, soit des enregistrements de même type.

La première méthode a été étudiée par C.A.R HOARE [Hoa] pour traiter les structures qu'il est possible de définir dans le langage décrit en [WiH] . Schématiquement, ces structures représentent un cas particulier des nouveaux types : celui où toutes les variables de nouveau type sont de niveau égal à 2 et où il n'existe pas d'autres variables que celles-ci de niveau supérieur à 1.

On peut décrire ainsi la gestion proposée pour la zone de liste dans cette étude :

- tous les emplacements libres de même taille sont reliés entre eux. A la création d'un enregistrement d'une certaine taille, celui-ci est transféré

  - soit dans un emplacement de même taille
  - soit dans un emplacement de taille supérieure auquel cas les éléments non utilisés sont replacés dans la chaîne de liaison correspondante.

- Dans le cas où il n'existe pas de tels emplacements, une récupération est déclenchée. Elle consiste à repérer tous les éléments utiles (c'est à dire accessibles par un certain nom) et à enchaîner entre eux les éléments inutiles.

- Au cas où cette récupération ne libérerait pas un emplacement suffisant pour l'enregistrement, un dispositif ingénieux introduit par Hoare permet de ne "retasser" la zone de liste qu'en dernière extrémité. Succintement, un "parapluie" (évaluant dynamiquement) est disposé au-dessus de la pile de telle sorte qu'on ne peut placer un enregistrement au dessous (c'est à dire entre le sommet de la pile et ce parapluie) que si une récupération n'a donné aucun résultat.

- Enfin, si même entre le sommet de la pile et la zone de liste, on ne peut placer l'enregistrement, un retassement est déclenché. Son effet est de "comprimer" tous les éléments utiles à l'extrémité de la zone de liste et de libérer ainsi la plus grande tranche d'éléments de mémoire possible.

L'inconvénient majeur d'une telle approche vient du fait que l'élimination de valeurs nouvelles en fin de validité du type associé nécessite une récupération générale.
Nous avons étudié la seconde méthode évoquée plus haut.
Pour ceci, nous utilisons une mémoire secondaire à accès lent.
L'apparition des pages en mémoire rapide déterminée comme dans [Col] étant étant déterminée. On verra que la division en pages permet de libérer aisément les enregistrements en fin de validité de leur type et même d'envisager d'effectuer des récupérations et des retassements partiels puisque les zones occupées par les enregistrements de différents types sont localisées. Toutefois, le défaut inhérent à cette solution est la perte de vitesse d'exécution due au fait que l'accès à un opérande nécessite un test sur la présence d'une page en mémoire rapide.

III.4.1. Organisation générale

Pages piles

Les pages utilisées pour chaque pile sont reliées entre elles symétriquement à l'aide des deux éléments d'adresses inférieures de chaque page .
En effet, des valeurs de taille supérieure à 1 sont composées d'éléments contigus qui peuvent se trouver dans des pages différentes, d'où la nécessité pour chacune de celles-ci, de connaître son successeur et son prédécesseur.

Pages contenant des enregistrements

Chacune d'entre elles est reliée à celle précédemment utilisée pour contenir des enregistrements du même type : de plus, leur deuxième élément contient le "module" du type associé c'est à dire : taille du type plus 1.
Les emplacements libres (s'il en existe) forment une "liste libre", c'est à dire que chacun d'eux contient l'adresse du suivant.

Pages libres.

L'indicateur "pages libres" repère la première page non utilisée. Les pages disponibles sont reliées entre elles.
L'adresse pile (c'est à dire le premier élément dans la pile type relatif à ce type) se réfère à deux éléments de la pile normale :

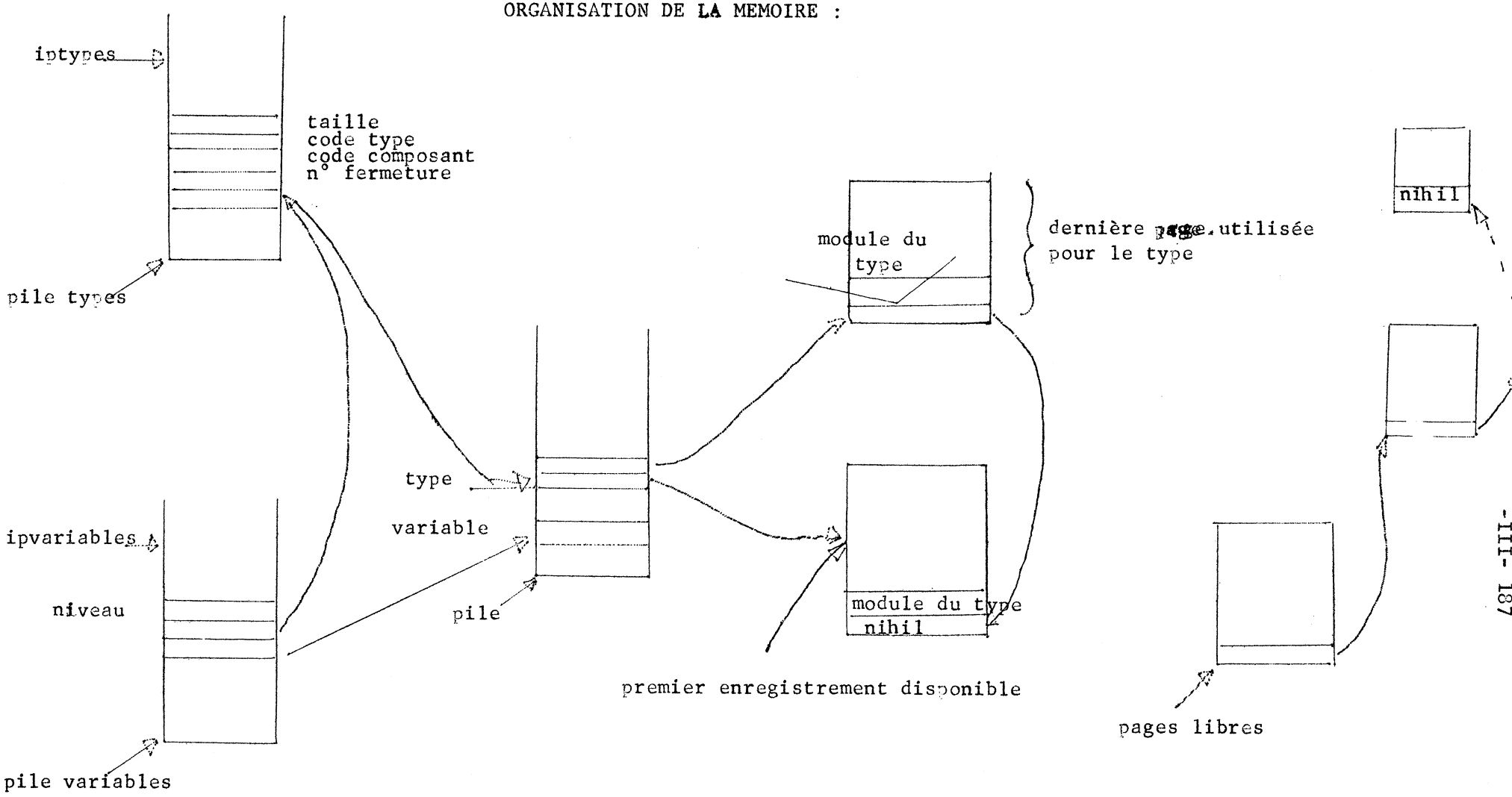
- Le premier indique l'emplacement prévu pour ranger le prochain enregistrement de ce type devant être transféré en zone de liste.
A chaque rangement, l'adresse contenue dans l'emplacement-hôte est affectée à ce premier élément.
Dans le cas où aucun emplacement libre n'existe, on doit attribuer une nouvelle page aux enregistrements du type considéré.

- le second élément indique la dernière page détachée de la liste des pages libres pour pourvoir au rangement des enregistrements.
Cette information est nécessaire

- pour rendre les pages utilisées à la liste des pages libres en fin de validité du type.

- pour permettre un "retassement" de ces pages (cf III.4.3.).

ORGANISATION DE LA MEMOIRE :



III.4.2. Récupération.

Une récupération est déclenchée :

-lorsqu'aucun emplacement n'est disponible dans les pages réservées à son type.

-lorsque le nombre de pages libres est inférieur à un certain palier, dit "nombre critique". En effet, la phase de marquage d'une récupération s'effectuant récursivement, la pile s'étendra durant son déroulement.
Le processus de récupération que nous avons adopté est largement inspiré de celui utilisé pour le langage LISP [McC] . Il procède en deux phases.

Marquage.

Le problème est de marquer tous les enregistrements qui sont accessibles : un élément spécial est rajouté ici à tout enregistrement présent dans la zone de liste, suivant sa valeur 0 ou 1 l'enregistrement est dit non marqué ou marqué.

Pour chaque type, une procédure marque a été générée. Son action est la suivante :

- étant donnée l'adresse d'une valeur de ce type et de niveau zéro au sommet de la pile.

- si cette valeur n'est pas dans la pile, il y a marquage soit de l'enregistrement formé par cette valeur, soit de l'enregistrement contenant cette valeur. Ceci est l'action de l'instruction objet "noter".

-Tous les composants sont ensuite examinés. L'effet de l'instruction "composant 1" est de placer au sommet de la pile l'adresse d'un composant (sans effacer l'adresse de l'enregistrement actuellement considéré). Suivant que ce composant est de niveau 1, de niveau supérieur à 1 ou de type libre, on exécute :

- la procédure marque correspondant à ce type.
- l'instruction :

marquer ref (<code du type du composant>,
<niveau du composant≥)

L'action de celle-ci est de placer la valeur du composant au sommet de la pile et de la rendre négative à son occurence dans le composant. Si le niveau de cette valeur est égal à 1 la procédure marque correspondant au type de cette valeur est appellée ; dans le cas contraire, il y a soit marquage de l'enregistrement qu'elle désigne soit marque de celui le contenant et cette valeur est à nouveau considérée par marquer ref.

- l'instruction

marquer libre <niveau>

Son effet est analogue à celui de marquer ref. Toutefois, aussitôt qu'une valeur i-terminale (i<niveau de la valeur au sommet de la pile) est d'un type donné on appelle marquer ref ou la procédure marque correspondant à ce type (i=0).

Une fois que tous les composants on été traités, il reste à désempiler l'adresse initiale de l'enregistrement qui se trouvait au sommet de la pile.

Exemple :

pour le nouveau type ntb tel que :

(entier ceb, référence nta cnb, logique cnb) ntb

la procédure marque associée est la suivante :

```
procédure marque 4 ;
    début noter (3,et 4) ;
          composant 1 (3) ;
          marquer 2  ;
          composant 1 (2)  ;
          marquer ref (3,2)  ;
          composant 1 (1)
          marquer 1  ;
      et 4:désempiler (1)
    fin ;
```

Une phase de marquage procède à l'examen de toutes les informations relatives aux variables contenues dans pile variable. On dispose pour chacune d'entre elles de leur adresse pile, de leur adresse pile types de son type et de leur niveau. Ceci permet d'activer comme pour un composant suivant leur type et leur niveau, une procédure marque, une instruction marquer ref ou une instruction marquer libre.

Remarque 1 *

De nombreuses améliorations sont possibles pour cette phase. On remarque en particulier ici que chaque fois qu'une procédure marque est appellée de l'intérieur d'une autre procédure marque le composant considéré pour cette dernière est à nouveau marqué. Ceci est du au fait que lorsqu'une procédure marque est appellée depuis "marquer ref" il faut marquer l'enregistrement désigné ou celui dans lequel il est contenu. On pourrait donc éviter un marquage redondant en utilisant dans "marquer ref" l'information relative à la taille des types contenue dans la pile types.

Formation de listes libres.

Une fois que les éléments utiles ont été marqués, comme dans McC on établit un chaînage ou "liste libre" entre tous les élémentsinutiles et on démarque les autres. Toutefois, dans le dernier cas, toutes les valeurs de niveau supérieur à 1 qui auraient été rendues négatives sont restaurées. Ceci est effectué à l'aide d'un parcours de toutes les pages appartenant à chaque type : le "code composant" qu'il contient parmi ses informations en pile types fournit en effet les adresses composant de chaque composant d'enregistrement correspondant et de niveau supérieur à 1 (donc susceptible d'avoir été rendu négatif).

III.4.3. Retassement.

Un retassement est déclenché lorsqu'aucune page n'est disponible, c'est à dire lorsque le nombre de pages libres est "critique". Ceci peut se produire :

- soit à la suite d'une demande de pages pour les piles.
- soit à la suite d'une récupération si celle-ci n'a donné lieu à aucune liste libre.

Le schéma adopté choisi ici est une adaptation de celui décrit en [Cot 1] . Il se décompose en deux phases.

Remplissage.

Les pages appartenant aux différents types valides sont considérées successivement.
Succintement, on place le maximum d'enregistrements utiles appartenant aux dernières pages créées dans les emplacements non marqués des pages les plus anciennes.
A l'emplacement abandonné on laisse l'adresse du nouvel enplacement.
Les éléments marqués, déplacés ou non, sont démarqués.

Remplacement.

Tous les contenus des variables de niveau m supérieur à 2 présentes dans la pile ou dans la zone de liste doivent être examinés pour déterminer si leur valeur m-1 terminale n'a pas changé d'adresse.

Cette information est fournie par le deuxième élément de chaque page, le contenu de celui-ci est négatif si la page a contenu des enregistrements transplantés et sa valeur absolue est l'adresse au dessus de laquelle se sont trouvés ces enregistrements : ceci est effectué par la précédente phase.
Dans le cas où une valeur m-1 terminale a changé d'emplacement, le nouvel emplacement est contenu à l'adresse de l'ancien.

Remarque :

Du fait que les enregistrements de même type sont essentiellement dans des pages différentes, on pourrait envisager d'effectuer des récupérations et des retassements partiels. Ainsi lorsque la demande d'un enregistrement de type a ne pourrait être satisfaite, on ne marquerait pas systèmatiquement tous les enregistrements, mais seulement une partie d'entre eux dite "fermeture" correspondant à a.
Dans notre implantation, cette possibilité est réduite du fait de la généralité des procédures marque attribuées à chaque type.
En effet

a) Supposons que les enregistrements de type b contiennent des valeurs de niveau supérieur à 0 et de type a.
Si les enregistrements de type a sont à marquer, on est amené à considérer ceux de b et par contrecoup à les marquer aussi.
Si ceux de b sont à marquer, il n'est pas à priori nécessaire de marquer ceux de a sauf s'ils contiennent des valeurs susceptibles d'aboutir à la désignation d'un enregistrement de b. Cependant, si les programmes marque de b sont activés, ceci entraîne aussi l'activation de ceux de b.

b) Supposons que les enregistrements de b contiennent une valeur de niveau 0 et de type a :

Si les enregistrements de type a sont à marquer, ceux de b peuvent l'être aussi, même si ceux de a ne contiennent pas de valeur de type b et de niveau supérieur à 0. Plutôt que d'avoir à démarquer ces derniers ensuite, il est préférable dans tous les cas d'activer les procédures marque associées à b.

Si les enregistrements de type b sont à marquer, il est nécessaire de marquer aussi ceux de type a : en effet, il faut éviter de libérer des enregistrements b contenant des enregistrements a valides.

On remarque donc que pour effectuer des marquages sur des ensembles de types, il faudrait dans notre implantation que les ensembles soient disjoints.

III.5. Réalisations pratiques.

1. Un compilateur expérimental a été écrit en ALGOL 60 pour les langages source et objet décrits en III.1 et III.2.
Il procède en plusieurs phases successives :

- codage de la grammaire. Ce programme a été réalisé par F. MARTIN [Mar] .
- codage de la chaîne source en fonction des codes attribués aux éléments terminaux par la précédente phase [Mar].
- Analyse syntaxique de la chaîne source. L'analyseur utilisé est dérivé de celui d'Irons[Ir]: il est décrit en[BCo] Cet analyseur procède par essais successifs et le résultat de l'analyse complète de la chaîne est contenu dans une pile. Le contenu de celle-ci est ensuite transformé en un arbre sous forme d'une liste LISP à l'aide de la procédure "construction" décrite dans [Co].
- Exploitation de cet arbre syntaxique par un programme ALGOL déduit de ceux présentés en III.3 suivant lesprincipes indiqués en II.2.2.

On trouvera dans les pages qui suivent, le déroulement du traitement de la chaîne source suivante :

```
début
     (entier cea, logique cla, référence nta cna)nta ;
      nta nva ;
      référence nta nvb ;
      nva := nta(5, vrai, nta(6, faux, nil)) ;
      nvb := cna de nva ;
      si valeur nvb est nva alors allera ea
         sinon () ;
      ea :
fin
```

Les instructions objet sont imprimées au fur et à mesure de leur production. Les entrées et sorties des chapîtres les plus significatifs et le contenu des descriptions les plus intéressantes sont indiqués.

On peut noter en fin d'élaboration du générateur d'enregistrement

```
nta(5, vrai, nta(6, faux, nil))
```

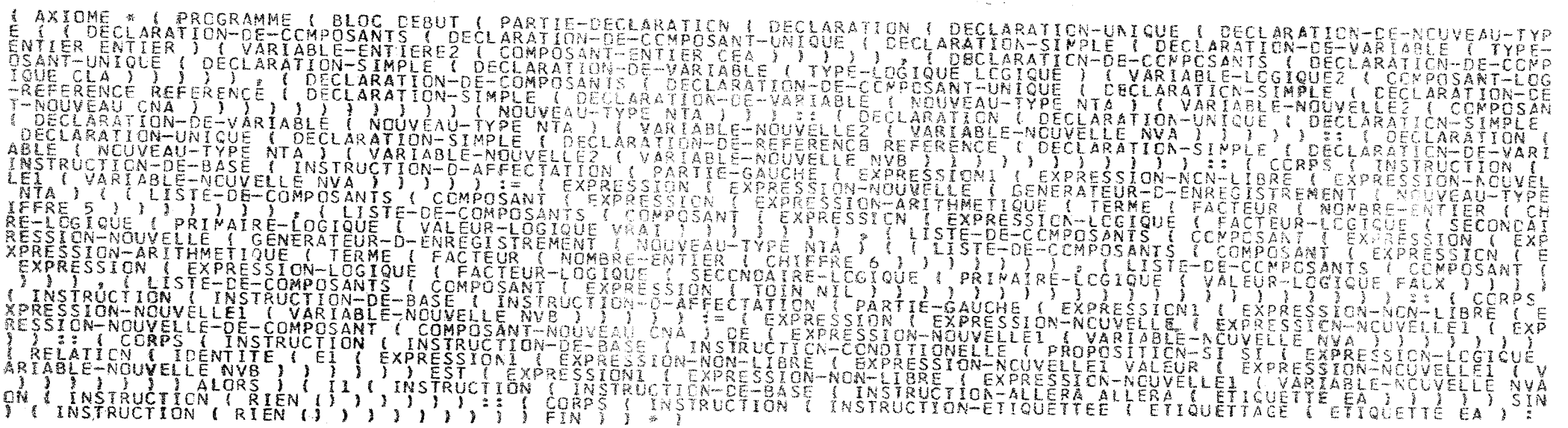
la production sur le segment local pc correspondant à son troisième composant de l'instruction

```
transfert z la(3, 7)
```

On retrouve celle-ci incluse dans le programme objet : elle opère le transfert en zone de liste de l'enregistrement

```
nta(6, faux, nil)
```

ARBRE SYNTAXIQUE:


(AXIOME * (PROGRAMME (BLOC DEBUT (PARTIE-DECLARATION (DECLARATION (DECLARATION-UNIQUE (DECLARATION-DE-NOUVEAU-TYP
E ((DECLARATION-DE-COMPOSANTS (DECLARATION-DE-COMPOSANT-UNIQUE (DECLARATION-SIMPLE (DECLARATION-DE-VARIABLE (TYPE-
ENTIER ENTIER) (VARIABLE-ENTIERE2 (COMPOSANT-ENTIER CEA)))) , (DECLARATION-DE-COMPOSANTS (DECLARATION-DE-COMP
OSANT-UNIQUE (DECLARATION-SIMPLE (DECLARATION-DE-VARIABLE (TYPE-LOGIQUE LOGIQUE) (VARIABLE-LOGIQUE2 (COMPOSANT-LOG
IQUE CLA))))) , (DECLARATION-DE-COMPOSANTS (DECLARATION-DE-COMPOSANT-UNIQUE (DECLARATION-SIMPLE (DECLARATION-DE
-REFERENCE REFERENCE (DECLARATION-SIMPLE (DECLARATION-DE-VARIABLE (NOUVEAU-TYPE NTA) (VARIABLE-NOUVELLE2 (COMPOSAN
T-NOUVEAU CNA))))))))) (NOUVEAU-TYPE NTA))) :: (DECLARATION (DECLARATION-UNIQUE (DECLARATION-SIMPLE
(DECLARATION-DE-VARIABLE (NOUVEAU-TYPE NTA) (VARIABLE-NOUVELLE2 (VARIABLE-NOUVELLE NVA))))) :: (DECLARATION (
DECLARATION-UNIQUE (DECLARATION-SIMPLE (DECLARATION-DE-REFERENCE REFERENCE (DECLARATION-SIMPLE (DECLARATION-DE-VARI
ABLE (NOUVEAU-TYPE NTA) (VARIABLE-NOUVELLE2 (VARIABLE-NOUVELLE NVB)))))))) :: (CORPS (INSTRUCTION (
INSTRUCTION-DE-BASE (INSTRUCTION-D-AFFECTATION (PARTIE-GAUCHE (EXPRESSION1 (EXPRESSION-NCN-LIBRE (EXPRESSION-NOUVEL
LE1 (VARIABLE-NOUVELLE NVA))))) := (EXPRESSION (EXPRESSION-NOUVELLE (GENERATEUR-D-ENREGISTREMENT (NOUVEAU-TYPE
NTA) ((LISTE-DE-COMPOSANTS (COMPOSANT (EXPRESSION (EXPRESSION-ARITHMETIQUE (TERME (FACTEUR (NOMBRE-ENTIER (CH
IFFRE 5))))))) , (LISTE-DE-COMPOSANTS (COMPOSANT (EXPRESSION (EXPRESSION-LOGIQUE (FACTEUR-LOGIQUE (SECONDAI
RE-LOGIQUE (PRIMAIRE-LOGIQUE (VALEUR-LOGIQUE VRAI)))))) , (LISTE-DE-COMPOSANTS (COMPOSANT (EXPRESSION (EXP
RESSION-NOUVELLE (GENERATEUR-D-ENREGISTREMENT (NOUVEAU-TYPE NTA) ((LISTE-DE-COMPOSANTS (COMPOSANT (EXPRESSION (E
XPRESSION-ARITHMETIQUE (TERME (FACTEUR (NOMBRE-ENTIER (CHIFFRE 6))))))) , (LISTE-DE-COMPOSANTS (COMPOSANT (
EXPRESSION (EXPRESSION-LOGIQUE (FACTEUR-LOGIQUE (SECONDAIRE-LOGIQUE (PRIMAIRE-LOGIQUE (VALEUR-LOGIQUE FAUX)))
))) , (LISTE-DE-COMPOSANTS (COMPOSANT (EXPRESSION (TOIN NIL)))))))))

```
GENERATION SUR PC DE :
'DEBUT'

GENERATION SUR PC DE :
( IGESTION:: ISYSTEME:: )

GENERATION SUR PC DE :
DEPART::

                    TRAITER DE : BLOC

                    TRAITER DE : PARTIE-DECLARATION
                    TRAITER DE : DECLARATION-DE-NOUVEAU-TYPE

GENERATION SUR PC DE :
( EMPILER( NIHIL ):: )

GENERATION SUR PC DE :
( EMPILER( NIHIL ):: )

COMPOSANTS DE LA DESCRIPTION DENOMMEE NTA :
        0     3     3     7     0      (      CNA      CLA      CEA )
                    FIN DE DECLARATION-DE-NOUVEAU-TYPE
    ...
                    TRAITER DE : DECLARATION-SIMPLE
                    FIN DE DECLARATION-SIMPLE
    ...

COMPOSANTS DE LA DESCRIPTION DENOMMEE NVA :
        NTA     1
                    TRAITER DE : DECLARATION-SIMPLE
                    FIN DE DECLARATION-SIMPLE
    ...

COMPOSANTS DE LA DESCRIPTION DENOMMEE NVB :
        NTA     2

COMPOSANTS DE LA DESCRIPTION DENOMMEE NTA :
        3     3     3     7     0      (      CNA      CLA      CEA )

GENERATION SUR PMARQUAGE DE :
( 'PROCEDURE' MARQUE 3 'DEBUT' )

GENERATION SUR PMARQUAGE DE :
( NOTER( 3 , ET 3 ):: )

COMPOSANTS DE LA DESCRIPTION DENOMMEE COMPOSANT :
        NTA     2     3

COMPOSANTS DE LA DESCRIPTION DENOMMEE NTA :
        3     3     3     7     0      (      CNA      CLA      CEA )

GENERATION SUR PMARQUAGE DE :
( COMPOSANT1( 3 ):: )

GENERATION SUR PMARQUAGE DE :
( MARQUERREF( 3 , 2 ):: )

COMPOSANTS DE LA DESCRIPTION DENOMMEE COMPOSANT :
        LOGIQUE     1     2

COMPOSANTS DE LA DESCRIPTION DENOMMEE LOGIQUE :
        1     2     0     3     6      NIL

GENERATION SUR PMARQUAGE DE :
( COMPOSANT1( 2 ):: )

GENERATION SUR PMARQUAGE DE :
( MARQUE 2 :: )

COMPOSANTS DE LA DESCRIPTION DENOMMEE COMPOSANT :
        ENTIER     1     1

COMPOSANTS DE LA DESCRIPTION DENOMMEE ENTIER :
        1     1     3     1     1      NIL

GENERATION SUR PMARQUAGE DE :
( COMPOSANT1( 1 ):: )

GENERATION SUR PMARQUAGE DE :
( MARQUE 1 :: )

GENERATION SUR PC DE :
( EMPILERTYPE( 7 , 0 , 3 , 3 , 2 ):: )

COMPOSANTS DE LA DESCRIPTION DENOMMEE NTA :
        3     3     3     7     16      (      CNA      CLA      CEA )

GENERATION SUR PMARQUAGE DE :
( ET 3 : DESEMPILER( 1 ) 'FIN' :: )

LE SEGMENT  PMARQUAGE CONTIENT :
    'PROCEDURE'MARQUE3
                    'DEBUT'NOTER(3,ET3)::
                    COMPOSANT1(3)::
                    MARQUERREF(3,2)::
                    COMPOSANT1(2)::
                    MARQUE2::
                    COMPOSANT1(1)::
                    MARQUE1::
                    ET3:DESEMPILER(1)'FIN'
                ::


VARIABLE OUBLIEE:GENERATION SUR PO DE :
( EMPILER( NIHIL ):: )

GENERATION SUR PC DE :
( EMPILERVARIABLE( 9 , 16 , 2 ):: )

VARIABLE OUBLIEE:GENERATION SUR PO DE :
( EMPILER( NIHIL ):: )

GENERATION SUR PC DE :
( EMPILER( NIHIL ):: )

GENERATION SUR PC DE :
( EMPILER( NIHIL ):: )

GENERATION SUR PC DE :
( EMPILERVARIABLE( 10 , 16 , 1 ):: )
                    FIN DE PARTIE-DECLARATION
    ...
```

```
                    TRAITER DE : INSTRUCTION-D-AFFECTATION
GENERATION SUR PC DE :
( EMPILERAD( 10 ):: )
                    TRAITER DE : GENERATEUR-D-ENREGISTREMENT
GENERATION SUR PC DE :
( EMPILER( 5 ):: )
GENERATION SUR PC DE :
( EMPILER( 1 ):: )
                    TRAITER DE : GENERATEUR-D-ENREGISTREMENT
GENERATION SUR PC DE :
( EMPILER( 6 ):: )
GENERATION SUR PC DE :
( EMPILER( 0 ):: )
GENERATION SUR PC DE :
( EMPILER( NIL ):: )

COMPOSANTS DE LA DESCRIPTION DENOMMEE COMPOSANTS-CREES :
        (       TNIL      20 )        (       LOGIQUE      0 )        (       ENTIER      0 )        (       LOGIQUE      0 )
   (       ENTIER      0 )

COMPOSANTS DE LA DESCRIPTION DENOMMEE CNA :
        NTA      2      3

COMPOSANTS DE LA DESCRIPTION DENOMMEE CLA :
        LOGIQUE      1      2

COMPOSANTS DE LA DESCRIPTION DENOMMEE CEA :
        ENTIER      1      1
                    FIN DE GENERATEUR-D-ENREGISTREMENT
    ...

COMPOSANTS DE LA DESCRIPTION DENOMMEE COMPOSANTS-CREES :
        (       NTA      21 )        (       LOGIQUE      0 )        (       ENTIER      0 )

COMPOSANTS DE LA DESCRIPTION DENOMMEE CNA :
        NTA      2      3

GENERATION SUR PC DE :
( TRANSFERTZLA( 3 , 7 ):: )

COMPOSANTS DE LA DESCRIPTION DENOMMEE CLA :
        LOGIQUE      1      2

COMPOSANTS DE LA DESCRIPTION DENOMMEE CEA :
        ENTIER      1      1
                    FIN DE GENERATEUR-D-ENREGISTREMENT
    ...
 NIVEAU GAUCHE: 1
NIVEAU CONSIDERE: NAISSANCE
GENERATION SUR PC DE :
( AFFECTATION( 3 ):: )
                    FIN DE INSTRUCTION-D-AFFECTATION
    ...
                    TRAITER DE : INSTRUCTION-D-AFFECTATION
GENERATION SUR PC DE :
( EMPILERAD( 9 ):: )
                    TRAITER DE : EXPRESSION-NOUVELLE-DE-COMPOSANT
GENERATION SUR PC DE :
( EMPILERAD( 10 ):: )
GENERATION SUR PC DE :

( COMPOSANT( 3 ):: )
                    FIN DE EXPRESSION-NOUVELLE-DE-COMPOSANT
    ...
 NIVEAU GAUCHE: 2
NIVEAU CONSIDERE: 2
GENERATION SUR PC DE :
VALEURPILE(1)::
GENERATION SUR PC DE :
( AFFECTATION( 1 ):: )
                    FIN DE INSTRUCTION-D-AFFECTATION
    ...
```

TRAITER DE : IDENTITE

```
GENERATION SUR PC DE :
( EMPILERAD( 9 ):: )

GENERATION SUR PC DE :
VALEURPILE(1)::

GENERATION SUR PC DE :
( EMPILERAD( 10 ):: )

GENERATION SUR PC DE :
( EGAL:: )

                FIN DE IDENTITE
    ...

GENERATION SUR PC DE :
( 'SI' COMPARER 'ALORS' 'DEBUT' )

GENERATION SUR PC DE :
( 'ALLERA' EA :: )

GENERATION SUR PC DE :
( 'FIN' 'SINON' 'DEBUT' )

GENERATION SUR PC DE :
( 'FIN' :: )

GENERATION SUR PC DE :
( EA : )

COMPOSANTS DE LA DESCRIPTION DENOMMEE NTA :
        3     3     3     7     16      (      CNA      CLA      CEA )

GENERATION SUR PC DE :


( LIBERERTYPE( 7 ):: )

                FIN DE BLOC
    ...

GENERATION SUR PC DE :
'FIN'
```

PROGRAMME OBJET :

LE SEGMENT PO CONTIENT :

```
'DEBUT''PROCEDURE'MARQUE3
    'DEBUT'NOTER(3,ET3)::
    COMPOSANT1(3)::
    MARQUERREF(3,2)::
    COMPOSANT1(2)::
    MARQUE2::
    COMPOSANT1(1)::
    MARQUE1::
    ET3:DESEMPILER(1)'FIN'
::
IGESTION::ISYSTEME::DEPART::
EMPILER(NIHIL)::
EMPILER(NIHIL)::
EMPILERTYPE(7,0,3,3,3)::
EMPILER(NIHIL)::
EMPILERVARIABLE(9,16,2)::
EMPILER(NIHIL)::
EMPILER(NIHIL)::
EMPILER(NIHIL)::
EMPILERVARIABLE(10,16,1)::
EMPILERAD(10)::
EMPILER(5)::
EMPILER(1)::
EMPILER(6)::
EMPILER(0)::
EMPILER(NIL)::
TRANSFERTZLA(3,7)::
AFFECTATION(3)::
EMPILERAD(9)::
EMPILERAD(10)::
COMPOSANT(3)::
VALEURPILE(1)::AFFECTATION(1)::
EMPILERAD(9)::
VALEURPILE(1)::EMPILERAD(10)::
EGAL::'SI'COMPARER'ALORS'
    'DEBUT''ALLERA'EA::
    'FIN'
'SINON'
    'DEBUT''FIN'
::
EA:LIBERERTYPE(7)::
'FIN'
```

Les instructions i gestion, i gestion, i système et départ correspondent respectivement :

- à l'initialisation du système de pages dans l'interpréteur
- à l'initialisation des variables de l'interpréteur
- à l'initialisation du programme de génération.

Etant donné que nous disposons de l'arbre complet de la chaîne source, nous n'avons utilisé ici qu'un seul chapitre d'entrée : celui correspondant à ⟨programme⟩. On aurait pu simuler le fonctionnement des chapitres d'entrées tels qu'ils sont décrits en III.3 en transformant le contenu de la pile fournie par l'analyse syntaxique en une liste donnant dans l'ordre convenu les arbres ayant des sommets correspondants.

2. Un interpréteur a été lui aussi écrit en ALGOL.

Il comprend principalement :

- des procédures effectuant la gestion du transfert des pages de mémoire lente (disque) en mémoire rapide ou vice-versa suivant le schéma suggéré en [co] .
- des procédures représentant les instructions objets
- des procédures effectuant une récupération et un retassement dans les circonstances critiques.

Dans les pages suivantes, on trouvera les configurations des pages correspondant aux nouveaux types nta et ntb aux principales étapes d'une récupération et d'un retassement au cours de l'exécution du programme objet suivant :

```
début
     (référence nta cna, référence entier cea, logique cla, logique clb) nta ;
     (entier ceb, référence nta cnb, logique clc) ntb ;
     entier eva ; référence nta nva ; référence ntb nvb ;
     eva := 1 ; nva := nil ;

ea : si eva < 21 alors
     début
          nvb := ntb(eva, nta(nil, eva, faux, faux), faux) ;
          nvb := ntb(eva, nva, faux) ;
          nva := nta(nva, ceb de nvb, vrai, vrai) ;
          eva := eva + 1 ; allera ea
     fin ;
          eva := 1 ;
eb : si eva < 20 alors
     début
          nva := nta(nva, eva, vrai, vrai) ;
          eva := eva + 1 ; allera eb
     fin
fin
```

On notera que tous les enregistrements restant valides sont ceux comportant des valeurs de composants égales à vrai.

Le nombre de pages total est 12, et le nombre de pages critique est fixé à 5. La dimension des pages est de 100 éléments.

Une première fois, une récupération est déclenchée à l'intérieur de la première boucle. Les pages appartenant aux types nta (d'adresse pile 9) et ntb (d'adresse pile 11) sont représentées par :

- le contenu de l'élément adressé par l'adresse pile
- le module des valeurs contenues dans les pages,
- puis pour chaque page :

    - l'adresse du premier élément
    - l'adresse de la page précédemment utilisée
    - enfin le contenu des éléments de la page.

99999 est la valeur de nihil.

On note, par exemple :

- en fin de marquage, l'enregistrement appartenant au type nta et d'adresse 514 n'est pas marqué : la valeur de l élément 513 est 0.
- en fin de récupération cet élément est placé dans une liste libre : à l'adresse 514 on trouve la valeur 524. Cette liste débute comme indiqué par le libellé "premier emplacement libre" à l'adresse 304.

Une seconde récupération est ensuite activée : elle n'est pas décrite ici.

Dans la seconde boucle du programme, un retassement est déclenché. Les enregistrements crées du type nta restent en effet tous valides. Les pages appartenant à ce type ne contiennent pas d'emplacements disponibles.

On note, par exemple :

- en début de retassement :

  - l'enregistrement de type ntb et d'adresse 652 n'est pas marqué
  - le deuxième composant de l'enregistrement de type nta et d'adresse 589 a pour valeur l'adresse 652

- en fin de remplissage, l'enregistrement d'adresse 652 est transféré à l'adresse 416.
- en fin de retassement, le deuxième composant de l'enregistrement 589 a pris pour valeur cette nouvelle adresse.

N.B On doit remarquer qu'en fait, en début de retassement tous les enregistrements relatifs au type nta sont démarqués : ils l'ont été au cours de la recherche d'une liste dans les pages correspondantes.

UNE PAGE EST DEMANDEE POUR LE TYPE: 9
LE NOMBRE DE PAGES LIBRES EST CRITIQUE
DECLENCHEMENT D UNE RECUPERATION

DEBUT DE RECUPERATION...

PAGES APPARTENANT AU TYPE: 9

PREMIER EMPLACEMENT LIBRE: 99999
TAILLE DES ENREGISTREMENTS: 5

ADRESSE DE LA PAGE: 501
ADRESSE DE LA PAGE PRECEDENTE: 301

	0	1	2	3	4	5	6
503	0	-1	13	0	0	0	394
510	484	1	1	0	-1	13	0
517	0	0	509	492	1	1	0
524	-1	13	0	0	0	519	604
531	1	1	0	-1	13	0	0
538	0	529	612	1	1	0	-1
545	13	0	0	0	539	620	1
552	1	0	-1	13	0	0	0
559	549	628	1	1	0	-1	13
566	0	0	0	559	636	1	1
573	0	-1	13	0	0	0	569
580	644	1	1	0	-1	13	0
587	0	0	579	652	1	1	0
594	-1	13	0	0	0	0	0

ADRESSE DE LA PAGE: 301
ADRESSE DE LA PAGE PRECEDENTE: 99999

	0	1	2	3	4	5	6
303	0	-1	-1	-1	-1	0	-1
310	13	0	0	0	-1	412	1
317	1	0	-1	13	0	0	0
324	314	420	1	1	0	-1	13
331	0	0	0	324	428	1	1
338	0	-1	13	0	0	0	334
345	436	1	1	0	-1	13	0
352	0	0	344	444	1	1	0
359	-1	13	0	0	0	354	492
366	1	1	0	-1	13	0	0
373	0	364	460	1	1	0	-1
380	13	0	0	0	374	468	1
387	1	0	-1	13	0	0	0
394	384	476	1	1	0	0	0

PAGES APPARTENANT AU TYPE: 11

PREMIER EMPLACEMENT LIBRE: 664
TAILLE DES ENREGISTREMENTS: 6

ADRESSE DE LA PAGE: 601
ADRESSE DE LA PAGE PRECEDENTE: 491

	0	1	2	3	4	5	6
603	0	12	519	0	0	13	534
610	0	0	13	529	0	0	14
617	546	0	0	14	539	0	0
624	18	554	0	0	15	549	0
631	0	16	564	0	0	16	589
638	0	0	17	574	0	0	17
645	569	0	0	18	584	0	0
652	18	579	0	0	19	594	0
659	0	19	589	0	0	668	0
666	0	0	672	0	0	0	676
673	0	0	0	680	0	0	0
680	684	0	0	0	688	0	0
687	0	692	0	0	0	696	0
694	0	0	99999	0	0	0	0

ADRESSE DE LA PAGE: 401
ADRESSE DE LA PAGE PRECEDENTE: 99999

	0	1	2	3	4	5	6
403	0	-1	-1	-1	0	1	309
410	0	0	1	-1	0	0	2
417	319	0	0	2	314	0	0
424	3	329	0	0	3	324	4
431	0	4	339	0	0	4	334
438	0	0	5	349	0	0	5
445	344	0	0	6	359	0	0
452	6	354	0	0	7	369	0
459	0	7	364	0	0	8	379
466	0	0	8	374	0	0	9
473	389	0	0	9	384	0	0
480	10	504	0	0	10	394	0
487	0	11	514	0	0	11	509
494	0	0	12	524	0	0	0

FIN DE MARQUAGE...

PAGES APPARTENANT AU TYPE: 9

PREMIER EMPLACEMENT LIBRE: 99999
TAILLE DES ENREGISTREMENTS: 5

ADRESSE DE LA PAGE: 501
ADRESSE DE LA PAGE PRECEDENTE: 301

	0	1	2	3	4	5	6
503	0	-1	13	0	0	1	-394
510	-484	1	1	0	-1	13	0
517	0	1	-509	-492	1	1	0
524	-1	13	0	0	1	-519	-604
531	1	1	0	-1	13	0	0
538	1	-529	-612	1	1	0	-1
545	13	0	0	1	-539	-620	1
552	1	0	-1	13	0	0	1
559	-549	-628	1	1	0	-1	13
566	0	0	1	-559	-636	1	1
573	0	-1	13	0	0	1	-569
580	-644	1	1	0	-1	13	0
587	0	1	-579	-652	1	1	0
594	-1	13	0	0	0	0	0

ADRESSE DE LA PAGE: 301
ADRESSE DE LA PAGE PRECEDENTE: 99999

	0	1	2	3	4	5	6
303	0	-1	-1	-1	-1	0	-1
310	13	0	0	1	-1	-412	1
317	1	0	-1	13	0	0	1
324	-314	-420	1	1	0	-1	13
331	0	0	1	-324	-428	1	1
338	0	-1	13	0	0	1	-334
345	-436	1	1	0	-1	13	0
352	0	1	-344	-444	1	1	0
359	-1	13	0	0	1	-354	-492
366	1	1	0	-1	13	0	0
373	1	-364	-460	1	1	0	-1
380	13	0	0	1	-374	-468	1
387	1	0	-1	13	0	0	1
394	-384	-476	1	1	0	0	0

PAGES APPARTENANT AU TYPE: 11

PREMIER EMPLACEMENT LIBRE: 664
TAILLE DES ENREGISTREMENTS: 4

ADRESSE DE LA PAGE: 601
ADRESSE DE LA PAGE PRECEDENTE: 401

	0	1	2	3	4	5	6
603	1	12	519	0	0	13	534
610	0	1	13	529	0	0	14
617	544	0	1	14	539	0	0
624	15	554	0	1	15	549	0
631	0	16	564	0	1	16	559
638	0	0	17	574	0	1	17
645	569	0	0	18	584	0	1
652	18	579	0	0	19	594	0
659	1	19	-589	0	0	668	0
666	0	0	672	0	0	0	676
673	0	0	0	680	0	0	0
680	684	0	0	0	688	0	0
687	0	692	0	0	0	696	0
694	0	0	99999	0	0	0	0

ADRESSE DE LA PAGE: 401
ADRESSE DE LA PAGE PRECEDENTE: 99999

	0	1	2	3	4	5	6
403	0	-1	-1	-1	0	1	309
410	0	1	1	-1	0	0	2
417	319	0	1	2	314	0	0
424	2	329	0	1	3	324	0
431	0	4	339	0	1	4	334
438	0	0	5	349	0	1	5
445	344	0	0	6	359	0	1
452	6	354	0	0	7	369	0
459	1	7	364	0	0	8	379
466	0	1	8	374	0	0	9
473	389	0	1	9	384	0	0
480	10	504	0	1	10	394	0
487	0	11	514	0	1	11	509
494	0	0	12	524	0	0	0

FIN DE RECUPERATION...

PAGES APPARTENANT AU TYPE: 9

PREMIER EMPLACEMENT LIBRE: 304
TAILLE DES ENREGISTREMENTS: 5

ADRESSE DE LA PAGE: 501
ADRESSE DE LA PAGE PRECEDENTE: 301

	0	1	2	3	4	5	6
503	0	514	13	0	0	0	394
510	484	1	1	0	524	13	0
517	0	0	509	492	1	1	0
524	534	13	0	0	0	519	0
531	1	1	0	544	13	0	604
538	0	529	612	1	1	0	0
545	13	0	0	0	539	620	554
552	1	0	564	13	0	0	1
559	549	628	1	1	0	574	0
566	0	0	0	559	636	1	13
573	0	584	13	0	0	0	1
580	644	1	1	0	594	13	569
587	0	0	579	652	1	1	0
594	99999	13	0	0	0	0	0

ADRESSE DE LA PAGE: 301
ADRESSE DE LA PAGE PRECEDENTE: 99999

	0	1	2	3	4	5	6
303	0	309	-1	-1	-1	0	319
310	13	0	0	0	1	412	1
317	1	0	329	13	0	0	0
324	314	420	1	1	0	339	13
331	0	0	0	324	428	1	1
338	0	349	13	0	0	0	334
345	436	1	1	0	359	13	0
352	0	0	344	444	1	1	0
359	369	13	0	0	0	354	452
366	1	1	0	379	13	0	0
373	0	369	460	1	1	0	389
380	13	0	0	0	374	468	1
387	1	0	504	13	0	0	0
394	384	476	1	1	0	0	0

PAGES APPARTENANT AU TYPE: 11

PREMIER EMPLACEMENT LIBRE: 404
TAILLE DES ENREGISTREMENTS: 4

ADRESSE DE LA PAGE: 601
ADRESSE DE LA PAGE PRECEDENTE: 401

	0	1	2	3	4	5	6
603	0	12	519	0	0	616	534
610	0	0	13	529	0	0	624
617	544	0	0	14	539	0	0
624	632	554	0	0	15	549	0
631	0	648	564	0	0	16	559
638	0	0	648	574	0	0	17
645	569	0	0	656	584	0	0
652	18	579	0	0	664	594	0
659	0	19	589	0	0	668	0
666	0	0	672	0	0	0	676
673	0	0	0	680	0	0	0
680	684	0	0	0	688	0	0
687	0	692	0	0	0	696	0
694	0	0	99999	0	0	0	0

ADRESSE DE LA PAGE: 401
ADRESSE DE LA PAGE PRECEDENTE: 99999

	0	1	2	3	4	5	6
403	0	408	-1	-1	0	416	309
410	0	0	1	1	0	0	424
417	319	0	0	2	314	0	0
424	432	329	0	0	3	324	0
431	0	440	339	0	0	4	334
438	0	0	448	349	0	0	5
445	344	0	0	456	359	0	0
452	6	354	0	0	464	369	0
459	0	7	364	0	0	472	379
466	0	0	8	374	0	0	480
473	389	0	0	9	384	0	0
480	488	504	0	0	10	394	0
487	0	496	514	0	0	11	509
494	0	0	608	524	0	0	0

UNE PAGE EST DEMANDEE POUR LE TYPE: 9
LE NOMBRE DE PAGES LIBRES EST CRITIQUE
DECLENCHEMENT D UNE RECUPERATION

DEBUT DE RECUPERATION...

FIN DE MARQUAGE...

FIN DE RECUPERATION...

UNE PAGE EST DEMANDEE POUR LE TYPE: 9
LE NOMBRE DE PAGES LIBRES EST CRITIQUE
DECLENCHEMENT D UNE RECUPERATION

DEBUT DE RECUPERATION...

FIN DE MARQUAGE...

RECUPERATION SANS SUCCES
UN RETASSEMENT S IMPOSE.

DEBUT DE RETASSEMENT...

PAGES APPARTENANT AU TYPE: 9

PREMIER EMPLACEMENT LIBRE: 99999
TAILLE DES ENREGISTREMENTS: 5

ADRESSE DE LA PAGE: 501
ADRESSE DE LA PAGE PRECEDENTE: 301

	0	1	2	3	4	5	6
503	1	389	13	1	1	1	394
510	484	1	1	1	504	13	1
517	1	1	509	492	1	1	1
524	514	13	1	1	1	519	604
531	1	1	1	524	13	1	1
538	1	529	612	1	1	1	524
545	13	1	1	1	539	620	1
552	1	1	544	13	1	1	1
559	549	628	1	1	1	554	13
566	1	1	1	559	636	1	1
573	1	564	13	1	1	1	569
580	644	1	1	1	574	13	1
587	1	1	579	652	1	1	1
594	584	13	1	1	0	0	0

ADRESSE DE LA PAGE: 301
ADRESSE DE LA PAGE PRECEDENTE: 99999

	0	1	2	3	4	5	6
303	1	589	660	1	1	1	594
310	13	1	1	1	1	412	1
317	1	1	304	408	1	1	1
324	314	420	1	1	1	319	13
331	1	1	1	324	428	1	1
338	1	329	13	1	1	1	334
345	436	1	1	1	339	13	1
352	1	1	344	444	1	1	1
359	349	13	1	1	1	354	452
366	1	1	1	359	13	1	1
373	1	364	460	1	1	1	369
380	13	1	1	1	374	468	1
387	1	1	379	13	1	1	1
394	384	476	1	1	0	0	0

PAGES APPARTENANT AU TYPE: 11

PREMIER EMPLACEMENT LIBRE: 404
TAILLE DES ENREGISTREMENTS: 4

ADRESSE DE LA PAGE: 601
ADRESSE DE LA PAGE PRECEDENTE: 401

	0	1	2	3	4	5	6
603	1	12	519	0	0	616	534
610	0	1	13	529	0	0	624
617	544	0	1	14	539	0	0
624	632	554	0	1	15	549	0
631	0	640	564	0	1	16	559
638	0	0	648	574	0	1	17
645	569	0	0	658	584	0	1
652	18	579	0	0	664	594	0
659	1	19	589	0	0	668	0
666	0	0	672	0	0	0	676
673	0	0	0	680	0	0	0
680	684	0	0	0	688	0	0
687	0	692	0	0	0	696	0
694	0	0	99999	0	0	0	0

ADRESSE DE LA PAGE: 401
ADRESSE DE LA PAGE PRECEDENTE: 99999

	0	1	2	3	4	5	6
403	0	416	309	0	1	20	-304
410	0	1	1	1	0	0	424
417	319	0	1	2	314	0	0
424	432	329	0	1	3	324	0
431	0	440	339	0	1	4	334
438	0	0	448	349	0	1	5
445	344	0	0	458	359	0	1
452	6	354	0	0	464	369	0
459	1	7	364	0	0	472	319
466	0	1	8	374	0	0	480
473	389	0	1	9	384	0	0
480	488	504	0	1	10	394	0
487	0	496	514	0	1	11	509
494	0	0	608	524	0	0	0

ZONE RETASSEE POUR LE TYPE: 9

FIN DE REMPLISSAGE...

PAGES APPARTENANT AU TYPE: 11

PREMIER EMPLACEMENT LIBRE: 484
TAILLE DES ENREGISTREMENTS: 4

ADRESSE DE LA PAGE: 601
ADRESSE DE LA PAGE PRECEDENTE: 401

	0	1	2	3	4	5	6
603	0	464	465	466	0	616	534
610	0	0	456	457	458	0	624
617	544	0	0	448	449	450	0
624	632	554	0	0	440	441	442
631	0	640	564	0	0	432	433
638	434	0	648	574	0	0	424
645	425	426	2	656	584	0	0
652	416	417	418	0	664	594	0
659	0	404	405	406	0	668	0
666	0	0	672	0	0	0	676
673	0	0	0	680	0	0	0
680	684	0	0	0	688	0	0
687	0	692	0	0	0	696	0
694	0	0	99999	0	0	0	0

ADRESSE DE LA PAGE: 401
ADRESSE DE LA PAGE PRECEDENTE: 99999

	0	1	2	3	4	5	6
403	0	19	589	0	0	20	304
410	0	0	1	1	0	0	18
417	579	0	0	2	314	0	0
424	17	569	0	0	3	324	0
431	0	16	559	0	0	4	334
438	0	0	15	549	0	0	5
445	344	0	0	14	539	0	0
452	6	354	0	0	13	529	0
459	0	7	364	0	0	12	519
466	0	0	8	374	0	0	11
473	509	0	0	9	384	0	0
480	10	394	0	0	488	481	482
487	0	496	514	0	0	472	473
494	474	0	608	524	0	0	0

FIN DE RETASSEMENT...

PAGES APPARTENANT AU TYPE: 9

PREMIER EMPLACEMENT LIBRE: 99999
TAILLE DES ENREGISTREMENTS: 5

ADRESSE DE LA PAGE: 501
ADRESSE DE LA PAGE PRECEDENTE: 301

	0	1	2	3	4	5	6
503	0	389	13	1	1	0	394
510	480	1	1	0	504	13	1
517	1	0	509	472	1	1	0
524	514	13	1	1	0	519	464
531	1	1	0	524	13	1	1
538	0	529	456	1	1	0	534
545	13	1	1	0	539	448	1
552	1	0	544	13	1	1	0
559	549	440	1	1	0	554	13
566	1	1	0	559	432	1	1
573	0	564	13	1	1	0	569
580	424	1	1	0	574	13	1
587	1	0	579	416	1	1	0
594	584	13	1	1	0	0	0

ADRESSE DE LA PAGE: 301
ADRESSE DE LA PAGE PRECEDENTE: 99999

	0	1	2	3	4	5	6
303	0	389	404	1	1	0	594
310	13	1	1	0	1	412	1
317	1	0	304	408	1	1	0
324	314	420	1	1	0	319	13
331	1	1	0	324	428	1	1
338	0	329	13	1	1	0	334
345	436	1	1	0	339	13	1
352	1	0	344	444	1	1	0
359	349	13	1	1	0	354	452
366	1	1	0	359	13	1	1
373	0	364	460	1	1	0	369
380	13	1	1	0	374	468	1
387	1	0	379	13	1	1	0
394	304	476	1	1	0	0	0

PAGES APPARTENANT AU TYPE: 11

PREMIER EMPLACEMENT LIBRE: 484
TAILLE DES ENREGISTREMENTS: 4

ADRESSE DE LA PAGE: 401
ADRESSE DE LA PAGE PRECEDENTE: 99999

	0	1	2	3	4	5	6
403	0	19	589	0	0	20	304
410	0	0	1	1	0	0	18
417	579	0	0	2	314	0	0
424	17	569	0	0	3	324	0
431	0	16	559	0	0	4	334
438	0	0	15	549	0	0	5
445	344	0	0	14	539	0	0
452	6	354	0	0	13	529	0
459	0	7	364	0	0	12	519
466	0	0	8	374	0	0	11
473	509	0	0	9	384	0	0
480	10	394	0	0	488	481	482
487	0	492	514	0	0	496	473
494	474	0	608	524	0	0	0

IV CONCLUSION.

Nous avons déjà examiné en II.2.5 les principales caractéristiques du langage de génération proposé. L'usage que nous en avons fait pour traiter différents problèmes (II.2.3. et III.3) conduit à envisager plusieurs extensions possibles :

- introduction de descriptions "locales". On remarque en effet qu'on utilise souvent des descriptions comme des piles. Par exemple, en III.3.1 la description "types valides" joue ce rôle : son premier élément contient la liste de tous les nouveaux types déclarés dans un bloc.
- introduction de procédures. On aurait pu penser que dans chaque chapître les actions effectuées, correspondant à des noms de noeuds d'arbres distincts soient essentiellement différentes. En fait, nous avons remarqué au cours du chapître III que dans de nombreux chapîtres on opère la réduction du niveau d'une variable à un niveau inférieur. Ainsi, l'action du chapître ⟨générateur d'**enregistrement**⟩ est en partie analogue à celle effectuée dans le chapître ⟨instruction d'affectation⟩ en ce qui concerne une "préparation à l'affectation". Nous avons d'ailleurs utilisé des procédures pour de telles parties communes dans le compilateur réalisé en ALGOL 60.
- enfin, nous avons vu en II.2.3.2. l'intérêt de disposer de communications avec l'analyse : dans un premier stade, on pourrait envisager d'introduire en cours d'exécution d'un programme de génération des règles de grammaires lexicographiques.

En résumé, ce langage nous paraît particulièrement intéressant pour concevoir et décrire des algorithmes de compilation.

D'une part sa facilité d'écriture et les possibilités de "réglage" des points de génération peuvent permettre de faire le "poids" d'une optimisation, c'est-à-dire d'estimer la complexité résultante du compilateur.

D'autre part, sa souplesse d'adaptation permettra d'effectuer aisément les transformations dues à des modifications limitées du langage objet ou du langage source.

Ainsi, en ce qui concerne le langage que nous avons considéré en III il faudrait, pour traiter les problèmes posés par les nouveaux types et les variables référence$_s$ tels qu'ils sont définis dans[VW2] :

- introduire des composants de type "tableau : ceci impliquerait essentiellement que la taille des valeurs nouvelles les contenant ne soit déterminée qu'à l'exécution
- remplacer les variables libres par des variables d'autres types eux aussi déclarés dans le programme ceci rendant incontestablement plus "orthogonal" [VW2] le langage et définis comme des "unions" d'autres types, En particulier, la valeur n-1 terminale d'une telle variable de niveau n($n>1$)est toujours du type de cette variable et ainsi est-il possible d'appliquer l'opérateur valeur à cette dernière : ceci rendra inutile l'instruction "valeur libre" telle que nous l'avons introduite dans le langage objet.

Les importantes questions relatives à la récupération et au retassement ont été traitées ici dans une optique particulière : division en page de la mémoire et un seul élément réservé pour marquer un enregistrement. Il est bien évident que ces questions ne peuvent être résolues qu'en fonction du calculateur considéré. On aura certes toujours la possibilité de rendre négatives les valeurs de niveau supérieur à zéro, toutefois la disposition d'un "bit" spécial par mémoire et réservé au marquage ne manquerait pas de faciliter le travail. L'adoption ou non d'une zone de liste en pages dépendra aussi de la structure technologique du calculateur.

Enfin, entre la représentation des variables que nous avons adoptée, qui est relativement condensée, et une représentation de celles-ci à l'aide de listes LISP, il existe d'autres solutions envisageables. Nous pensions ici en particulier à une représentation des variables déclarées à l'aide d'un élément de pile contenant une adresse désignant la valeur attribuée à cette variable. Ceci permettrait en fin de validité de cette dernière de ne pas détruire la valeur attribuée.
(cf. problèmes relatifs à l'introduction du symbole "local" dans [VW2]).

Pour terminer nous devons formuler deux remarques importantes à propos de ce travail :

(1) L'intérêt d'un langage de génération pour l'écriture effective des compilateurs ne pourra être démontré qu'en réalisant une implantation aussi efficace que possible d'un tel langage et du système qui l'environne.

(2) Les aspects d'ALGOL X considérés au chapitre III sont loin de constituer l'unique source de problèmes posés par la compilation de ce langage.

En ce sens, on peut considérer l'étude présentée comme un point de départ pour des recherches, et des réalisations plus vastes dans ces domaines.

B I B L I O G R A P H I E

Bou J.C BOUSSARD, Etude et réalisation d'un compilateur ALGOL 60 sur calculatrice électronique du type IBM 7090/94 et 7040/44, thèse, Université de Grenoble, Juin 1964.

Br R.B et al, The compiler compiler, third annual rewiew of automatic programming, pergamon press, 1963

BCo M. BRASSEUR et J. COHEN, Algorithmes d'analyse syntaxique pour langages context-free, chiffres volume 8, numéro 2 et 3, 1965.

Co J. COHEN, Langages pour l'écriture des compilateurs, thèse, Université de Grenoble, Juin 1967.

CON J. COHEN, et Nguyen Huu Dung, Définition de procédures LISP en ALGOL, exemples d'utilisation, R.F.T.I., chiffres, volume 8, numéro 4, 1965.

Co T1 J. COHEN, L. TRILLING. Description d'un transducteur traduisant en langage machine une chaîne sous forme préfixée, communication Congrès A.F.I.R.O, Juin 1966.

Co T2 J. COHEN, L. TRILLING, Remarks on "Gabarge Collection" using a two level storage, BIT 7,1, 1967.

Col A. COLMERAUER. Précédence, analyse syntaxique et langages de programmation, thèse, Université de Grenoble, Septembre 1967.

Cou J. COURTIN, étude des langages analysables de gauche à droite et applications, thèse Université de Grenoble -à paraître-

CGr I Currie, M. GRIFFITHS, A self-transferning compiler, RRE Mémo n° 2358, Février 1967.

Ch T. E. CHEATHAM, J.R, The introduction of définitional facilities into higher level programming languages, AFIPS, Fall Joint Computer Conference, 1966.

Fl J. FELDMAN, A formal semantics for computer languages and its application in a compiler-compiler, C A C M , Janvier 1966.

Gre S. GREIBACH, Formal parsing systems, communication A.C.M, Août 1964.

GP T.H. GRIFFITHS, et S.R PETRICK, On the relative efficiencies of context-free grammar recognizers, comm A.C.M, Mai 1965.

Hoq C. A.R. HOARE, Record hardling Summer Nato School, Septembre 1966.

Ir E.T. IRONS, A syntax-directed compiler for ALGOL 60, CACM 4, 1961.

Iso International Standards Organization, Survey of Programming Languages and Processors, CACM 6, 1963.

Jor P. JORRAND, An edition language, note technique, Laboratoire de Calcul de Grenoble, Août 1967.

Kn D.E KNUTH, on the translation of languages from left to right, Inf. And Control, 8, 1965.

Kuo S. KUNO, A.G OETTINGER, Multiple Path syntactic Analyser, Mathematical, Linguistics and Automatic Translation, Harward, University, 1963.

Le P J. LE PALMEC, Etude d'un langage intermédiaire pour la compilation d'ALGOL 60, Thèse Université de Grenoble, JUIN 1966;

Mar F. MARTIN, Détermination de certaines caractéristiques des grammaires et langages C.F., thèse Université de Grenoble. - à paraître-

McC J. Mc CARTHY, et al, Lisp 1.5 Programmer's Manual, The Mit Press, Cambridge, Mass. 1962.

Na P. NAUR, The design of the Gier ALGOL compiler, Nordisk Tidskrift for Information Beharding, BIT 3. 1963.

Nau P. NAUR et al. Revised report on the algorithmic language ALGOL 60, Comm ACM, vol. 6, Janvier 1963.

Sh SHOW, Christopher J. A specification of jovial. C.A.C.M. Juin 1963.

ShF M. SHOW, J. FIERST, Carnegie Institute of Technology, Technical Memo CCU-51, Janvier 1967.

Tas
Note technique I.M.A.G., -à paraître-

VVV G. VEILLON, J. VEYRUNES, B. VAUQUOIS, un métalange de grammaires transformationnelles, C.E.T.A. G. 2300-A, Janvier 1967.

Was S, WARSCHALL, et R.M SCHAPIRO, A Genral-Purpose Table-Driven Compiler, Proceedings Spring Joint Computer Conference, A.F.I.P.S, 1964.

WiH N. WIRTH, CAR HOARE, A contribution to the development of ALGOL, Comm A.C.M, Janvier 1966.

Woo P.M WOODWARD, A note on Foster's syntax Improving Syntax, RRE Memo 2352.

VW1 A. VAN WIJNGAARDEN, Rapport préliminaire ALGOL X, Varsovie, Octobre 1966.

VW2 A. VAN WIJNGAARDEN, Draft Proposal for the Algorithmic Language ALGOL 67, Zandvoort, Mai 1967.

VU

Grenoble, le

Le Président de la Thèse

VU

Grenoble, le

Le Doyen de la Faculté des Sciences

VU, et permis d'imprimer,

Le Recteur de l'Académie de GRENOBLE